LONDON'S GLOBAL UNIVERSITY
Department of Computer Science

A Digital Contract Model Representing Smart Computable Contracts as Petri Nets using Controlled Natural Language

Dominic Kloecker

Supervisor: Professor Christopher Clack

Faculty of Engineering

Department of Computer Science

University College London

This report is submitted as part requirement for the
MSc Computer Science degree at UCL.

Monday 11th September, 2023

Abstract

Amid the developments towards smart contracts and legal digitisation, much attention has been given to automating contractual performance through the use of tools such as smart contracts. However, the foundational necessity to first develop a comprehensive digital model of a legal agreement has been largely overlooked. Analogous to Computer Aided Design (CAD) in engineering, where components are defined by a series of interconnected coordinates forming a foundational model for subsequent analysis, this paper aims to create a model capable of capturing the foundational geometry of a contract. By expressing a contract in a Controlled Natural Language (CNL) such a geometry can be represented in the form of a Petri Net, facilitating performance simulation and providing a clear feedback on contractual expectations.

A literature review on the state of computable contracting and the advancements towards digital representation of contractual agreements was compiled. On the basis of the literature, a logical model and formal syntax for representing a contract as a Petri Net was developed. Syntax translators that facilitate the conversion of two CNLs into the proposed formal syntax were constructed. This translation process enables the novel ability to automatically generate a Petri Net capable of representing a contract and simulating its performance under various scenarios.

By expressing contracts as Petri Nets, the model was able to enhance the clarity and traceability of contractual obligations and conditions. Furthermore, the representation could offer a significant advantage in the identification of "legal bugs", consisting of subtle miss steps in phrasing or wording that can significantly alter the underlying logic of a contract. Although such bugs might escape notice in a textual format, their visual representation through Petri Nets makes anomalies more discernible.

Keywords— Contract - Petri Net - Lexon - CoLa

The life of the law has not been logic; it has been experience.

— Oliver Wendell Holmes Jr.

Acknowledgements

I would like to extend my deepest gratitude to Professor Christopher D. Clack, not only for his invaluable guidance throughout this project, but also for introducing me to the fascinating world of functional programming, which I have come to deeply appreciate and enjoy.

My heartfelt thanks go to my incredibly supportive family who have stood by me throughout this year. Special thanks to my grandparents, Heinz and Rosemarie, my mother Astrid, and my siblings Constantin and Isabelle.

Lastly, I would like to thank all my friends who have been pillars of support this year. A particular mention goes to Stefan Strat, Tobias Klaus, Ashley Ip and Zain Jaffal, all of whom were instrumental in inspiring me to pursue the field of computer science.

Table of Contents

List of Figures

List of Listings

List of Abbreviations

AST: Abstract Syntax Tree

CAD: Computer Aided Design

CNL: Controlled Natural Language

DFA: Deterministic Finite Automata

DSL: Domain Specific Language

DSPL: Domain Specific Programming Language

LSP: Legal Specification Protocol

1 | Introduction

1.1 Smart and Computable Contracting

In recent decades, nearly all industries have undergone radical transformations due to the digital representation of their core products. However, the core product of the legal profession, a contractual agreement between two or more parties exchanging goods or services, has remained unchanged for the most part, remaining a document expressed in natural language [6].

The potential benefits of a new paradigm concerning the expression and formatting of computable contracts are vast, including applications such as automated performance or inconsistency analysis of contracts. However, such a paradigm shift comes with a multitude of unique challenges, such as the lack of a common consensus regrading the definition and scope of Smart Contracts [7]. This confusion can be partially attributed to the rise of blockchain-related technologies, like Ethereum, that define the operational aspects of digital agreements. Here, software agents take ownership or control of assets within a shared ledger [7].

Another significant challenge is the current lack of a mature language capable of fully expressing a legal agreement in a way that is human and computer readable. Several languages, in the form of Controlled Natrual Languages (CNL) and Domain Specific Languages (DSL), are being developed to bridge this research gap. Comparisons of several such languages by Idelberger [8] and analysis of the challenges any such language faces by Clack [9] have found the existing languages to offer promising glimpses into the future of computable contracting, but that they are yet in their developing stages and too immature for large scale adoption as of now.

However, languages such as Lexon [10] and CoLa [2] can already be used to highlight the potential benefits that a computable contracting future can hold. This project will use these languages, as input sources in Section 4, to produce a logical contract model that facilitates deeper analysis.

1.2 The Logical Contract Model

A contract can be viewed as an event processing machine, detailing the expectations of parties under various possible event sequences. Thus, a contract should be expressible as a logical model, detailing its structure and allowing reasoning about the clauses and expectations.

Flood and Goodenough [11], proposed that a well-written financial agreement should follow a state transition logic, allowing its mathematical formalisation. Research highlighted the relative simplicity of the computational logic involved in such an agreement by manually converting and representing a simple loan agreement as a Deterministic Finite Automaton (DFA).

A different logical model for electronic contracting, representing contracts as Petri Nets, was proposed by Lee [1]. While only simple agreements were expressed in this logical model, the methodology was successful in providing a graphical representation of the contracts as a Petri Net, from which further reasoning could be done. However, this Petri Net did not represent the contract in an accessible manner that could provide insight on its preformative state.

These past attempts have highlighted the benefits of a theoretical and logical contractual model but have shown limitations in their application. Based on Lee's electronic contract model, this paper will investigate the development of a new logical mode in Seciton 4, with the aim of improving the flexibility and expressability of the contract as a Petri Net.

1.3 Project Aims

The primary aim of this project is to develop a logical model of a legal agreement expressed in a Controlled Natural Language (CNL), using Petri Nets.

1. Merge the domain of Controlled Natural Language and Logical contract modelling.
2. Develop a methodology to faithfully represent a contract as a Petri Net.
3. Develop a methodology to convert contracts written in a CNL into a Petri Net.

4. Investigate the ability to simulate contract performance using the Petri Net representation of a contract.

1.4 Objectives

The objectives of the project are the specific and measurable tasks that must be completed to achieve the broader aims outlined above. This section details each of these objectives, providing a clear roadmap for the project's development.

1. Conduct a literature review on the state of computable contracting and languages used for it.
2. Develop a formal intermediate syntax that can express contracts written in CoLa and Lexon, serving as a bridge between CNLs and Petri Nets.
3. Develop a methodology to generate the intermediate syntax representation of a contract from the Lexon and CoLa Abstract Syntax Trees (ASTs).
4. Develop a methodology to convert the intermediate syntax representation of a contract into a Petri Net, enabling a graphical and logical representation of the contract.
5. Convert multiple contracts written in CoLa and Lexon into Petri Nets using the developed methodology to validate its effectiveness and flexibility.
6. Develop a methodology to simulate contractual performance based on the Petri Net representation of the contract.
7. Evaluate the simulated contract performance and the overall effectiveness of the developed methodology in enabling reasoning and analysis on contracts.

2 | Background

2.1 Controlled Natural Languages for Contracting

Controlled natural languages allow for the flexibility inherent in standard English while being sufficiently structured and restricted to be convertible into computer code [9]. This allows the contract to simultaneously act as its own source code, eliminating the need to draft a contract in English by a lawyer and then translate it into code by a programmer, where a significant risk of unsuccessful conversion can occur.

2.1.1 Lexon

Lexon is a highly readable CNL developed by H. Diedrich [10]. It facilitates the creation of contracts that are both computer-executable and human-readable. A layperson without programming experience, such as a lawyer, can easily understand a contract written in Lexon, while it can also be directly compiled into the respective Solidity and Sophia code, ready for deployment on the Blockchain.

An example of a Lexon contract and the corresponding Solidity Code can be seen in Listing 1 and Listing 2.

```
LEX Paid Escrow.
LEXON: 0.2.12
'Payer' is a person.
'Payee'is a person.
'Arbiter' is a person.
'Fee' is an amount.
The Payer pays an Amount into escrow,
appoints the Payee,
appoints the Arbiter,
and also fixes the Fee.
CLAUSE: Pay Out.
The Arbiter may pay from escrow the Fee to themselves,
and afterwards pay the remainder of the escrow to the Payee.
CLAUSE: Pay Back.
The Arbiter may pay from escrow the Fee to themselves,
and afterwards return the remainder of the escrow to the Payer.
```

Listing 1: Lexon contract detailing an Escrow controlled by a third party [3].

```
pragma solidity ^0.5.0;
contract PaidEscrow{
    constructor(address payable _payee, address payable _arbiter, uint _fee)
    public payable {
        payer=msg.sender;
        amount=msg.value;
        escrow.transfer(amount);
        payee=_payee;
        arbiter=_arbiter;
        fee=_fee;
    }
    function Pay_Out() external {
        if (msg.sender == arbiter){
            arbiter.transfer(fee);
            payee.transfer(address(this).balance);
        }
        else{
            require(false);
        }
    }
    function Pay_Back() external {
        if (msg.sender == arbiter){
            arbiter.transfer(fee);
            payer.transfer(address(this).balance);
        }
        else{
            require(false);
        }
    }
 }
```

Listing 2: Solidity code of Escrow contract in Listing 1.

Lexon's fundamental grammar closely mirrors that of natural English, adhering to the order of: Subject, Verb, Object [12]. In this structure, verbs and the object are grouped together into a predicate. The use of nouns is highly flexible, permitting terms such as "Payer", "Person", "A", "John Smith" etc. Unlike nouns however verbs such as "pay" or "request" are restricted in their use.

Lexon's design is focused on the creation of machine-executable contracts. This design focus makes expressing concepts outside the blockchain realm challenging and cumbersome, making Lexon unsuitable at expressing most legal agreements.

Consider a tenancy agreement requiring a tenant to maintain the property or a Landlord to provide the tenant with furniture for the property. Services like property maintenance cannot be independently verified on the Blockchain, and likewise the actual asset, the property, is not inherent to the Blockchain. Although expressing these types of obligations is not impossible in Lexon, it is an intricate process that requires clauses in which parties can certify obligations are fulfilled and define consequences if not.

Additionally, the nature of Lexon restricts it's expressive potential in comparison to normal legal contracts. In conventional legal agreements, the contract sets a structure and a set of boundaries over what is expected of each party without providing a step-by-step guide on how each party should meet its expectations. This approach allows the contract to describe a vast array of possible performances, potentially allowing room for flexibility based on the intentions and interpretations of circumstances. This is needed, as it is either impossible or unfeasible to describe the contractual expectations under all possible scenarios that could occur.

In contrast, Lexon is designed to describe a singular specific performance of a contract in a clear step-by-step manner. As such, Lexon describes the performance of an agreement more than a contract in itself. This has a significant impact on the design of Lexon and what a contract written in it can describe. Due to this, the concept of prohibitions, what one is forbidden from doing, is redundant in Lexon. There is simply no need to describe what is prohibited, as the step-by-step description of the performance inherently ensures that anything that is not specifically permitted simply cannot occur. In the legal realm, however, the ability to express these prohibitions is invaluable, as they serve as clear boundaries and provide guidance to the parties.

2.1.2 CoLa

CoLa, the Controlled Natural Language for Computable Contracting was developed by S. Fattal for a Master's Thesis [2]. Unlike Lexon, CoLa is focused on the digital representation of a legal contract and does not cover the aspect of automated performance or code generation.

CoLa allows for the expression of the deontic elements of a contract, capturing obligations, permissions and prohibitions. Cola's syntax additionally allows for the ability of "priority clauses", that is the expression of a statement that supersedes another under certain conditions. Furthermore, CoLa's ability to produce a parse tree of the contract allows for relatively easy identification of all relevant components relating to a statement.

```
IF   [1] it is the case that PartyA shall pay AMOUNT 'A' on ADATE
         AND
     [2] it is the case that PartyB shall pay AMOUNT 'B' on THEDATE
THEN[3] it is not the case that PartyA shall pay AMOUNT 'A' on THEDATE
         AND
     [4] it is not the case that PartyB shall pay AMOUNT 'B' on THEDATE
C-AND
     [5] it is the case that ExcessParty shall pay AMOUNT "ExcessAmount" on THEDATE
C-AND
IF   [6] it is the case that PartyA paid more than PartyB
THEN[7] ExcessParty IS PartyA
         AND
     [8] ExcessAmount EQUALS AMOUNT 'A' MINUS AMOUNT 'B'
C-AND
IF   [9] it is the case that PartyB paid more than PartyA
THEN[10]ExcessParty IS PartyB
         AND
     [11]ExcessAmount EQUALS AMOUNT 'B' MINUS AMOUNT 'A'.
```

Listing 3: CoLa contract detailing the ISDA Master Agreement [2].

Although CoLa offers a human-readable format, it is less intuitive than Lexon for the lay person that is not familiar with the language. At times, it can also be difficult to discern between statements and conditions, an example of which are the components [1] and [4] in Listing 3. Nonetheless, CoLa's ability to logically express deontic statements of a legal agreement under clearly laid out conditions is powerful.

Owing to the nature of being a Masters research Project, it is not yet fully mature, and its syntax too limited to allow for the expression of many essential concepts. For instance, unlike Lexon, CoLa currently does not allow the specification of a secondary subject in a statement such as *"PartyA shall pay Amount A to Party B"*. Although one might argue that this is implied within the contract's context, it limits CoLa's ability to

express contracts that concern multiple stakeholders.

Moving forward, the available version of CoLa was a slight iteration before the finished version in Fattal's thesis. Although their functionalities are nearly indistinguishable, this preliminary CoLa version does not allow the expression of ambiguous dates like "ADATE". Therefore, it required the explicit specification of a date for any statement. This limitation was deemed acceptable since it neither altered the contract's structure or spirit, nor the inherent logic of CoLa.

2.2 Petri Net

The Petri Net, a type of state machine capable of describing the flow of information of complex and recursive systems, was developed by Carl Adam Petri in 1966 [13].
The basic Petri Net, is a weighted directed graph, of two types of nodes, place nodes and transition nodes, connected through weighted arcs. place nodes, as shown in Figure 2.1, describe the state or substate of a system and may hold an unlimited number of tokens. These tokens, represented as black dots, signify the value of the state or substates. The Petri Net combines the features of a PERT diagram, useful for capturing the relative timing of parallel or sequentially occurring activities, and decision trees that can display the set of available choices of actions given a sequence of events [1].

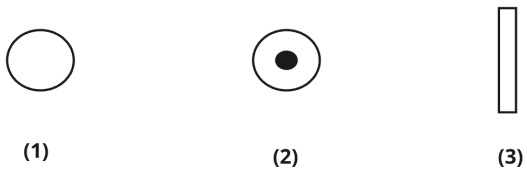


Figure 2.1: Components of Petri Net (1) place node with no token; (2) place node with one token; (3) transition node

Transition nodes do not hold tokens, but rather symbolise actions or events that cause a transition between states. A transition becomes enabled when each of its input place nodes has tokens equal to or greater than the weight of the connecting arc. Once a transition is enabled, it can fire at any time, referring to the process of transferring

tokens from the input place nodes of the transition node to the output place nodes, according to the weight of the connecting arcs [14], as illustrated in Figure 2.2.

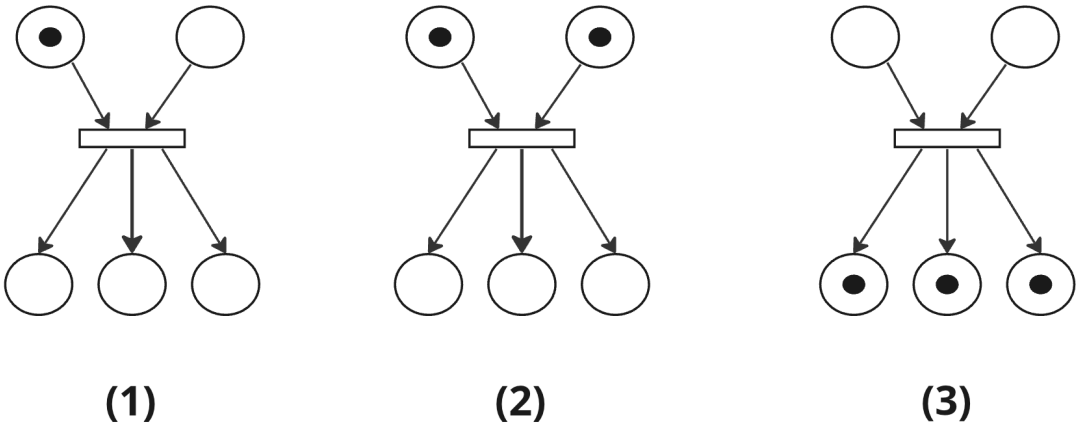


Figure 2.2: Illustration of Petri Net: (1) transition node not enabled; (2) transition node enabled, before firing; (3) transition node, no longer enabled after firing

Since its inception, there have been several extensions to Petri Nets. Two notable extensions include the introduction of *read arcs* and *inhibitor arcs*. These allow a transition to test for the presence or absence of tokens at place nodes, without influencing the place node upon firing [15], as illustrated in Figure 2.3.

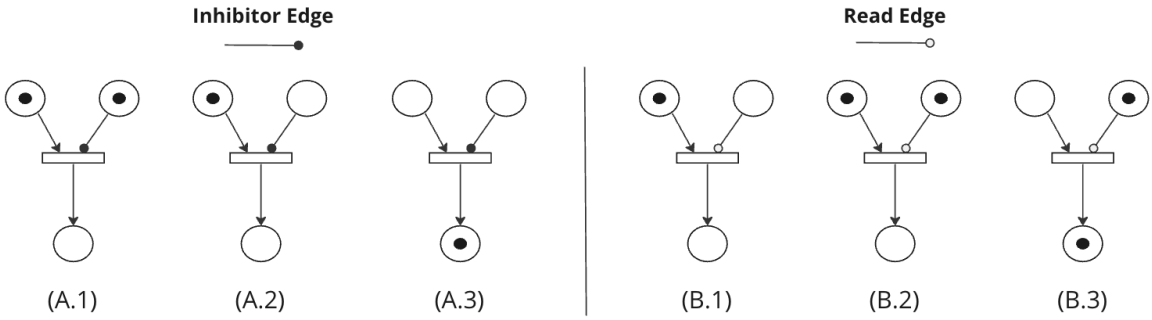


Figure 2.3: (A.1) Non enabled transition node with inhibitor arc; (A.2) Enabled transition node with inhibitor arc before firing; (A.3) transition node with inhibitor arc after firing. (B.1) Non enabled transition node with read arc; (B.2) Enabled transition node with read arc before firing; (B.3) transition node with read arc after firing.

2.3 Literature Review

Clack et al.[7] provided insight into the paradigm of smart contract templates, exploring the semantics, parameters, and common terminology of smart contracts. The paper proposed a semantic framework encompassing operational and nonoperational aspects of legally enforceable smart contracts, as well as a high-level definition of the term "smart contract". The work serves as a research compass, highlighting various research directions, such as the potential development of a formal language aimed at writing smart contracts.

A new framework merging a web-based interface with a Python API, facilitating the development of machine-readable contracts through human and machine-readable legal language repositories [16]. The Z3 prover developed by Microsoft Research was integrated into the framework, enabling the formal verification and validation of agreements. The research analysed a number of sample agreements, identifying 21 unique legal actions, 17 of which could be incorporated into the framework. The paper noted that the framework thus far lacks the ability to "cleanly" represent recurring events. While appearing promising, neither the implementation details of the framework nor the identified actions were fully detailed, suggesting the need for further investigation.

R. Kowalski [17] investigated the ability and practicality of representing legislation as logic programmes. Through the analysis of various pieces of legal language such as the British Nationality Act or the University of Michigan lease termination clause, Kowalski managed to successfully articulate these agreements as logical statements.

Flood and Goodenough [11] found that a Deterministic Finite Automata (DFA) was able to accurately represent the intent and logic behind a well written financial contract. They successfully represented the logic of a simple Loan Agreement as a DFA, noting that while the DFA representation may be a useful starting point, it is likely not a practical approach. The paper concluded that due to the inherent limitations of a DFA, the representation of more complex legal agreements requires an expansion of the pure DFA or use a Nondeterministic Finite Automata.

R. Lee [1] proposed a logical programming formulation based on the combination of Von

Wright's 'logic of change' and other logical expression frameworks such as deontic logic and a modified version of the temporal logic system developed by Rescher and Urquhart. Based on this logical programming formulation, Lee modelled a contract as a Petri Net where events and actions are specified as attributes of transition nodes. Lee describes the substate of a contract system to be a conjunction of elementary literals which depict either a proposition or its negation. Various simple contracts were expressed through the formulation developed, demonstrating that they can be represented as Petri Nets. Lee subsequently developed a Natural Language interface using Prolog which was able to provide answers to performative state of the contract given various hypothetical "what if" scenarios.

Castaneda [18] provided a critique of Von Wright's logic of change (which Lee utilised), stating that the T calculus suffers from significant ambiguity in it's interpretation of the T operator. Castaneda suggested the introduction of a new T* operator to address this ambiguity. Additionally he critiques von Wright's axioms of the OP-Calculus, used to describe Obligations and Permissions, claiming the assertion of "You ought to do A and B" does not entail "You ought to do A", is incorrect, as statements such as "Open the window and shut the door" would entail that one should open the window even if the door was already shut.

An extensive background on the state of computable programming and the underlying requirements facing the field was provided by the LSP Working Group [19]. The goal of the LSP working group is to define a Legal Specification Protocol (LSP) to provide a set of standards and conventions for capturing the legal event space and legal formulations in a way that follows the logical structure of legal documents and is machine-executable. The White Paper compiles a relatively detailed taxonomy of the information requirements needed to accurately represent a legal computational model. The LSP group anticipates that an interface to a legal protocol would employ a symbolic or graphical user interface and would not be "a big-data emulation based on the flawed system of natural language contractual representation".

J. Hazard and H. Haapio [20] extended the idea of wise contracts that work for humans and machines based on the "Ricardian Contract" paradigm [21], and the requirements

of smart contract templates set out by Clack et al.[7]. The paper highlights how prose objects could serve as a connection between human and automated systems in a way that enables programmers to automate computable interactions without automating the entire contract, providing incentive for a gradual codification of the law. The paper offers insight into how a "barebones" core Contract Object could be used and extended with a variety of jurisdictions and languages.

C. Clack conducted a detailed investigation of the languages used for smart and computable contracts [9]. This work covers the various challenges faced when attempting to automate the operational performance of a legal agreement through smart contracts. Through an exploration of the differences in semantic interpretations and logical analysis of the highly specialised fields of computer science and law, the difficulties of translating a legal agreement onto code are highlighted. The paper addresses the need of drafting contracts through a language that results in a single artefact, where the contract is the code. Various approaches to enable such a single artefact contract, like markup languages, controlled natural languages, and domain specific languages, are investigated and compared, concluding that while some languages allow for the expression of complicated legal prose, they are mostly untested on larger contracts or inaccessible lack a style that makes the drafting accessible to lawyers.

Languages developed for smart and computable contract creation vary considerably in scope and application. Generally speaking, in their implementation, the various languages fit within the three categories describe Clack [9]: (i) Markup languages and templates; (ii) Domain-Specific Languages (DSL); (iii) Controlled Natural Languages (CNL).

Markup Languages and Templating systems, such as the Cicero templating system of the Accord Project [22] or LegalML [23], combine natural language with parameters, offering a framework for reusable contract templates.

Domain Specific Languages attempt to improve the process of legal contract drafting as the contract is code, thus the drafting of a contract is more akin to programming than writing a natural language text. The Catala language [24] developed by Merigoux et al.

is one example of such a DSL, where each line of a legislative text is annotated with a snippet of code that represents the codified meaning of the text. While this enhances the comprehension of legal code for non-legal experts, the manual codification of legal prose remains a significant challenge.
Another notable DSL, the L4 language [25], allows contracts to be constructed in a highly structured spreadsheet environment. While the L4 language enables a range of powerful outputs such as interactive exploration of decision logic and contract visualisation as a Petri Net, the spreadsheet-style drafting process makes the language inaccessible to non-experts and may deter widespread adoption.

Controlled Natural Languages are a subset of a natural language such as English, with a restricted grammar and vocabulary allowing for an unambiguous text that be easily parsed by a computer. The Lexon language, developed by H. Diedrich [10] is a combined CNL and DSL, with the aim of providing a computer executable and human-understandable contract drafting language. The Lexon language provides an extremely powerful combination of old-style contractual drafting with a syntax and grammar that can easily be understood by lawyers and non-programming professionals while also automatically generating smart contract code in Solidity or Sophia to be executable on the Blockchain. Unlike natural English whose core vocabulary is estimated to consist of roughly 850 words, Lexons core vocabulary consists of only 130 words, as it only allows the expression of a limited number of verbs [12].

3 | Requirements and Analysis

3.1 The Need for a Logical Contract Model

Upon review of the current advancements towards Smart Contracts and the digitisation of law, it has become apparent that a significant effort has been put towards the automated performance of contracts. Interestingly, this has largely eclipsed the foundational task of creating a comprehensive and logical model of a legal agreement first. At its core, the digital contract model could be likened to the *geometry* of a contract, laying out permissions, obligations, and prohibitions detailed in the agreement according to various possible scenarios.

A suitable analogy to such a model can be drawn from the realm of Computer Aided Design (CAD). Using CAD, engineering components can be defined by a series of interconnected coordinates, i.e. the geometry of a 3 dimensional part. Once this geometry is established, further analysis can be built on top of this foundational model. This includes analysis such as stress testing, thermodynamic simulations, or even producing the instructions needed for a 3D printer to manufacture the part straight from the software. Tasks that were previously time and resource intensive, such as producing a physical engineering drawing, evaluating with technicians to see whether it can be manufactured or redrawing the part for a design iteration, have become significantly faster and cheaper [26].

Similarly to how the CAD model serves as a foundation for more integrated tasks, the digital contract, when constructed as a logical model, will lay the groundwork for further layers of application for the legal profession. Such a model would be capable of representing the expected interactions between the contractual parties given a range of scenarios. Once established, it could enable additional applications and analysis such as automated inconsistency checks, compliance assessments against current legislation, comparison of contracts drafted with slight changes in clauses, contract visualisation, and automated contractual performance.

The attempt to create such a model, such as the representation of a loan agreement as a DFA by Flood and Goodenough [11], demonstrated the possibility of representing a complex real contract as a state machine. However, this attempt required manual conversion of the contract into the DFA and only served as a proof of concept. The DFA was chosen for its simplicity but would likely be an impractical medium for the representation of complex contracts [11].

The LSP working group suggests that the creation of a digital contract would likely be done through a graphical user interface, to avoid the flawed and ambiguous nature of natural language [19]. However, the user experience of any such model for the drafting lawyer is a critical issue and will likely require a solution that is familiar and accessible to the drafting lawyer, such as a formal language as suggested by C. Clack [7]. An example of such a model, based on the use of a formal language, is the logical contract model developed by R. Lee [1], who used a natural language interface to convert simple written agreements into a Petri Nets. The ability of automatically representing contracts as Petri Nets appeared promising, but insufficient, as the generated Petri Net, was not easily interpretable regarding the details on the performative state of the contract as a whole, and only simple contracts were generated.

Thus, there exists a gap in the ability to automatically create a logical representation of a contractual agreement expressed through natural language. This paper proposes that the use of CNL structures and restricts the ambiguity in legal agreements sufficiently to allow the representation of a contracts *geometry* in the form of a Petri Net state machine. This representation could be an enormous advantage to drafting lawyers, providing them with the ability to easily visualise a contracts logic and enabling them to pinpoint elements tied to any clause in the agreement. Furthermore, the Petri Net representation of the contract can facilitate performance simulation of the agreement, offering detailed insights into a contracts preformative state for a variety of scenarios.

3.2 Requirements for a Digital Contract Model

Based on the analysis of existing literature on smart computing contracts, controlled natural languages, and logical representations of contracts as state machines, the following section presents a list of requirements for the logical contract model.

(R1) ***Use a controlled natural language as input source***
Smart and Computable contracts will need to be expressed in a manner that is understandable to both humans and computers. While this can be achieved through various methods, including the production of two separate artefacts, one being the contract and one the representative code, this can carry significant risks for validation and interpretation of linguistic semantics [9]. Thus, a single artefact, likely generated by a CNL where the contract is the code, will be adapted and used for this model.

(R2) ***Ensure the output is faithful to the original contract.***
When converting a contract into a logical model, it is required to maintain the spirit of the contract, so that the model represents the original artefact without deviation. Among other things, this requires the model to successfully include the conditional links expressed throughout the contract, such as conditional obligations or sanctions. Flood and Goodenough elaborated on the importance of such sanctions and conditionals, as the majority of a contract is often composed of "unhappy paths", describing the ramifications and implications if a party fails to meet its expectations [11].

(R3) ***Support the fulfilment and voiding of expectations.***
A methodology is required to ensure that expectations can be voided or considered fulfilled, such as an agreement stating that *"If the tenant did not pay the rent at the specified date, the Landlord may charge interest until the tenant has paid all outstanding charges"*. While it is important to show that the permission to charge interest can only be considered active upon confirmation of its precondition, that is, the tenant has not paid rent, it is just as important to model that this permission should be voided after the tenant paid the outstanding charges. Furthermore, obligations such as *"Party A must pay $500 to Party B"*, should be considered fulfilled and not active once Party A has met its obligations and paid.

(R4) ***Do not imply that which is not specified.***
The translation of the contract into a logical model should not imply that which is not specified in the original contract and avoid deontic assumptions. Consider the example statement *"It is not the case that Party A shall pay some amount to Party B"*, which could be expressed as deontically equivalent to *"Party A may not pay some amount to Party B"*. The translated contract should not leap to such conclusions, as they may diverge from the contract drafter's intent. Likewise, the translation of the model should not make assumptions about unspecified repercussions for violated expectations. Lee's model [1], assumes that a statement such as *"Party A shall do B after which Party C shall do D"* implies that if *Party A did not do B* the contract is in a state of default. This model may not make any such assumption, as the statement simply did not specify what ought to occur in such a scenario.

(R5) ***Allow the contract to be represented as a Petri Net.***
The logical model should enable the contract to be visually expressed through a Petri Net. This should be of sufficient complexity to allow for a faithful representation of the source contract but remain intuitive and accessible for those unfamiliar with Petri Nets, like legal professionals.

(R6) ***Support the performance simulation of a contract.***
A logical contractual model should encompass contractual expectations in a way that supports simulation of contractual performance. This simulation should be visually represented in the Petri Net, through the recognition of facts and observations by an oracle.

(R7) ***Provide insight into the current state of a contract.***
Using the contract performance simulation, the model, given a series of events, should clearly and accurately express the expectations according to the source contract.

4 | Design And Implementation

The following chapter details the design and implementation decisions made for the project.

4.1 The Contract as a Petri Net

This section describes the nuanced design challenges of representing the contract as a Petri Net. This design process required a multitude of assumptions and compromises, as there proved to be a recurring trade-off between capability of the model and readability of the final output.

4.1.1 Statements and Conditions

The core of a contract Λ are the deontic statements Φ. These refer to obligations $O(\Phi)$, permissions $P(\Phi)$ or prohibitions $F(\Phi)$, that is, the actions each party should, could or shan't do, which will be referred to as action statements moving forward.

$$\Lambda = \{\Phi_0, ..., \Phi_n\}$$

Often times, in contracts, an action statement depends on a condition Ψ before it becomes active, such as *"Party A shall do B if party C did E"*, which we will express in the Petri Net as such:

$$P_\Psi \rightarrow T^E_\Phi \rightarrow P_\Phi$$

P_Ψ refers to the condition place node, T^E_Φ to what we will refer to as the enabling transition, and P_Φ to the place node of the action statement itself. We can also express unconditional statements through $T^E_\Phi \rightarrow P_\Phi$.

As described in Section 2.2, a transition, fires if all its input places have a sufficient amount of tokens. Based on this, Figure 4.1 illustrates that the enabling transition of the unconditional obligation would always be able to fire, and so the place node *"Party A shall do B"* will always evaluate true. The conditional obligation however, can only obtain a token from its enabling transition if the conditions place node has a token.

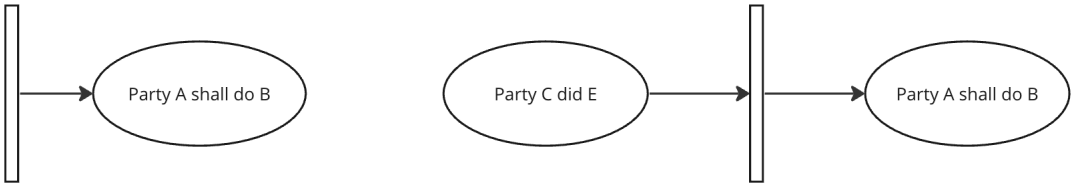


Figure 4.1: Simple statements represented as Petri Nets. Left: (Unconditional Obligation) enabling transition connected to obligation place node; Right: (Conditional Obligation) condition place node connected to enabling transition.

4.1.2 Token management

A major challenge in representing the contract as a Petri Net comes from the issue of token management. The methodology described by Lee [1] mostly avoids this issue, as it is concerned with substates as a whole rather than with individual conditions and statements. This model attempts to represent the contract in a more accessible manner, aiming to provide detailed insight into the contract, and as such, the following assumptions are made.

(A1) Each place node must contain a statement that can be evaluated as a boolean, e.g. *"Party A did B"* or *"Party A shall do B"* but not a statement that could not be logically evaluated, such as *"Hello"*. Given this, each place node, is either true and holds a token or false and holds no token. Thus each place node represents a distinct substate of the contract, e.g. a condition, definition or expectation, and whether it is true or false.

(A2) Whether a place node that refers to a condition holds a token, is evaluated by an oracle. This is an outside source that evaluates the result and can arbitrarily create or destroy a token at a conditional place node.

(A3) place nodes referring to conditions, only change evaluation, i.e. gain or lose a token, due to an event being registered by the oracle. Such an event can be the observation of a fact (e.g. $x > 5$) or the confirmation of an action (e.g. *"Party C did E"*).

Moving forward, a coloured Petri Net is adopted for the sake of visual clarity, where white means that a place node has no token and green means that a place node has a token.

Referring back to Figure 4.1, the need for read-only edges in the system can be deduced, since the enabling transition for an obligation such as *"Party A shall do B"* should not consume the token present at the conditional node *"Party C did E"* as this condition would continue to be true. Although the example in Figure 4.1 would not be affected by the consumption of the token, a contract where multiple obligations depend on the same condition would no longer be logically valid as shown in Figure 4.2. Here, the consumption of the token at the *"Party C did E"* place node, at either enabling transition, would cause the other transition, to no longer be enabled.

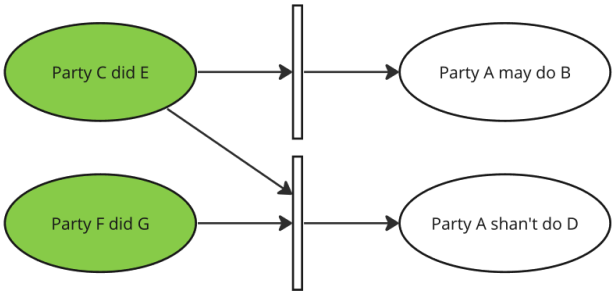


Figure 4.2: Petri Net of two action statements depending on the same condition. *"If Party C did E then Party A may do B. If Party C did E and Party F did G, then Party A shan't do D"*.

Therefore, conditional place nodes will be connected to enabling transitions through the use of read-only arcs described in Section 2.2, denoted as $\xrightarrow{\mathrm{r}}$, as illustrated in Figure 4.3.

$$P_{\Psi} \xrightarrow{\mathrm{r}} T_{\Phi}^{E} \rightarrow P_{\Phi}$$

Not only are statements dependent on confirmation of conditions, often they also depend on the negation of a condition such as *"If party C did not do E, Party A shall do X"*. To incorporate this into the Petri Net, each condition ψ must be represented with its negation using a mutual exclusive connection, presented by Wang [27], since ψ and $\neg\psi$ cannot be true at the same time.

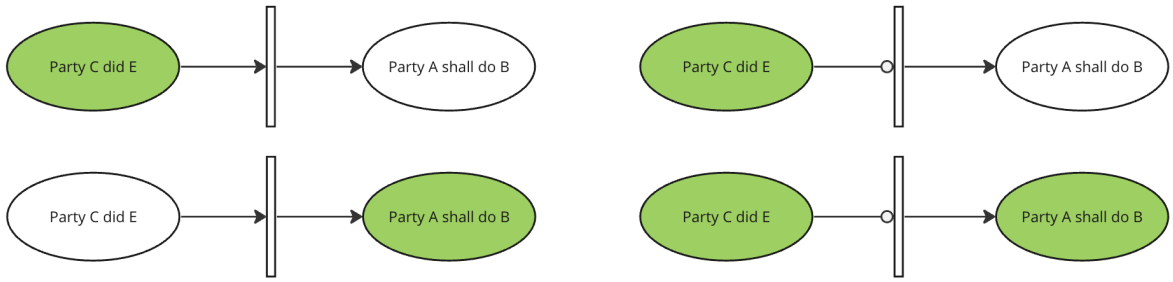


Figure 4.3: Illustration of the difference between normal and read-only arcs on the Petri Net before (top) and after triggering (bottom). Left: Conditional place node connected to enabling transition through normal arc; Right: Condition place node, connected to enabling transition through read-only arcs.

The style of negation representation, shown in Figure 4.4, creates a set of new challenges. For instance, a transition fires, once all its input places have a token. Therefore, once a token is present at either *"Party C did E"* or *"Party C did not do E"*, there is always a transition able to fire, and the token will continuously flow around, even though this should not be the case as the statement is either true or false. Additionally, any such representation of negation through the inclusion of a place node of the negation condition significantly increases the number of nodes and arcs in the final Petri Net, causing the output to be less accessible.

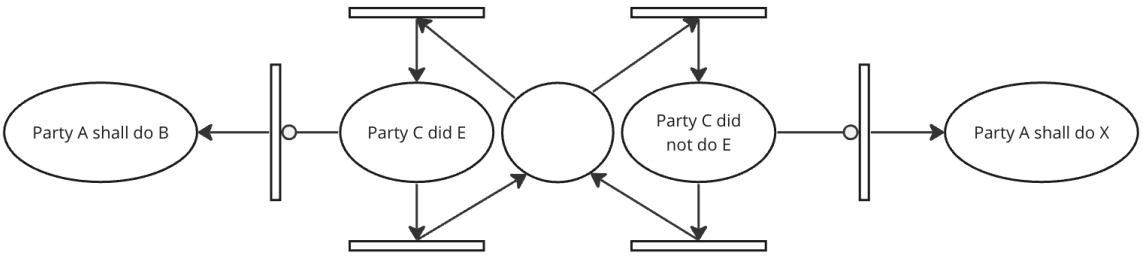


Figure 4.4: Petri Net of conditional obligations containing mutual exclusivity between confirmation and negation of a conditional place node.

Therefore, moving forward, each place node describes the confirmation of a proposition, and the negations are implemented through inhibitor arcs $\xrightarrow{i}$ as shown in Figure 4.5.

$$P_\Psi \xrightarrow{\text{i}} T_\Phi^E \rightarrow P_\Phi$$

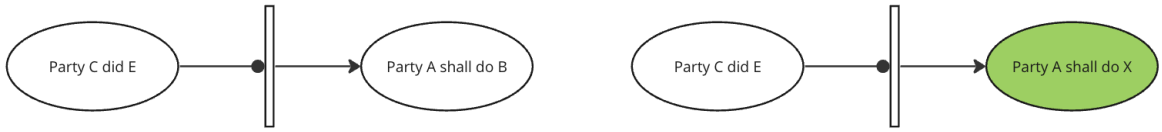


Figure 4.5: Petri Net of conditional obligation before (left) and after (right) firing of transition. Condition connected to enabling transition through inhibitor arc.

4.1.3 Evaluating the Unknown

With the previous addition of the inhibitor arc, a new issue is created. Suppose the statement *"If party C did not do E before 01/09/2024, Party A shall do X"*, where the condition now includes a deadline. As stated by Lee [1], while normal logical programming has failure by negation, temporal negation is more complicated, as if the deadline has not yet been reached, and *Party c did not do E* so far, it cannot be said whether or not the condition is true (has a token) or false (has no token), as it is not yet known whether the condition will hold until the deadline has passed.

To address this issue, an alteration to **(A1)** was made, stating that a condition is true, false, or unknown. This concept is already somewhat included in the mutual exclusivity example in figure 4.4, as the node in between confirmation and negation. The choice of a confirmation node-only approach will instead require the addition of a new colour, Grey, to the Petri Net, to represent an unknown value.

The addition of a new colour warrants the addition of a further rule to the requirement of a transition node to fire, as a transition node should only fire if none of its input places contains an unknown value.

4.1.4 Conjunction and Disjunction of Conditions

Often times, it is also the case that statements are dependent on more than one condition. This can come in the form of a conjunction $\wedge$ (and) or a disjunction $\vee$ (or).

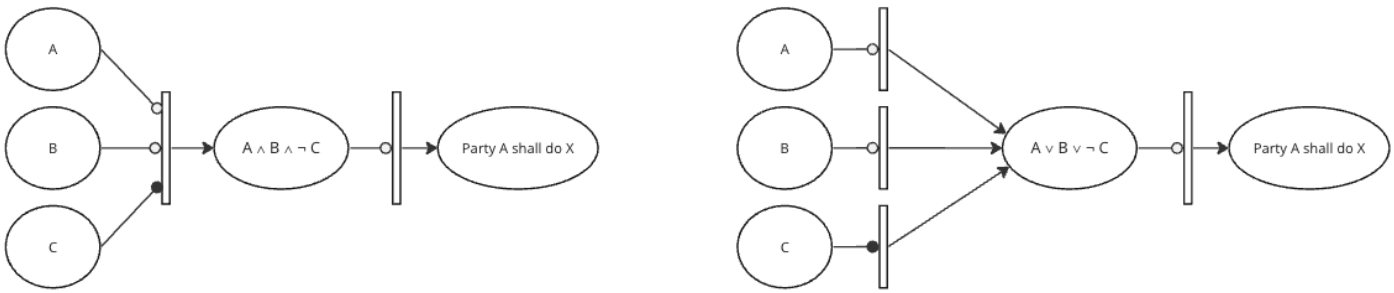


Figure 4.6: Left: Conjunction *"If A and B and not C then Party A shall do X"*; Right: Disjunction *"If A or B or not C then Party A shall do X"*

Figure 4.6 highlights the need for a new place node to represent a disjunction, as place nodes can only connect to transitions and vice versa. The conjunction does not require an extra place node; however, it was found that including a result place node for the conjunction of conditions resulted in a simpler final Petri Net while maintaining the functionality and was thus adopted.

4.1.5 Fulfilment and Voidance

Although statements such as obligations, permission, or prohibitions may become active only under certain conditions, it is similarly critical for a contractual model to allow for the representation of the conditions under which an obligation, permission, or prohibition no longer holds, either through fulfilment or voidance. Consider the simple agreement used by Lee [1].

> "Jones agrees to pay $500 to Smith by May 3, 1987. Following that, Smith agrees to deliver a washing machine to Jones within 10 days."

The conversion of this agreement into a Petri Net, and the evaluation of the sce-

nario where Jones paid the $500 to Smith before 3 May 1987, but Smith has not yet delivered the washing machine (before passing of the deadline) is illustrated in Figure 4.7.

Figure 4.7: Incomplete Petri Net of simple transfer agreement by Lee [1]. Conversion of the agreement into the Petri Net, does not provide a cohesive link between the obligation (*"Jones shall pay $500 to Smith by May 3, 1987"*) and the place node which confirms the fulfilment of the obligation (*"Jones paid $500 to Smith by May 3, 1987"*).

Figure 4.7 highlights the lack of a cohesive link between the place node of the obligation *"Jones shall pay..."* and the place node that marks the fulfilment of the obligation *"Jones paid ..."*. Due to the lack of such a link, the obligation place node, holds a token even after the obligation has been fulfilled. This is incorrect, as if Jones fulfilled the obligation, the obligation should no longer hold, and thus lose it's token.
Furthermore, by expressing only the statements and conditions specified in the agreement, there would be no place node containing the confirmation of the second obligation (i.e. *"Smith delivered a washing machine..."*), making the evaluation of whether this obligation has been fulfilled from the Petri Net impossible.

The issue of the missing link and the non existing place nodes for the fulfilment condition is remedied through the following rule. Any enabling action statement, must always include a fulfilment condition place node $P_{\Phi C}$ and a voiding transition T_{Φ}^{V}. Unlike the enabling transition, which acts as a source for tokens, the voiding transition acts as a sink. Furthermore, each action statement's place node, will always be linked to its voiding transition through a normal arc, while the place node of the fulfilment condition $P_{\Phi C}$ will be connected to the voiding transition through a read-only arc.

The following syntax is introduced, $\Theta \rightarrow T$ where Θ represents a set of place nodes, $\{P_0, ..., P_n\}$, each connected to the transition T through a normal arc. This is further

extended to allow for the representation of the various types of arcs in the system and is expressed as $(\Theta_n \rightarrow, \Theta_r \xrightarrow{\text{r}}, \Theta_i \xrightarrow{\text{i}})T$, where Θ_n represents all the place nodes connected to the transition through a normal arc, while Θ_r and Θ_i represent the set of place nodes connected to the transition through read-only and inhibitor arcs respectively. Based on this syntax, the rule given above is expressed as:

$$(\{P_\Phi\} \rightarrow, \{P_{\Phi C}\} \xrightarrow{\text{r}}, \{\} \xrightarrow{\text{i}})T_\Phi^V$$

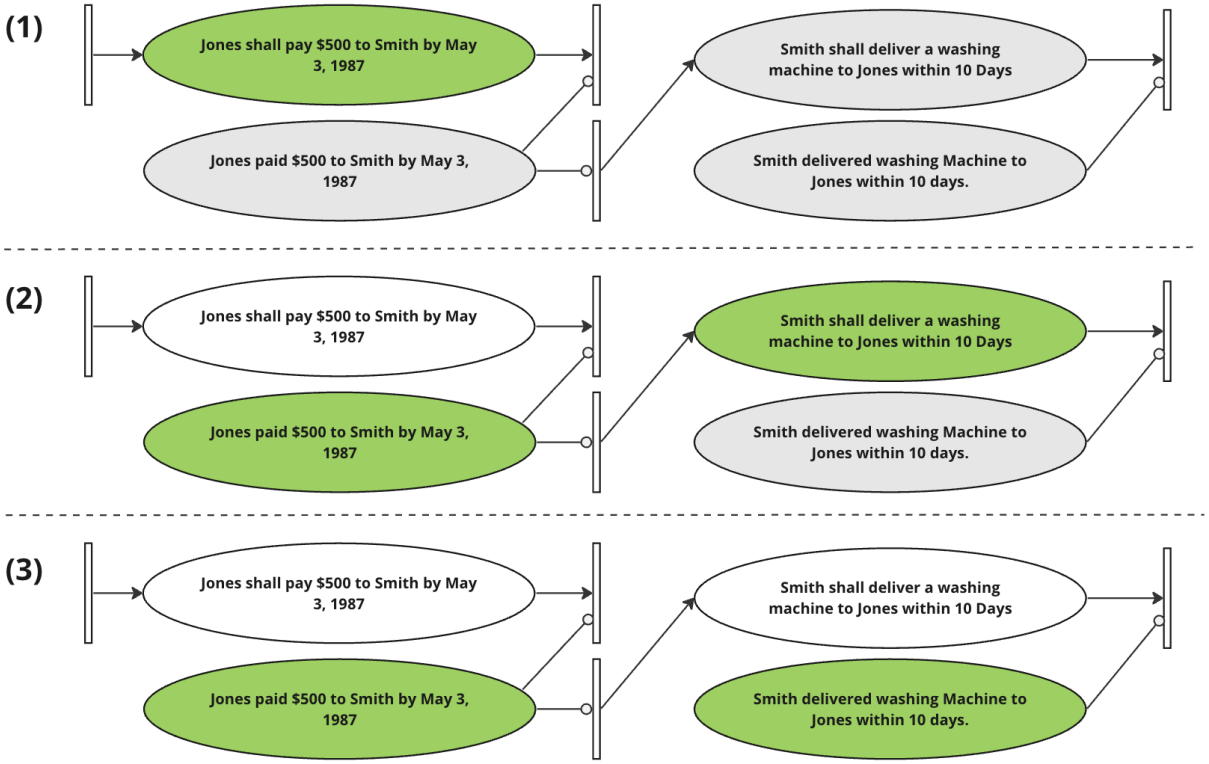


Figure 4.8: Performance simulation of simple transfer agreement by Lee [1], with addition of voiding transition. **(1)** Beginning of Agreement; **(2)** Jones paid $500 to Smith by May 3, 1987; **(3)** Smith delivered washing Machine to Jones within 10 days.

The introduction of this rule allows us to faithfully and accurately simulate the agreement against a series of events. The newly generated Petri Net, illustrated in Figure 4.8, now correctly highlights the active obligations after each event occurs.
The addition of the voiding transition also ensures that if an expectation has been fulfilled,

its place node will never have a token. Even if its enabling condition remains true, causing a token to flow from the enabling transition to the statement place node, this token is immediately consumed by the voiding transition. This ensures that once an action is done, its action statement can no longer be active, unless the oracle resets the status of the fulfilment condition, e.g. for actions such as repeated payments.

4.1.6 Definitions

The discussion of contracts so far has mostly referred to action statements, the principles concerning duties and obligations. However, the contractual model also accommodates conditional and unconditional definitions, denoted as D. These definitions facilitate expressions like *"The Excess Party is Party A"* or conditional assertions such as *"If Party B performed action C, the Fee is 500"* and are expressed similarly to statements as:

$$P_\Psi \rightarrow T_D^E \rightarrow P_D$$

Thus, definitions are similar to statements in that they can be conditional or unconditional. Unlike statements, however, definitions only have an enabling transition and no voiding transition, nor do definitions have a fulfilment place node.

4.1.7 Substates

Substates represent distinct phases or statuses in the contractual lifecycle, highlighting events such as a party defaulting or a breach of contract. These phases might arise as a result of certain conditions being met or might depend on the decision of an oracle and are expressed as:

$$P_\Psi \rightarrow T_S^E \rightarrow P_S$$

This allows the definition of complex conditional relationships in the contract. For example, referring to the prior agreement shown in Figure 4.8. A possible breach substate for this agreement could triggered by the following condition: "*If Jones did not pay $500 to Smith by May 3, 1987. Or If Jones paid $500 to Smith by May 3, 1987 and Smith failed to deliver a washing machine to Jones within 10 days*", is illustrated in Figure 4.9.

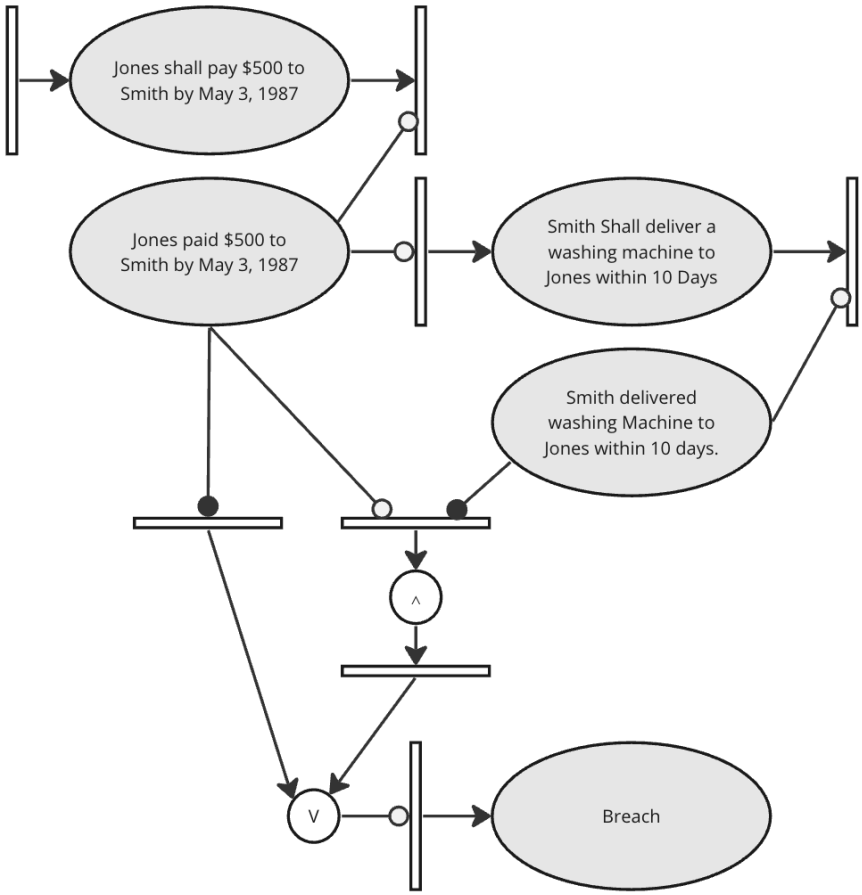


Figure 4.9: Example of a breach substate for the simple transfer agreement by Lee [1], dependent on the condition: "*If Jones did not pay $500 to Smith by May 3, 1987. Or If Jones paid $500 to Smith by May 3, 1987 and Smith failed to deliver a washing machine to Jones within 10 days*"

4.2 CNL as Input Source

This section describes the conversion of the source languages Lexon and CoLa into the logical model used for Petri Net generation.

To facilitate this conversion, translators were developed for both languages, translating the Abstract Syntax Tree (AST) generated by either the languages parser onto an intermediate syntax, capable of expressing the elements of both languages.

4.2.1 Intermediate Syntax

The intermediate syntax acts as a bridge for translating contracts written in Lexon or Cola into a common format that can be used for Petri Net generation. This was required because the fundamental differences between the two languages meant that adopting one language's structure for this format would lead to a loss in expressiveness and functionality of the other.

One such difference between the two languages is Lexon's requirement and CoLa's lack thereof, to express the target subject of certain actions. Thus, while Cola would allow the expression *"PersonA shall pay SomeAmount at SomeDate"*, Lexon would require the specification of who PersonA shall pay SomeAmount to, e.g. *"PersonA pays SomeAmount to PersonB"*. The ability to express the target subject of an action is a critical one, and likely desirable, for contracts of any significant complexity. However, as CoLa is one of the source languages and it's syntax does not allow for the specification of the target subject for an action, we must allow for statements with or without a second subject.

Another major difference that needed to be addressed was CoLa's requirement of expressing a deadline in each statement, e.g. "PersonA shall pay SomeAmount at SomeDate". Lexon, while allowing statements based on temporal conditions such as *"If the date is equal to SomeDate, then PersonA pays SomeAmount to PersonB"*, does not require the specification of a temporal deadline for each statement. Therefore, the intermediate syntax must allow for the expression of a statement with or without temporal limitations.

The last significant deviation between the two languages that needed to be addressed at the syntax level is CoLa's ability to not only allow for the enabling of action statements but also for their voiding, e.g. *"It is not the case that PersonA shall pay SomeAmount at SomeDate"*. This concept is fundamental to CoLa's logic, and as such the intermediate syntax must contain it. Lexon, however, allows only for the expression of enabling action statements. Thus, the combined syntax will adopt CoLa's design in allowing actions statements to be of enabling or voiding nature.

4.2.1.1 Backus-Naur Form (BNF) Syntax

The intermediate syntax is formally defined using Backus-Naur Form (BNF). This syntax includes intentional ambiguities to accommodate unique features of each source language, such as CoLa's deontic conditions (e.g. *"If it is the case that PartyA shall pay SomeAmount at SomeDate, then..."*) and is in the form of the Python Classes used for it's implementation. The primary role of this intermediate syntax is to enable accurate mapping from an Abstract Syntax Tree (AST) generated by the parsers for Lexon and CoLa into a common format. The ambiguities that were present in the original contract have already been addressed during the creation of the AST, thus eliminating any uncertainty in the understanding of the text.

⟨*contract*⟩ ::= Contract([⟨*statement-list*⟩], [⟨*definition-list*⟩], [⟨*state-list*⟩])

⟨*statement-list*⟩ ::= ⟨*statement*⟩ | ⟨*statement*⟩, ⟨*statement-list*⟩

⟨*definition-list*⟩ ::= ⟨*definition*⟩ | ⟨*definition*⟩, ⟨*definition-list*⟩

⟨*state-list*⟩ ::= ⟨*state*⟩ | ⟨*state*⟩, ⟨*state-list*⟩

⟨*statement*⟩ ::= Statement(⟨*one-sbj-stmt*⟩)
| TemporalStatement(⟨*one-sbj-temp-stmt*⟩)
| TwoSubjectStatement(⟨*two-sbj-stmt*⟩)
| TemporalTwoSubjectStatement(⟨*two-sbj-temp-stmt*⟩)
| ConditionalStatement(⟨*condition*⟩, ⟨*statement*⟩)

⟨*one-sbj-stmt*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*modal-verb*⟩, ⟨*verb*⟩, ⟨*object*⟩

⟨*one-sbj-temp-stmt*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*modal-verb*⟩, ⟨*verb*⟩, ⟨*object*⟩, ⟨*temporal-expression*⟩

⟨*two-sbj-stmt*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*modal-verb*⟩, ⟨*verb*⟩, ⟨*object*⟩, ⟨*prefix*⟩, ⟨*subject*⟩

⟨*two-sbj-temp-stmt*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*modal-verb*⟩, ⟨*verb*⟩, ⟨*object*⟩, ⟨*prefix*⟩, ⟨*subject*⟩, ⟨*temporal-expression*⟩

⟨*condition*⟩ ::= ActionCondition(⟨*one-sbj-cond*⟩)
| TemporalActionCondition(⟨*one-sbj-temp-cond*⟩)

| TwoSubjectActionCondition(⟨*two-sbj-cond*⟩)
| TemoralTwoSubjectActionCondition(⟨*two-sbj-temp-cond*⟩)
| StatementCondition(⟨*statement*⟩)
| StateCondition(⟨*test*⟩, ⟨*state*⟩)
| ExpressionCondition(⟨*test*⟩, ⟨*expression*⟩)
| AndCondition([⟨*condition-list*⟩])
| OrCondition([⟨*condition-list*⟩])

⟨*condition-list*⟩ ::= ⟨*condition*⟩ | ⟨*condition*⟩, ⟨*condition-list*⟩

⟨*one-sbj-cond*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*verb-status*⟩, ⟨*object*⟩

⟨*one-sbj-temp-cond*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*verb-status*⟩, ⟨*object*⟩, ⟨*temporal-expression*⟩

⟨*two-sbj-cond*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*verb-status*⟩, ⟨*object*⟩, ⟨*prefix*⟩, ⟨*subject*⟩

⟨*two-sbj-temp-cond*⟩ ::= ⟨*test*⟩, ⟨*subject*⟩, ⟨*verb-status*⟩, ⟨*object*⟩, ⟨*prefix*⟩, ⟨*subject*⟩, ⟨*temporal-expression*⟩

⟨*definition*⟩ ::= ⟨*basic-definition*⟩
| ⟨*conditional-definition*⟩

⟨*definition-list*⟩ ::= ⟨*definition*⟩
| ⟨*definition*⟩, ⟨*definition-list*⟩

⟨*basic-definition*⟩ ::= IsDefinition(⟨*subject*⟩, ⟨*value*⟩)
| EqualsDefinition(⟨*subject*⟩, ⟨*expression*⟩)

⟨*conditional-definition*⟩ ::= ConditionalDefinition(⟨*condition*⟩, [⟨*definition-list*⟩])

⟨*state*⟩ ::= State(⟨*condition*⟩, ⟨*label*⟩)

⟨*subject*⟩ ::= Subject(⟨*label*⟩)

⟨*expression*⟩ ::= NumericExpression(⟨*numeric-object*⟩, ⟨*numeric-operator*⟩, ⟨*numeric-object*⟩)
| TemporalExpression(⟨*temporal-operator*⟩, ⟨*temporal-object*⟩)
| PropertyExpression(⟨*subject*⟩, ⟨*property*⟩)
| BooleanExpression(⟨*subject*⟩, ⟨*verb-status*⟩, ⟨*comparison*⟩, ⟨*subject*⟩)

⟨*numeric-object*⟩ ::= ⟨*subject*⟩ | ⟨*number*⟩

⟨*temporal-object*⟩ ::= ⟨*subject*⟩ | ⟨*time*⟩

⟨*object*⟩ ::= ⟨*subject*⟩

⟨*prefix*⟩ ::= To | From | Into

⟨*comparison*⟩ ::= Greater Than | Less Than | More Than
| Equal | Not Equal | Less Or Equal | More Or Equal

⟨*numeric-operator*⟩ ::= Plus | Minus | Times | Divided

⟨*temporal-operator*⟩ ::= During | On | Throughout | Within | Before | After | By

⟨*test*⟩ ::= True | False

⟨*evaluation*⟩ ::= True | False | Unknown

⟨*modal-verb*⟩ ::= MAY | SHALL | SHOULD | SHANT | MUST

⟨*verb*⟩ ::= pay | deliver | transfer | fix | notify | issue | give
| demand | appoint | envoke | increase | decrease | change
| charge | return | certify

⟨*verb-status*⟩ ::= paid | delivered | transferred | fixed | notified
| issued | given | demanded | appointed | envoked | increased
| decreased | changed |charged | returned | certified

⟨*label*⟩ ::= ⟨*string*⟩

4.2.2 Translating Lexon

As described in Section 2.1.1, Lexon is a highly readable and powerful CNL, allowing for the expression of smart contracts through a syntax that is very close to natural English, enabling the lay person without a background in computer science or law to read and understand a contract written in it.
However, it is crucial to recognise Lexon's dual purpose: to seamlessly serve as both legal agreement and smart contract. This duality, inspired by the idea that "the contract is the code", underscores Lexon's inherent purpose, to generate executable code. Such code does not simply define the concepts of an agreement and what each party should do, but sets the act of the fulfilment of the agreement. Thus, Lexon describes the performance of a contract, more than the contract itself, which has implications on how we need to interpret it in our model.

4.2.2.1 The Deontics of Lexon

With Lexons, design being tailored towards the generation of executable code for smart contracts, this design distinguishes it from traditional contracts, especially in its approach to deontic elements.

For example, Lexon forgoes the explicit prohibition of an action. Given its executable nature, actions that are not explicitly permitted are never written. Furthermore, instead of imposing obligations, Lexon delineates permissible actions under specific conditions. Thus, the word *May*, which traditionally signals permission, serves a somewhat different purpose in Lexon, where it can be used within a clause.

Once parsed, a clause written in a Lexon contract is interpreted as a function that encompasses the logic of the clause as a distinct piece of code. Unlike the code generated by the recitals, which gets executed as the first action of the performance, a clause function has no set time of execution. Any contractual party may call the function at any time once the performance of the contract has begun, essentially equating the act of calling or envoking a clause with a permission. Listing 4, contains an altered version of a Lexon clause provided by Diedrich [3], and its compiled Solidity code.

From this example, it is evident that the purpose of *May* in Lexon, is not to signal a permission, but rather to act as a guard for which party can trigger the execution of specific actions covered by the clause. This is necessary because it enables the exclusivity of actions for individual parties. Referring to the Solidity code in Listing 4, only the Arbiter can trigger the execution of lines 5 and 6, while only the payer can trigger the execution of line 3. This also means that while a third party such as the Payee may call the function, this would result in no action being taken as they do not fulfil the guard conditions.

```
CLAUSE: Pay Out.
The Payer may pay the escrow to the Payee.
The Arbiter may pay from escrow the Fee to themselves,
and afterwards pay the remainder of the escrow to the Payee.
```

```
function Pay_Out() external {
    if (msg.sender == payer){
        payee.transfer( address(this).balance );
    }else if (msg.sender == arbiter){
        arbiter.transfer(fee);
        payee.transfer(address(this).balance);
    }else{
        require(false);
    }
}
```

Listing 4: Altered Lexon Pay Out clause and generated Solidity code of escrow agreement [3], highlighting the use effect of May statements.

The generated code also underscores the sequential nature of code execution in a synchronous language such as Solidity [28]. This entails that each action can only be taken, if the previous action is completed, e.g. the arbiter can only transfer the rest of the escrow, once the fee is paid to themselves. Therefore, when sequential actions are transferred into the contractual model, any action statement that follows another must be conditional on the fulfilment of the previous statement.
Sequential code execution has another significant implication as once a code block starts execution, all subsequent code within it will execute, unless prevented by a further nested condition. This means that although the first action within a code block might represent

a permission (e.g. it is a permission for the arbiter to pay a fee from the escrow to themselves), the subsequent code within the same code block describes obligations. This distinction is crucial when translating Lexon into the contractual model, as it affects the interpretation of the parties' responsibilities. For example, to stay faithful to the spirit of the Lexon contract, once the Pay Out clause is envoked, the permission for the arbiter to pay the fee from the escrow to themselves stands, but once they choose to do so, they are obligated to transfer the remainder of the escrow to the payee.

4.2.2.2 Structure of a Lexon Contract

The structure of Lexon introduces further intricacies that must be taken into account when translation into the logical model. A Lexon contract, also called a Lexon, is based on the following high-level structure [12]:

1. Sentence → Subject + Predicate [, Predicate]
2. Predicate → Verb + [Object]
3. Lexon → Head + Terms + Contract
4. Terms → Head + Definitions + Recitals + Clause
5. Contract → Head + Definitions + Recitals + Clause
6. Clause → Head + Definitions + Permissions + Conditions + Statement

In Lexon, terms define the clauses and recitals that will be applicable to each constituent contract. This is needed as Lexon has the ability to express different contracts for different stakeholders within the same programme, e.g. a contract "per member" or "per payee", all of which share the common functionality laid out in the terms.

Further diverging from traditional legal interpretation, the term Recitals, typically referring to an accounting of contextual facts or events, in Lexon describes the code that will be executed before any other actions can be taken in the contract.

Another noteworthy detail about the structure of Lexon is the distinction between the actions specified in a clause and the ability to invoke the clause function itself. For example, a clause might specify an action for a particular party, such as *"Arbiter pays someAmount to Payee"*, however, the clause function itself does not inherently restrict

who can call it and trigger its execution. Although "May" statements within a clause can restrict certain actions to specific parties, this is not mandatory. As such, a clause could include general statements like "The contract is terminated," which do not specify any particular party. Similarly, a statement like "The Arbiter pays someAmount to Payee," if not accompanied by additional conditions, could technically be triggered by any party involved in the contract. This implies that, in general, it is not possible to know which party can invoke a clause but only that it can be invoked.

4.2.2.3 Rules for Lexon Translation

Based on the previous discussion, the following list outlines the key rules that are applied during the process of translating a Lexon contract into the intermediate syntax.

- Every statement in the Lexon contract is considered to be an enabling statement, and is translated based on the statement type.
  - Pay, Return, Increase and Decrease statements are converted to two subject statements.
  - Fix, Appoint and Certify statements are converted to one subject statements.
  - Type definitions (e.g. "PersonA is a 'person'") are ignored.
  - Be statements (e.g. "The Bet is terminated") are converted to definitions.
- If the contract contains recitals, the first statement in the recitals is conditional on a "Contract Active" substate.
- A distinct "Recitals Met" substate exists, conditional on the completion of the fulfilment of the last recital statement.
- The first statement of any clause or any "May" statements within it are dependent on a "Clause Invoked" substate being active.
- For every "May" statement in a clause, the first action of the statement is considered a permission, with all subsequent statements that fall within the same "May" statement are considered as obligations.
- The first statement of a clause is considered a permission, with any subsequent non-"May" statement being obligations.
- Any sequential statements in the recitals or a clause are modelled as statements dependent on the completion of the previous statement.

4.2.3 Translation of Lexon Contracts

The translation of Lexon into the intermediate syntax was achieved through the following methodology. The Lexon Compiler [4] was emplyoed to output the AST of a Lexon contract in JSON format into a text file. This text file is then processed by the Python API in Listing 15, which converts the contract into the corresponding representation in the intermediate syntax based on the rules set out in the previous sections.

The Lexon translation process was a significant hurdle in the project due to the multitude of conversion rules required to represent a Lexon contract as faithfully as possible in the Petri Net. The process was further complicated by the irregular structure of the Lexon AST when rendered in JSON format, an example of which can be found in Listing 11.
A particular complication introduced by its unconventional format relates to the determination of the subject of a statement. For instance, the recitals of the escrow agreement in Listing 1, *"The Payer pays an Amount into escrow, appoints the Payee, appoints the Arbiter..."*. Once transformed into the AST, this results in five distinct statements: (i) "Payer pays an Amount into escrow"; (ii) ","; (iii) "appoints the Payee"; (iv) ","; (v) "appoints the Arbiter". Here, a "," statement indicates that the next statement is part of a sequence of statements, each of which describes an action performed by the same subject as the previous statement. Crucially, only the initial statement identifies the primary subject of these actions, while subsequent statements lack any reference to the subject.

Python was the language of choice for converting the Lexon AST into the intermediate syntax, selected both for its handling of intricate data structures and for the author's familiarity with it. Although employing Lexon's source language, Rust, might have simplified the process, the author's lack of experience with Rust rendered this option impractical within the project's time frame.

4.2.4 Translating of CoLa Contracts

The translation of CoLa contracts proved to be a straightforward task because the intermediate syntax is based mainly on CoLa's formal syntax [2]. The conversion process was further simplified, as CoLa was designed for the expression of legal contracts, and therefore, unlike Lexon, no assumptions were required for the interpretation of a CoLa contract.

The methodology adopted to translate a contract written in CoLa into intermediate syntax is as follows: A Miranda script (CoLa source language) was developed. This script, available in Listing 13, utilises the AST output of the CoLa parser, transforming each contract component into its corresponding format within the intermediate syntax. Once all components had been converted to the appropriate format, the script produced Python code that represents the contract in the class-based implementation of the intermediate syntax. This code could then be incorporated into a Python file and executed to generate the appropriate Petri Net.

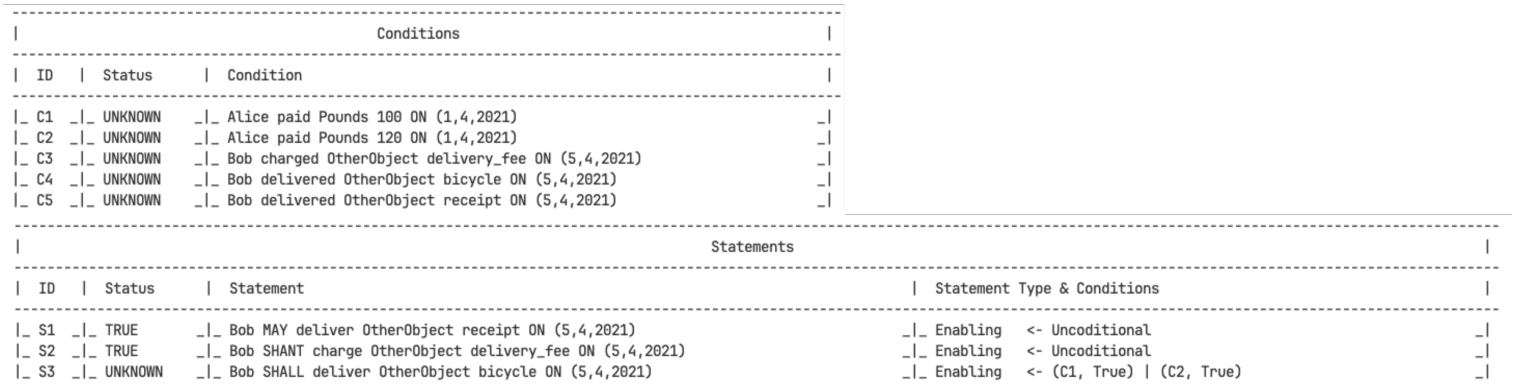
4.3 Petri Net Generation and Simulation

Upon conversion of the contract into the intermediate syntax, the Python API available in Appendix A.1.5, is used to generate the corresponding Petri Net, rendering all the required place nodes, transitions and arcs according to the logic laid out in Section 4.1. The implementation of the Petri Net in can be found in Listing 12.

Following the conversion process, the Python API enables contractual performance simulation. This is achieved by outputting a tabular representation of the various components of the Petri Net contract, as shown in Figure 4.10. Based on the components given by this output, the user can register an event by entering the ID of a condition from this table and designate a new value for this condition.

Once a condition has been updated, it's corresponding place node, will be updated in the Petri Net, providing it with the appropriate number of tokens, after which a breadth first search algorithm similar to Listing 5, was implemented to ensure that transitions

trigger in the correct order. Once the token flow has finished propagating, the updated Petri Net is rendered, and the updated state of the contract is printed out to the user in a new table. This process continues until the user cancels the simulation; thus, ending the simulated scenario.



Conditions		
ID	Status	Condition
C1	UNKNOWN	Alice paid Pounds 100 ON (1,4,2021)
C2	UNKNOWN	Alice paid Pounds 120 ON (1,4,2021)
C3	UNKNOWN	Bob charged OtherObject delivery_fee ON (5,4,2021)
C4	UNKNOWN	Bob delivered OtherObject bicycle ON (5,4,2021)
C5	UNKNOWN	Bob delivered OtherObject receipt ON (5,4,2021)

Statements			
ID	Status	Statement	Statement Type & Conditions
S1	TRUE	Bob MAY deliver OtherObject receipt ON (5,4,2021)	Enabling <- Uncoditional
S2	TRUE	Bob SHANT charge OtherObject delivery_fee ON (5,4,2021)	Enabling <- Uncoditional
S3	UNKNOWN	Bob SHALL deliver OtherObject bicycle ON (5,4,2021)	Enabling <- (C1, True) | (C2, True)

Figure 4.10: Screenshot of tabular output of components of the CoLa bike delivery agreement in Listing 9.

```
Function CascadeUpdate(placenode)
    Que = OutgoingEdges(placenode)
    While Que is not empty
        Arc = PopFirstItem(Que)
        Transition = TragetNode(Arc)
        If canFiretransition(Transition)
            FireTransition(Transition)
        End If
        Append(Que, OutgoingEdges(Transition))
    End While
End Function

Input: UpdatedConditionplacenode
fireplacenode(UpdatedConditionplacenode)
```

Listing 5: Pseudo code of BFS algorithm used for token propagation.

5 | Results and Evaluation

5.1 Verification

A functional testing approach was selected for project validation, as the various components were highly intertwined with each other. In total, 3 Lexon Contracts and 6 CoLa Contracts were converted into Petri Nets, each of which was tested against at least 3 different scenarios in performance simulation.

5.1.1 Contract Conversion

The contract conversion and Petri Net generation process were tested as follows. Contracts written in CoLa and Lexon were translated into intermediate syntax, from which the representative Petri Net was generated. The Python API used for Petri Net generation additionally generates an overview of all statements, definitions, and conditions within the contract, as shown in Figure 4.10. This overview is compared against the original source contract and AST, to ensure that all components and conditional links are present in the Petri Net. An example of the generated Petri Net used for a test can be seen in image (1) of Figure 5.1, with the extensive set of test cases available in Appendix A.2.

5.1.2 Performance Simulation

Performance simulation testing was carried out to ensure that the logical model was able to accurately represent the contract as a Petri Net. This was achieved by simulating each converted contract against a set of scenarios consisting of a series of events through an interactive text interface. After each event, the performative state of the contract highlighted by the updated Petri Net was compared against the expected state of the contract (i.e., expected outstanding obligations, permissions, and prohibitions). If the generated output matches the expected state of the contract at each state, then the Petri Net was considered to be a faithful representation of the contract. The complete set of test cases is available in the Appendix A.2, with an example simulation shown in Figure 5.1.

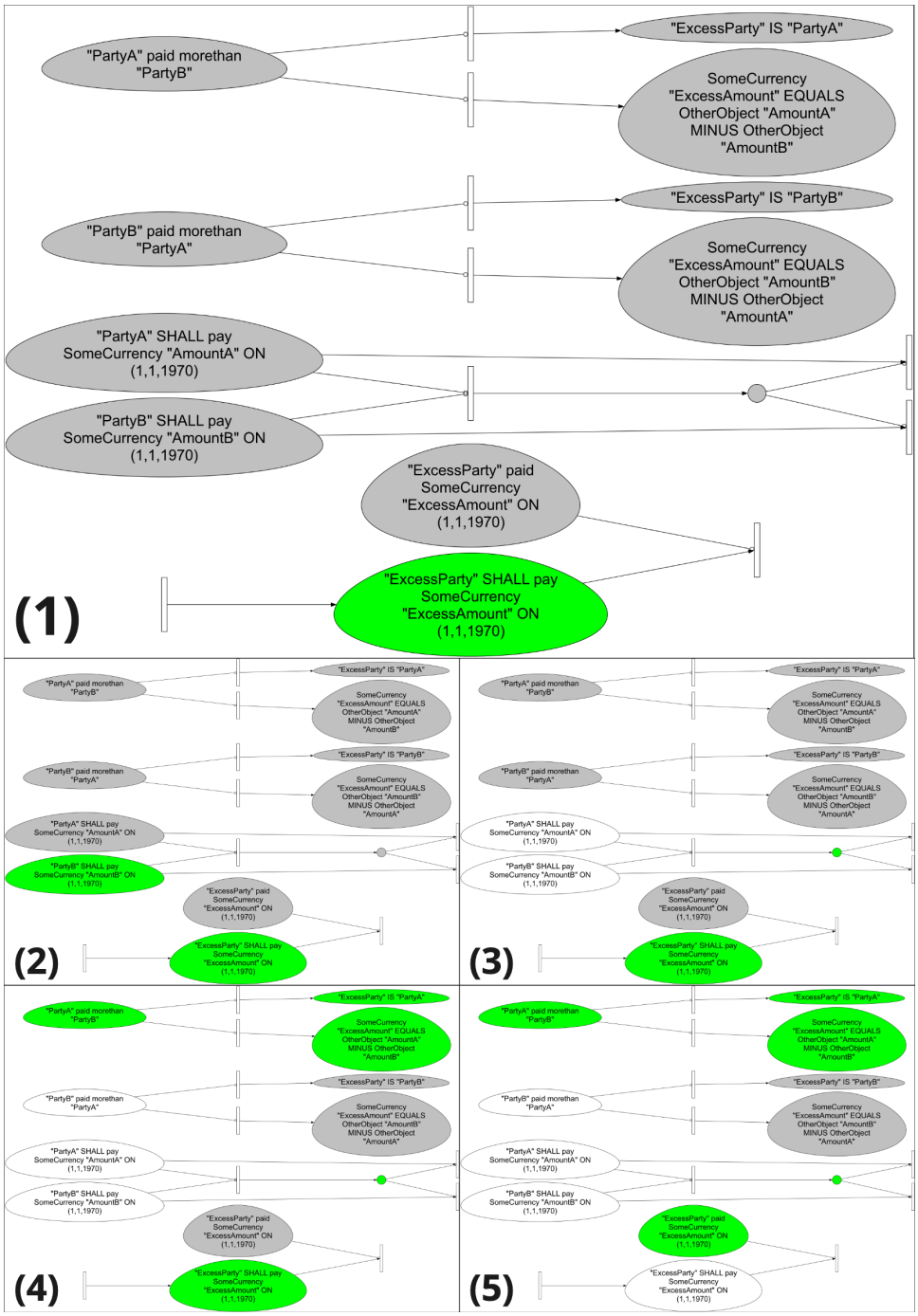


Figure 5.1: Performance simulation contract in Listing 3. (1) Initial state of the contract (2) State of contract after event declaring "PartyB shall pay ..." as True; (3) State of contract after event declaring condition "PartyA shall pay ..." as True; (4) State of contract after events declaring "PartyB paid more..." as False and "PartyA paid more..." as True; (5) State of contract after Event declaring "ExcessParty paid..." as True.

5.2 Critical Evaluation

The following section provides a critical evaluation of the methodology developed for contract conversion, representation of the contract as a Petri Net, and performance simulation.

5.2.1 Contract Conversion

The methodology developed in Section 4 allowed the successful translation of contracts written in CoLa and Lexon into an intermediate syntax that could be used to generate a representative Petri Net of the source contract. While the tests in Appendix A.2 found that all CoLa contracts were faithfully represented in the final Petri Net output, this was only partially the case for Lexon contracts. The Lexon contract conversion process suffered due to the many assumptions described in Section 4.2.2 and due to the use of the Lexon AST for syntax translation.
Consider the *Returnable Bet* Lexon contract in Listing 18. This contract contains the statement *"the Bet is deemed closed"*, whose representation in the Lexon AST is shown in Listing 6. Figure A.17 shows the Petri Net generated for this contract, where the statement is represented through a place node with the label *"Bet IS Undef"*. Although this interpretation is faithful to the Lexon AST representation of the contract, it does not capture the essence of the written contract text.

```
{
    "stmt": {
        "Be": {
            "def": "Undef",
            "expression": null}},
        "fillers": ["the "],
    "original": "the Bet is deemed Closed",
    "varnames": ["Bet"]
}
```

Listing 6: Excerpt of Lexon AST of UCC Financing Statement [3], showing AST representation of *"Bet is deemed Closed" as Undef.*

5.2.2 Petri Net Model

The model for representing a contract as a Petri Net proposed in Section 4 resulted in a Petri Net that can faithfully represent simple agreements. However, the model currently suffers from a variety of limitations.

One such limitation is due to the model being designed to produce a Petri Net that is accessible and easy to understand, leading to introduction of read and inhibitor arcs in Section 4.1.2 to reduce the number of arcs and nodes in the system. Specifically, the introduction of read-only arcs for conjunction and disjunction conditions resulted in a potential problem where once a conjunction or disjunction condition place node holds a token, it will indefinitely maintain it. An example of this can be observed in image (3) of Figure 5.1, as both obligations are voided, the place node of the 'and' condition continues to hold the token when logically it should not.

Various implementations were considered to address this issue, one of which was the use of normal arcs to connect conjunction and disjunction conditions to transition and thus consuming the token if the transition fires. This approach is unsuitable, as it causes the token of the condition to be consumed by either of the connected voiding transitions, rendering the other voiding transitions unable to fire, and thus only one of the two obligations being voided. Another solution to this problem is the inclusion of a sink transitions for conjunction and disjunction conditions, as shown in Figure 5.2. While this would result in the desired effect and remove the token from the conjunction place node, the inclusion of such a sink transition would cause a significant increase in the number of nodes and arcs, reducing the overall readability of the Petri Net.

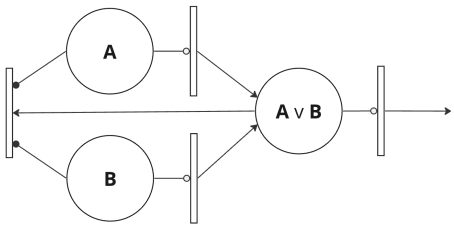


Figure 5.2: Example of a sink transition for an OR condition. The conditional place node $(A \vee B)$ is connected to sink transition enabled if $((A \vee B) \wedge \neg A \wedge \neg B)$.

It is worth noting that while this issue causes a condition to incorrectly hold a token in specific cases, the inclusion of voiding and fulfilment transitions cause this to not have an effect on the performance simulation. Figure A.8 offers such an example, as where the and condition holds the token when it should not but the statement enabled by the condition *"ExcessParty SHALL pay ..."* will not be falsely enabled by this as the voiding transition causes any token at the statement place node to be immediately consumed.

5.2.3 Performance Simulation

The performance simulation of a contractual Petri Net as shown in Figure 5.1 provides insightful information on the performance state of a contract, clearly highlighting the known facts (i.e. value of conditions) and expectations of each party given a series of events. However, the current implementation suffers from several issues.

The registration process currently only allows for the registration of one event at a time, assuming that the events do not occur simultaneously. This assumption is flawed, as our implementation of an event is the process of declaring a fact to be true or false, i.e. *"PartyA paid SomeAmount..."* or *"PartyA paid more than PartyB"*. The flaw in this becomes apparent in the ISDA master agreement in Figure 5.1, which has the two place nodes *"PartyA paid more than PartyB"* and *"PartyB paid more than PartyA"*. When it is established which party paid more, both statements should be updated simultaneously in a single event.

Furthermore, the simulation process currently does not have the ability to prevent illogical inputs registered by the user. This is highlighted in the simulation of the ISDA master agreement in Figure A.2 where two mutually exclusive facts, *"PartyA paid more than PartyB"* and *"PartyB paid more than PartyB"*, were declared true. However, such contradictory declarations result from incorrect input by the oracle; they are not inherent flaws of the model itself.

Lastly, the model was designed to not infer that which is not specified, e.g. a default state if an obligation is not met. This means that actions can be registered out of sequence, for example, Figure 5.5 where one could register the condition *"Bob delivered bicycle....."* as

true even if the corresponding obligation does not stand without the system flagging an error to the user. It is so far unclear whether this is a desirable behaviour of the model, as the scenario where Bob delivers a bicycle even though he is not obligated to do so, while unlikely, can occur. It is not possible for us to state what the consequences of such a scenario are, unless the contract itself mentions it.

5.3 Controlled Natural Languages and the Legal Bug

A shift towards computable contracts expressed through a controlled natural languages (CNL) promises various advantages. One of these advantages could be easing the discovery and correction of what we will call the "legal bug".

In the realm of software, a "bug" denotes an error or flaw within a computer programme. The implications of these bugs can vary in severity from substantial errors that compromise the system's entire logic, like neglecting to account for an edge case, to minor glitches that do not disrupt the overarching logic. A notable instance of such a bug led to the Ariane 5 rocket crash in 1996, incurring a loss exceeding US $370 million [29].

A similar issue can be found in legal contracts, where nuanced variations in language can drastically alter, and occasionally misconstrue, the intended meaning of a statement. This could be observed in a Canadian legal case, where a single comma in a 14-page contract caused the wrongful interpretation of a cancellation clause, costing Rogers Communications close to 1 million Canadian dollars [30].

Employing CNL in contract drafting does not eliminate the risk of introducing such "legal bugs" into a contract. Paradoxically, the use of a CNL might even amplify the dangers of ill-crafted contracts because of its strict interpretive nature leaving little room for disputing the deontic logic based on ambiguous phrasing. However, the benefit of the CNL becomes apparent when integrated with the developed Petri Net model. Within this framework, "legal bugs" that might escape human scrutiny in textual form, but diverge from the contracts intended outcome, can be spotted through the visualisation and performance simulation of the contract.

During the analysis, two potential "legal bugs" in the translated CoLa contracts were detected. The subsequent section will elaborate on these, highlighting how subtle linguistic missteps can inadvertently alter or uphold specific permissions or obligations and how these mistakes become more apparent with Petri Net visualisation.

5.3.1 ISDA Master Agreement

Fattals interpretation of the ISDA master agreement in CoLa can be seen in Listing 3. This agreement produces the Petri Net shown in Figure 5.3 and demonstrates the existence of what could be considered a "legal bug", namely, that without any other facts or knowledge the original contract states that the obligations of the ExcessParty to pay the ExcessAmount holds no matter what.

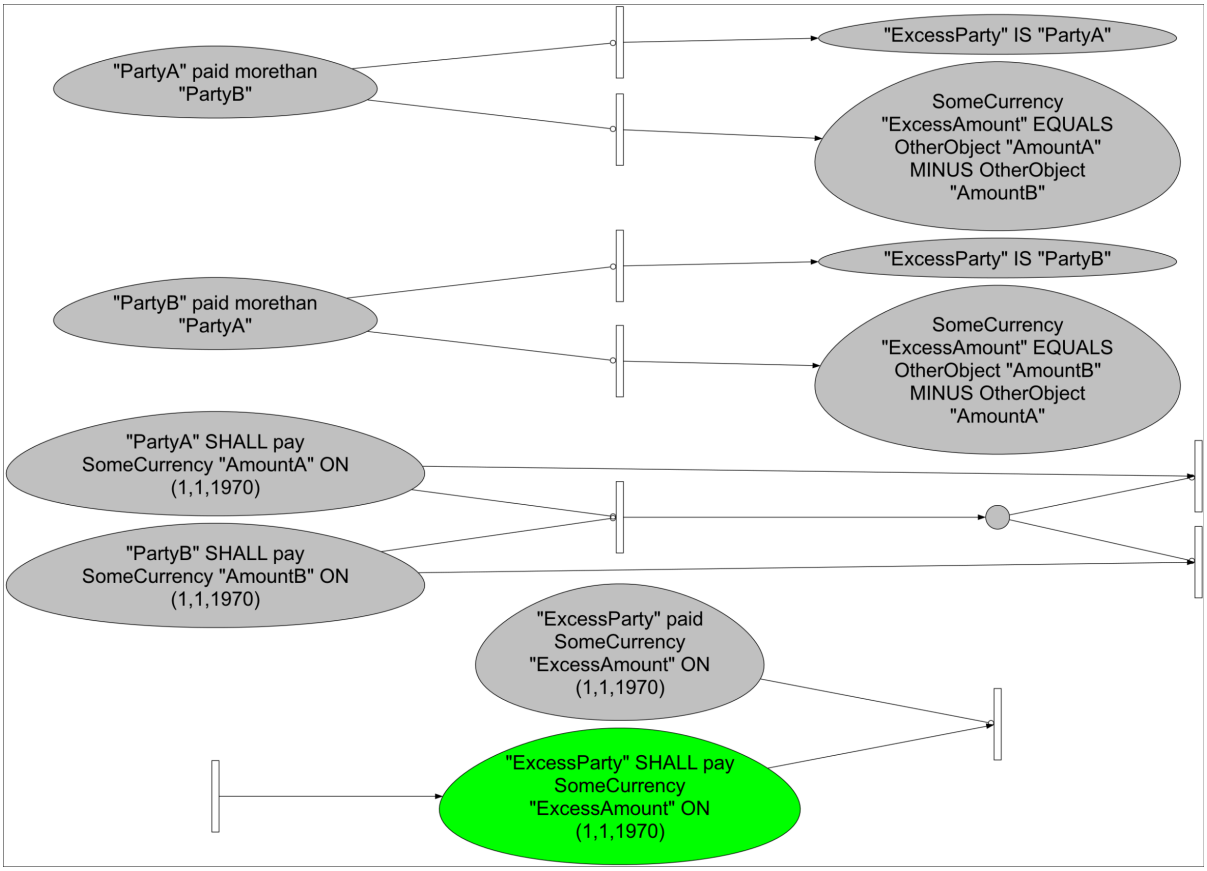


Figure 5.3: Petri Net of ISDA master agreement by Fattal [2] highlighting a "legal bug" in the form of an unconditional obligation for the ExcessParty.

The cause of this is highlighted in Listing 7, as according to the parsing principles of CoLa, component [5] is interpreted as a separate component and therefore unlike component [3] and [4], it is not conditional on conditions [1] and [2].

```
IF  [1] it is the case that PartyA shall pay AMOUNT 'A'  on the 01 January 1970
        AND
    [2] it is the case that PartyB shall pay AMOUNT 'B' on  on the 01 January 1970
THEN[3] it is not the case that PartyA shall pay AMOUNT 'A' on  on the 01 January
↪  1970
        AND
    [4] it is not the case that PartyB shall pay AMOUNT 'B' on  on the 01 January
    ↪  1970
C-AND
    [5] it is the case that ExcessParty shall pay AMOUNT "ExcessAmount" on  on the
    ↪  01 January 1970
C-AND
...
```

Listing 7: Excerpt of the CoLa interpretation of the ISDA master agreement by [2].

Whether it is intended that the obligation of the ExcessParty stands unconditionally regardless of the previous actions, is ultimately dependent on the intent of the person drafting the contract.

For illustration purposes, however, the same contract will be expressed with a slight change in wording. By changing the *C-AND* [5] into an *AND* [5] in Listing 8, the ExcessPartys obligation is made dependent on the previous conditionals, thus changing the underlying logic of the contract, as can be seen in Figure 5.4.

```
IF  [1] it is the case that PartyA shall pay AMOUNT 'A' on the 01 January 1970
        AND
    [2] it is the case that PartyB shall pay AMOUNT 'B' on the 01 January 1970
THEN[3] it is not the case that PartyA shall pay AMOUNT 'A' on the 01 January 1970
        AND
    [4] it is not the case that PartyB shall pay AMOUNT 'B' on the 01 January 1970
        AND
    [5] it is the case that ExcessParty shall pay AMOUNT "ExcessAmount" on the 01
    ↪  January 1970
```

```
9  C-AND
10 ...
```

Listing 8: Excerpt of modified ISDA master agreement from Listing 7, making component [5] conditional by changing the C-AND to an AND in line 7.

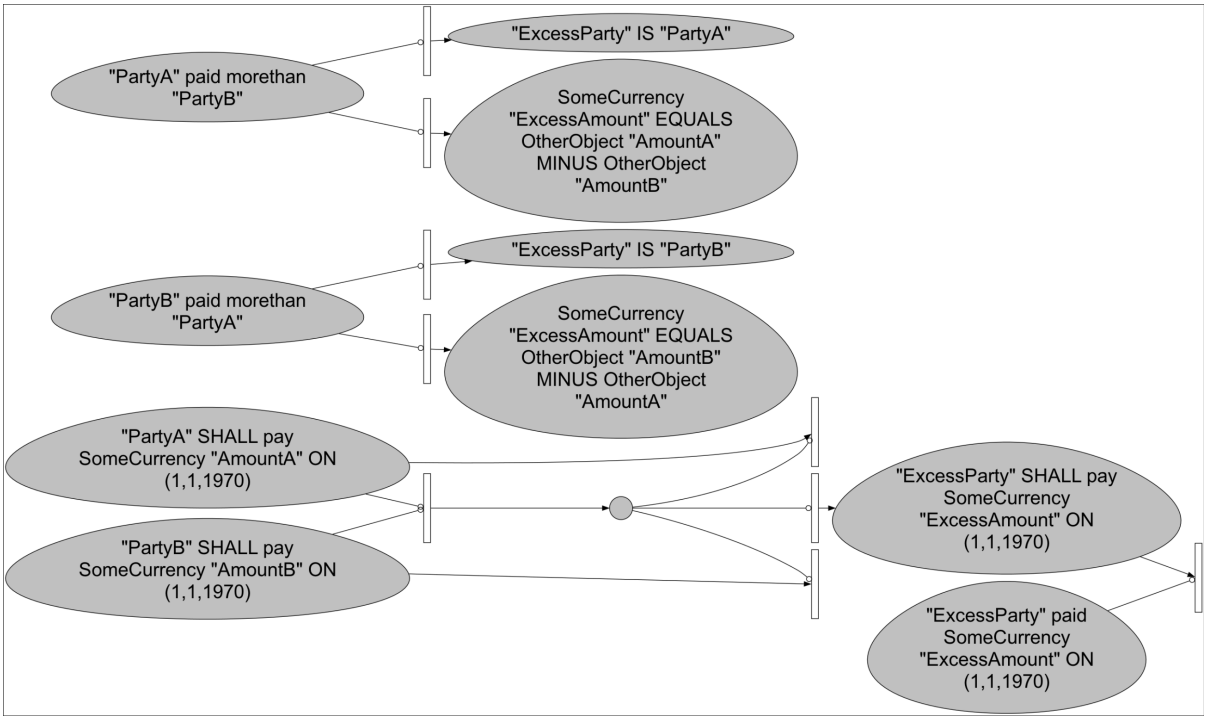


Figure 5.4: Petri Net of modified ISDA master agreement in Listing 8, removing the unconditional obligation.

For a human reader, keywords such as *C-AND* and*AND* may seem similar enough to avoid appearing suspicious at first glance, but the impact made by the choice of language on the general logic of the contract is significant.

5.3.2 Bike Delivery

Another possible "legal bug" could be found in the contract in Listing 9 , obtained from the CoLa uni tests [2].

```
1  IF  [1a] it is the case that Alice paid 100 pounds on the 1 April 2021
2          OR
```

```
3       [1b] it is the case that Alice paid 120 pounds on the 1 April 2021
4   THEN[2] it is the case that Bob must deliver a bicycle on the 5 April 2021
5   C-AND
6       [3a] it is the case that Bob may deliver a receipt on the 5 April 2021
7           AND
8       [3b] it is the case that Bob is forbidden to charge a delivery_fee on the 5
        ↪  April 2021.
```

Listing 9: CoLa contract describing a simple bike delivery [2].

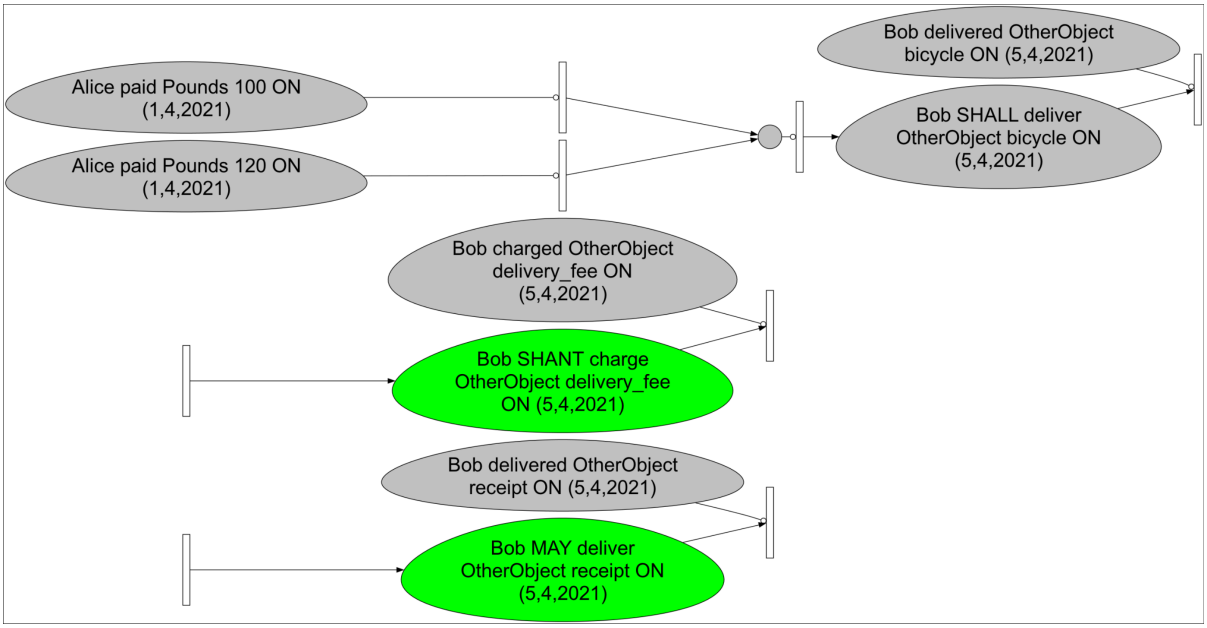


Figure 5.5: Petri Net of simple bike delivery in Listing 9, highlighting potential legal bug through unconditional expectations.

Figure 5.5 illustrates the Petri Net generated for this contract, where it could be argued that it does not make sense to have permission to deliver a receipt if Alice did not pay the funds and Bob did not need to deliver the bike. It might make more sense to make the permission to deliver the receipt and the prohibition of charging a delivery fee conditional on Alice paying. In Listing 10 this modified contract is presented, with a change in wording from *C-AND* to *AND* in line 5 and 7.

The Petri Net resulting from this modified contract, shown in Figure 5.6, clearly shows that the updated wording caused a significant change in the overall logic of the contract. Once again, this raises the question of which contract is the correct one, a question which

can only be answered by the person drafting the contract in accordance with their desires.

```
IF  [1a] it is the case that Alice paid 100 pounds on the 1 April 2021
        OR
    [1b] it is the case that Alice paid 120 pounds on the 1 April 2021
THEN[2] it is the case that Bob must deliver a bicycle on the 5 April 2021
        AND
    [3a] it is the case that Bob may deliver a receipt on the 5 April 2021
        AND
    [3b] it is the case that Bob is forbidden to charge a delivery_fee on the 5
    ↪  April 2021
```

Listing 10: Modified bike delivery agreement, making all statements conditional.

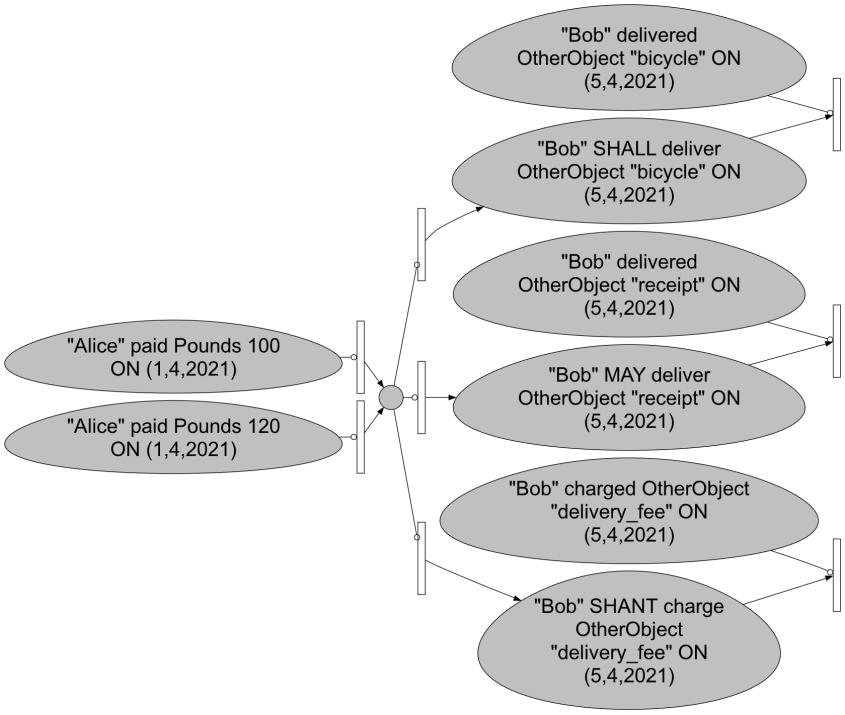


Figure 5.6: Petri Net of modified bike delivery agreement in Listing 10.

6 | Conclusion

6.1 Project Evaluation

All core objectives for the project, laid out in Section 1.4, have been met. A comprehensive review of the literature on the state of computable contracting is presented in Section 2.3. The literature review covered the languages used for the drafting of smart and computable contracts and highlighted some previously unaddressed research questions. Section 3.1 summarised the research gap that the logical representation of CNL contracts as Petri Nets aims to bridge, followed by an outline of the requirements for the logical contract model in Section 3.2.

Section 4.1 adresses the creation of the logical contract model as a Petri Net, highlighting the challenges, design requirements, and implementation details of the undertaking. Section 4.2.1.1 saw the introduction of a formal syntax used for Petri Net generation, capable of expressing the linguistic features of both source languages. The methodology employed for the translation of CoLa and Lexon written contracts from their respective ASTs was detailed in Section 4.2.4 and Section 4.2.2 with details of the Petri Net generation and performance simulation process given in Section 4.3.

Section 5.1, covered the test strategy implemented to evaluate the project success. A critical evaluation of the project was conducted in Section 5.2, finding that while the developed methodology was able to convert contracts written in either source language into a Petri Net, the final output was completely faithful to the source contract only for CoLa written contracts. The section also detailed how the conversion process of Lexon, due to the many assumptions required for it and the use of its AST for syntax translation, lead to results that are not always entirely faithful in representing the spirit of the source contract. The section further discussed the successful implementation of the Performance Simulation of contracts as Petri Nets. This simulation resulted in an output from which the underlying logic of a contract and the outstanding expectations for each party during contract performance could be understood. Section 5.3 expanded on a potential benefit of using the developed model for contract drafting, highlighting how the visualisation and performance simulation of a contract can enable the spotting

of what is termed "legal bugs", where misssteps in semantics, which might espace human interpretation, can significantly and adversally affect a contracts logic.

6.2 Future Work

A major challenge encountered during the project was the use of the selected source languages. While CoLa offers insight into what a CNL for lawyers might resemble, it is not mature enough to encompass diverse legal notations and concepts. Similarly, Lexon's design, with its focus on actions verifiable on the Blockchain, limits the legal concepts that can be cleanly expressed in it. A more refined CNL, tailored towards computable contracts and legal professionals, is required to allow for the expression of more complicated real-world contracts. Furthermore, a model based on a single source language could be further optimised and streamlined as it would not have to encompass the unique concepts and functionality expressed in two different languages.

There are possibilities to enhance the existing model, amplifying its versatility and functionality. For instance, modifying Place Nodes to support multiple tokens at once may allow complex event representations, such as "*If Party A misses payments 3 times, then...*". Here, the place node might denote *"Party A missed payment"*, and be connected to the Transition through an arc weighted at 3. This would require the Place Node to obtain 3 tokens before the Transition can fire. Furthermore, an exploration of the trade-off between model capability and output readability is critical to find suitable solutions to issues such as wrongful keeping of tokens for conjunction and disjunction place nodes, described in Section 5.2.2. This section also describes a limitation of the current model, where events can only update the value of one condition at a time.

The projects' central ambition was to fascilitate the automatic representation of a CNL contract's geometry through a Petri Net. This objective was achieved, laying a foundation on which further applications could be built. Such applications could include the integration of a theorem prover, avoiding illogical user inputs, as detailed in Section 5.2.3, and a knowledge service to autonomously register events and update node values. Not only could this facilitate automated clarification regarding the expected behaviour of parties during contract performance, but might also pave the way for semi or fully

automated performance mechanisms.

Lastly, there are several opportunities to develop methodologies for contractual analysis using the model. Such analysis may include the detection of inconsistencies, the comparison of clauses using different semantics, or even the application of machine learning algorithms aimed at simulating event chains, pinpointing overlooked scenarios or potential contractual loopholes.

Bibliography

[1] R. M. Lee, "A logic model for electronic contracting," *Decision Support Systems*, vol. 4, no. 1, pp. 27–44, 1988.

[2] S. Fattal, "CoLa: A Controlled Natural Language for Computable Contracting," MEng Dissertation, University College London, Department of Computer Science, May 2021.

[3] H. Diedrich, "Lexon Vocabulary," Apr. 2023. Available at http://lexon.org/vocabulary.html#examples.

[4] "Lexon Compiler," Jan. 2020. Available at https://gitlab.com/lexon-foundation/lexon-rust.

[5] "Lexon Online Compiler." Available at http://3.lexon.tech/.

[6] J. Cummins and C. D. Clack, "Transforming commercial contracts through computable contracting," *Journal of Strategic Contracting and Negotiation*, vol. 6, pp. 3–25, Mar. 2022. Publisher: SAGE Publications.

[7] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart Contract Templates: foundations, design landscape and research directions," Mar. 2017. arXiv:1608.00771 [cs].

[8] F. Idelberger, "Merging traditional contracts (or law) and (smart) e-contracts a novel approach,"

[9] C. D. Clack, "Languages for Smart and Computable Contracts," Apr. 2021. arXiv:2104.03764 [cs].

[10] H. Diedrich, "Lexon, Legal Smart Contracts," Sept. 2017. Originally available at https://lexon.org/lexon-whitepaper-2017.pdf. URL no longer accessible.

[11] M. D. Flood and O. R. Goodenough, "Contract as automaton: representing a simple financial agreement in computational form," *Artificial Intelligence and Law*, vol. 30, pp. 391–416, Sept. 2022.

[12] H. Diedrich, *Lexon Book*. No. 0.3.5.9.3 in Draft 3, Jan. 2020. Available at https://lexon.org/docs/Lexon%20Book%202020.pdf.

[13] C. A. Petri, *Communication with automata*. PhD thesis, University of Bonn, 1966.

[14] U. Bielefeld, "Petri Nets." Available at https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html.

[15] P. Baldan, N. Busi, A. Corradini, and G. M. Pinna, "Domain and event structure semantics for Petri nets with read and inhibitor arcs," *Theoretical Computer Science*, vol. 323, pp. 129–189, Sept. 2004.

[16] R. Rahman, K. Liu, and L. Kagal, "From Legal Agreements to Blockchain Smart Contracts," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–5, May 2020.

[17] R. A. Kowalski, "Legislation as Logic Programs," in *Informatics and the Foundations of Legal Reasoning* (Z. Bankowski, I. White, and U. Hahn, eds.), Law and Philosophy Library, pp. 325–356, Dordrecht: Springer Netherlands, 1995.

[18] H. N. Castaneda, "The Logic of Change, Action, and Norms," *The Journal of Philosophy*, vol. 62, no. 13, pp. 333–344, 1965. Publisher: Journal of Philosophy, Inc.

[19] L. W. Group, S. Salkind, and O. R. Goodenough, "Legal Specification Protocol (LSP) Working Group White Paper," Feb. 2019. Available at https://law.stanford.edu/publications/developing-a-legal-specification-protocol-technological-considerations-and-requirements/.

[20] H. Haapio and J. Hazard, *Wise Contracts: Smart Contracts that Work for People and Machines*. Feb. 2017.

[21] I. Grigg, "The Ricardian Contract." Available at https://iang.org/papers/ricardian_contract.html.

[22] "Accord Project." Available at https://accordproject.org/.

[23] "OASIS LegalXML Electronic Court Filing TC | OASIS." Available at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=legalxml-courtfiling#overview.

[24] D. Merigoux, N. Chataing, and J. Protzenko, "Catala: A Programming Language for the Law," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, p. 77:1, 2021. Conference Name: International Conference on Functional Programming.

[25] M. Wong, "L4 DSL." Available at https://github.com/smucclaw/dsl.

[26] D. K. Joshi and D. T. W. Lauer, "Impact of information technology on users' work environment: A case of computer aided design (CAD) system implementation," *Information & Management*, vol. 34, pp. 349–360, Dec. 1998.

[27] J. Wang, "Petri Nets for Dynamic Event-Driven System Modeling," *Handbook of Dynamic System Modeling*, Jan. 2007.

[28] O. Ethereum, "Introduction to Smart Contracts â Solidity 0.8.21 documentation." Available at https://docs.soliditylang.org/en/v0.8.21/introduction-to-smart-contracts.html.

[29] M. Dowson, "The Ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes*, vol. 22, p. 84, Mar. 1997. Available at https://dl.acm.org/doi/10.1145/251880.251992.

[30] I. Austen, "The Comma That Costs 1 Million Dollars (Canadian)," *The New York Times*, Oct. 2006. Available at https://www.nytimes.com/2006/10/25/business/worldbusiness/25comma.html.

A | Appendix

```
[{"name": "Certify",
    "payable": false,
    "statements": [{"stmt": {
          "May": ["filing_office", [{
                "stmt": { "Certify": ["", ["File_Number"]]
                },
                "fillers": [],
                "original": "certify the File_Number",
                "varnames": []}]]},
        "fillers": ["The "],
        "original": "The Filing_Office may certify the File_Number.",
        "varnames": []}],
    "ret": []},
  { "name": "Set_File_Date",
    "payable": false,
    "statements": [{
        "stmt": {
          "May": ["filing_office",[{
                "stmt": {"Fix": {
                    "ops": [],"terms": [{
                        "ops": [], "factors": [{"Time": "now"}]}]}},
                "fillers": ["the "],
                "original": "fix the Initial_Statement_Date as the current time",
                "varnames": ["Initial_Statement_Date"]}]]},
        "fillers": ["The "],
        "original": "The Filing_Office may fix the Initial_Statement_Date as the
↪  current time.",
        "varnames": []}],
    "ret": []},
  {"name": "Set_Lapse",
    "payable": false,
    "statements": [
      {"stmt": {"May": ["filing_office", [{
                "stmt": {"Fix": {
                    "ops": [], "terms": [{"ops": [], "factors": [
                          {"Sym": {
                              "sym": "now",
                              "given": false,
                              "filler": ""}}]}]}},
                "fillers": ["the "],
                "original": "fix the Lapse_Date as now",
                "varnames": ["Lapse_Date"]}]]},
        "fillers": ["The "],
        "original": "The Filing_Office may fix the Lapse_Date as now.",
        "varnames": []}],
    "ret": []}]
```

Listing 11: Excerpt of Lexon UCC Financing Statement Contract (Listing 19) AST in JSON format.

A.1 Source Code

A.1.1 Petri Net

```python
import collections
import textwrap
import datetime
from graphviz import Digraph

def node_from_list(node_label: str, node_list: list) -> "Place | Transition":
    """Get node object from list based on node label"""
    ind = [n._label for n in node_list].index(node_label)
    return node_list[ind]

class Edge:
    """Class representing a connection edge in a Petri net."""

    def __init__(self, from_node: "Place | Transition", to_node: "Place |
    ↪  Transition", weight: int = 1):
        self._from_node = from_node
        self._to_node = to_node
        self.weight = weight
        self._hash_id = Edge._hash(from_node, to_node)
        self.arrow_head = "normal"

    def _type(self) -> str:
        """Returns the type of the edge."""
        return "connect"

    def __repr__(self) -> str:
        """Returns a string representation of the edge."""
        return self._hash_id

    @staticmethod
    def _hash(from_node, to_node) -> str:
        """Generate a hash ID for an edge to ensure edge ids are unique."""
        return f"C - {from_node._id} - {to_node._id}"

    @staticmethod
    def connect(from_node, to_node):
        """Connect nodes with an edge if they aren't already connected."""
        edge_hash = Edge._hash(from_node, to_node)
        if edge_hash not in [e._hash_id for e in from_node._out_edges]:
            edge = Edge(from_node, to_node)
            from_node._out_edges.append(edge)
            to_node._in_edges.append(edge)

```

```
43      def canFire(self) -> bool:
44          """Check if the edge can be triggered."""
45          return self._from_node.canFire()
46
47      def active(self) -> bool:
48          """Check if the edge is active."""
49          return self._from_node.active()
50
51
52  class InhibitorEdge(Edge):
53      def __init__(self, from_node: "Node | Place",
54                   to_node: "Node | Transition", weight: int = 1):
55          super().__init__(from_node, to_node, weight)  # type:ignore
56          self.arrow_head = "dot"
57
58      def _type(self) -> str: return "inhibit"
59
60      def canFire(self) -> bool:
61          """Check if the edge can be triggered."""
62          return self._from_node.canFire() == False and self._from_node.active() ==
            ↪  True
63
64      @staticmethod
65      def _hash(from_node, to_node) -> str:
66          """Generate a hash ID for an edge to ensure edge ids are unique."""
67          return f"I - {from_node._id} - {to_node._id}"
68
69      @staticmethod
70      def connect(from_node: "Node | Place",
71                  to_node: "Transition | Place") -> None:
72          """Connect nodes with an edge if they aren't already connected."""
73          edge_hash = InhibitorEdge._hash(from_node, to_node)
74          if edge_hash not in [e._hash_id for e in from_node._out_edges]:
75              edge = InhibitorEdge(from_node, to_node)
76              from_node._out_edges.append(edge)
77              to_node._in_edges.append(edge)
78
79  class ReadEdge(Edge):
80      def __init__(self, from_node: "Node | Place",
81                   to_node: "Node |Transition") -> None:
82          super().__init__(from_node, to_node)  # type:ignore
83          self.arrow_head = "odot"
84
85      def _type(self) -> str: return "read"
86
87      @staticmethod
88      def _hash(from_node, to_node) -> str:
89          """Generate a hash ID for an edge to ensure edge ids are unique."""
90          return f"R - {from_node._id} - {to_node._id}"
```

```
91
92      @staticmethod
93      def connect(from_node: "Place", to_node: "Transition | Place") -> None:
94          """Connect nodes with an edge if they aren't already connected."""
95          edge_hash = ReadEdge._hash(from_node, to_node)
96          if edge_hash not in [e._hash_id for e in from_node._out_edges]:
97              edge = ReadEdge(from_node, to_node)
98              from_node._out_edges.append(edge)
99              to_node._in_edges.append(edge)
100
101
102 class Node:
103     """Base class representing a node in a Petri net."""
104     cnt = 0  # Counter for unique node IDs
105
106     def __init__(self, label: str, in_edges: list, out_edges: list, active=True):
107         self._id = Node.cnt
108         self._label = label
109         self._active = active
110         self._in_edges = in_edges if in_edges else []
111         self._out_edges = out_edges if out_edges else []
112         self._color = 'white'
113
114         Node.cnt += 1
115
116     def connect(self, target):
117         """Connect to another node through normal edge"""
118         Edge.connect(self, target)
119
120     def read(self, target):
121         """Connect to another node through read edge"""
122         ReadEdge.connect(self, target)  # type:ignore
123
124     def inhibit(self, target):
125         """Connect to another node through inhibitor edge"""
126         InhibitorEdge.connect(self, target)
127
128
129 class Place(Node):
130     def __init__(self, label, in_edges: list[Edge],
131                  out_edges: list[Edge | ReadEdge | InhibitorEdge],
132                  tokens=-1, substate: bool = False, show_label: bool = True):
133         super().__init__(label, in_edges, out_edges)
134         self._tokens = tokens
135         self._substate = substate
136         self._show_label = show_label
137
138     def shape(self) -> str:
139         """Return the shape of the place."""
```

```
140             return "egg" if self._show_label else "circle"  # type:ignore
141
142         def canFire(self) -> bool:
143             """Check if the place can be triggered."""
144             return self._tokens >= 1
145
146         def active(self) -> bool:
147             """Check if the place is active."""
148             return self._tokens >= 0
149
150         def setTrue(self) -> None:
151             """Set the place to have a token."""
152             self._tokens = 1
153
154         def setFalse(self) -> None:
155             """Set the place to not have a token."""
156             self._tokens = 0
157
158         def setUnknown(self) -> None:
159             """Set the place to have an unknown number of tokens."""
160             self._tokens = -1
161
162         def color(self) -> str:
163             """Return the color of the place."""
164             if self._tokens <= -1:
165                 return "grey"
166             elif self._tokens == 0:
167                 return "white"
168             else:
169                 return "green"
170
171         def label(self):
172             """For visual clarity, format the label to be within a 30 character
                ↪  bounding box"""
173             if not self._show_label:
174                 return ""
175             label = self._label
176             if len(label) > 30:
177                 label = textwrap.fill(label, 25, break_long_words=False)
178
179             return label
180
181         def fire(self):
182             """BFS traversal to fire outgoing transitions, if possible"""
183             Q = collections.deque([e for e in self._out_edges])
184             while Q:
185                 edge = Q.popleft()
186                 transition = edge._to_node
187                 if transition.canFire():
```

```
188                 transition.fire()
189
190             # add next set of edges to Q
191             Q.extend([e for e in transition._out_edges])
192
193
194 class Transition(Node):
195     def __init__(self, label: str, in_edges: list[Edge | ReadEdge | InhibitorEdge],
196                  out_edges: list[Edge]):
197         super().__init__(label, in_edges, out_edges)
198         # self.shape = "box"
199
200     def shape(self) -> str:
201         """Return the shape of the transition."""
202         return "rectangle"
203
204     def canFire(self) -> bool:
205         """Check if the transition can be triggered."""
206         return all([e.canFire() for e in self._in_edges]) or self._in_edges == []
207
208     def active(self) -> bool:
209         """Check if the transition is active."""
210         return all([e.active() for e in self._in_edges]) or self._in_edges == []
211
212     def fire(self):
213         if self.canFire():
214             for e in self._out_edges:
215                 e._to_node.setTrue()
216
217             for e in self._in_edges:
218                 # Only consume tokens if the edge is a edge and not a read or
                    ↪  inhibitor edge
219                 if e._type() == "connect":
220                     e._from_node.setFalse()
221
222
223 class PetriNet:
224     """Class representing a Petri net."""
225
226     def __init__(self):
227         self.places = []
228         self.transitions = []
229
230     def place(self, label: str, show_label=True) -> Place:
231         """Get or create a place in the Petri net with the given label."""
232         if any([p._label == label for p in self.places]):
233             return node_from_list(label, self.places)  # type: ignore
234         else:
235             place = Place(label, in_edges=[], out_edges=[], show_label=show_label)
```

```
236             self.places.append(place)
237             return place
238 
239     def transition(self, label: str) -> Transition:
240         """Get or create a transition in the Petri net with the given label."""
241         if any([t._label == label for t in self.transitions]):
242             return node_from_list(label, self.transitions)  # type: ignore
243         else:
244             tran = Transition(label, in_edges=[], out_edges=[])
245             self.transitions.append(tran)
246             return tran
247 
248     @staticmethod
249     def draw_edge(dot, edges_drawn, edge: Edge):
250         """Draw an edge if it hasn't been drawn already and add it to the list of
        ↪  drawn edges."""
251         if edge._hash_id in edges_drawn:
252             return
253         dot.edge(str(edge._from_node._id), str(edge._to_node._id),
        ↪  arrowhead=edge.arrow_head)
254         edges_drawn.append(edge._hash_id)
255 
256     def assemble(self, engine: str = 'dot'):
257         dot = Digraph(comment='PetriNet')
258         dot.engine = engine
259         dot.graph_attr['overlap'] = 'false'
260         edges_drawn = []
261 
262         # Draw places nodes
263         for place in self.places:
264             dot.node(str(place._id),
265                      label=place.label(),
266                      style="filled",
267                      fillcolor=place.color(),
268                      shape=place.shape(),
269                      fontsize='30', fontname="Arial", fontweight='bold')
270 
271             # Draw edges for place
272             for edge in place._in_edges:
273                 PetriNet.draw_edge(dot, edges_drawn, edge)
274             for edge in place._out_edges:
275                 PetriNet.draw_edge(dot, edges_drawn, edge)
276 
277         # Draw transition nodes
278         for transition in self.transitions:
279             # Draw transition nodes, only if they have least one connection
280             if transition._in_edges == [] and transition._out_edges == []:
281                 continue
282 
```

```
283             dot.node(str(transition._id),
284                      label="",
285                      style="filled",
286                      fillcolor=transition._color,
287                      height="1.5",
288                      width="0.15",
289                      shape=transition.shape())
290 
291             # Draw Edges for transition
292             for edge in transition._in_edges:
293                 PetriNet.draw_edge(dot, edges_drawn, edge)
294             for edge in transition._out_edges:
295                 PetriNet.draw_edge(dot, edges_drawn, edge)
296 
297         return dot
298 
299     def fire_transitions(self):
300         """Trigger all transitions in Petri Net.
301          Used for initial value evaluation (i.e. unconditional transitions)"""
302         for transition in self.transitions:
303             transition.fire()
304 
305     def render(self):
306         """Render the Petri net using Graphviz."""
307         dot = self.assemble()
308         # Make the graph left to right (instead of top to bottom)
309         dot.attr(rankdir='LR')
310         # Reduce the spacing between nodes and edges for visual clarity
311         dot.attr(nodesep='0.15')
312         dot.attr(ranksep='0.3')
313         dot.attr(newrank='true')
314         # Make edges thicker for visual clarity
315         dot.attr(penwidth='2')
316         dot.format = 'png'
317         # Add timestamp to filename to avoid overwriting
318         filename = f"outputs/_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}"
319         # Render the graph
320         dot.render(filename, view=True, cleanup=True)
```

Listing 12: Python Implementation of Petri Net for contract representation.

A.1.2 CoLa Syntax Translation

```
|| Type Definitions for intermediate syntax
|| Types starting with t_... are types from Fattals CoLa implementation
|| Types starting with m_... are new types
m_test ::= TRUE | FALSE

m_modal_verb ::= M_SHALL | M_SHANT | M_MAY

m_statement ::= M_TemporalActionStatement(m_test, t_subject, m_modal_verb, t_verb,
↪  t_object, t_date)

m_conditional_statement ::= M_ConditionalStatement(m_condition, m_statement)

m_condition ::= M_StatementCondition(m_statement)
                | M_TemporalActionCondition(m_test, t_subject, t_verb_status,
↪  t_object, t_date)
                | M_ExpressionCondition(m_test, t_subject, t_verb_status,
↪  t_comparison, t_subject)
                | M_AndCondition([m_condition])
                | M_OrCondition([m_condition])

m_definition ::= M_IsDefinition(t_subject, t_subject)
                 | M_EqualsDefinition(t_object, t_numericalexpr)
                 | M_DefAnd([m_definition])

m_conditional_definition ::= M_ConditionalDefinition(m_condition, m_definition)

|| Helper Functions to reduce code length
|| Conditional Statements
cond_stmt :: m_condition -> m_statement -> m_conditional_statement
cond_stmt cond stmt = M_ConditionalStatement(cond, stmt)

|| Conditional Definitions
cond_def :: m_condition -> m_definition -> m_conditional_definition
cond_def cond def = M_ConditionalDefinition(cond, def)

|| Translate parsed cola contract into combined syntax
m_translate_contract :: maybe t_contract ->  ([m_statement],
↪  [m_conditional_statement], [m_definition], [m_conditional_definition])
m_translate_contract (Nothing)                          = ([], [], [], [])
m_translate_contract (Just (TrueContract))              = ([], [], [], [])
m_translate_contract (Just (Contract comp))             = (convert_comonent comp)
m_translate_contract (Just (Contracts_and comps))       = combined_components
↪  (map convert_comonent comps) ([], [], [], [])

|| Combine of contract into one tuple
combined_components [] (s, cs, d, cd) = (s, cs, d, cd)
```

```
42  combined_components ((s, cs, d, cd):rest) (sa, csa, da, cda) = combined_components
    ↪  rest ((s++sa), (cs++csa), (d++da), (cd++cda))
43
44  || Convert parsed CoLa component into combined syntax
45  convert_comonent :: t_component -> ([m_statement], [m_conditional_statement],
    ↪  [m_definition], [m_conditional_definition])
46  convert_comonent comp
47   = xconvert_component comp
48     where
49     || Pattern matching on the different types of components
50     xconvert_component (CompDefinition t_def)                    = ([], [],
    ↪  [(trans_def (t_def))], [])
51     xconvert_component (CompStatement t_stmt)                    = ((trans_stmt
    ↪  (t_stmt)), [], [], [])
52     xconvert_component (CompCondStatement t_cond t_stmt)         = ([],
    ↪  (trans_cond_stmts t_cond t_stmt) , [], [])
53     xconvert_component (CompStatStatement t_stmt1 t_stmt2   )    = ([],
    ↪  (trans_stmt_stmts t_stmt1 t_stmt2), [], [])
54     xconvert_component (CompExprDefinition t_expr t_def   )      = ([], [], [],
    ↪  [(trans_expr_def t_expr t_def)])
55
56     || Convert CoLa unconditional definitions
57     || trans_def :: t_definition -> m_definition
58     trans_def (Definitions_and comps)         = M_DefAnd((map trans_def comps))
59     trans_def (Def_IS t_ID sbj1 sbj2)         = M_IsDefinition(sbj1, sbj2)
60     trans_def (Def_EQ t_ID obj num_exp)       = M_EqualsDefinition(obj, num_exp)
61
62     || Convert CoLa conditional definitions
63     || trans_expr_def :: t_condition -> t_definition -> m_conditional_definition
64     trans_expr_def condExpr defin = xtrans_expr_def (trans_expr_cond condExpr)
    ↪  (trans_def defin)
65     xtrans_expr_def prsedExpr defin = M_ConditionalDefinition(prsedExpr, defin)
66
67     || Convert CoLa expression conditions
68     || trans_expr_cond :: t_condition -> m_condition
69     trans_expr_cond (Expressions_or exprs)  = M_OrCondition( (map trans_expr_cond
    ↪  exprs))
70     trans_expr_cond (Expressions_and exprs) = M_AndCondition( (map trans_expr_cond
    ↪  exprs))
71     trans_expr_cond (Expression tId Holds sbj1 vrb_sts comp sbj2)      =
    ↪  M_ExpressionCondition(TRUE, sbj1, vrb_sts, comp, sbj2)
72     trans_expr_cond (Expression tId NotHolds sbj1 vrb_sts comp sbj2)   =
    ↪  M_ExpressionCondition(FALSE, sbj1, vrb_sts, comp, sbj2)
73
74     || Convert CoLa conditional statements
75     || trans_stmt_stmts :: t_statement -> t_statement -> [m_conditional_statement]
76     trans_stmt_stmts stmt_cond stmt = xtrans_stmt_stmts (trans_stmt_cond stmt_cond)
    ↪  (trans_stmt stmt)
77     xtrans_stmt_stmts prsed_cond stmtlist = (map (cond_stmt prsed_cond) stmtlist)
```

```
78
79      || Convert CoLa statement conditions
80      || trans_stmt_cond :: t_condition -> m_condition
81      trans_stmt_cond (Statements_and stmts)      = M_AndCondition((map
        ↪  trans_stmt_cond stmts))
82      trans_stmt_cond (Statements_or  stmts)      = M_OrCondition((map trans_stmt_cond
        ↪  stmts))
83      trans_stmt_cond (Statement tId Holds stmtType date sbj vrb obj)     =
        ↪  M_StatementCondition(M_TemporalActionStatement(TRUE, sbj, (deontic_to_verb
        ↪  stmtType), vrb, obj, date))
84      trans_stmt_cond (Statement tId NotHolds stmtType date sbj vrb obj)  =
        ↪  M_StatementCondition(M_TemporalActionStatement(FALSE, sbj,(deontic_to_verb
        ↪  stmtType), vrb, obj, date))
85
86      || CoLa CoLa conditional statements
87      || trans_cond_stmts :: t_condition -> t_statement -> [m_conditional_statement]
88      trans_cond_stmts cond stmt = xtrans_cond_stmts (trans_conds cond) (trans_stmt
        ↪  stmt)
89      xtrans_cond_stmts prsed_cond stmtlist = (map (cond_stmt prsed_cond) stmtlist)
90
91      || Convert cola conditions
92      || trans_conds :: t_condition -> m_condition
93      trans_conds (Conditions_or conds)        = M_OrCondition((map trans_conds
        ↪  conds))
94      trans_conds (Conditions_and conds)       = M_AndCondition((map trans_conds
        ↪  conds))
95      trans_conds (Condition tId Holds date subj vrb_sts obj)    =
        ↪  M_TemporalActionCondition(TRUE, subj, vrb_sts, obj, date)
96      trans_conds (Condition tId NotHolds date subj vrb_sts obj) =
        ↪  M_TemporalActionCondition(FALSE, subj, vrb_sts, obj, date)
97
98      || Convert cola statements
99      || trans_stmt :: t_statement -> [m_statement]
100     trans_stmt (Statements_and stmts)      = (concat (map trans_stmt stmts))
101     trans_stmt (Statements_or  stmts)      = (concat (map trans_stmt stmts))
102     trans_stmt (Statement tId Holds stmtType date sbj vrb obj)     =
        ↪  [M_TemporalActionStatement(TRUE, sbj, (deontic_to_verb stmtType), vrb, obj,
        ↪  date)]
103     trans_stmt (Statement tId NotHolds stmtType date sbj vrb obj)  =
        ↪  [M_TemporalActionStatement(FALSE, sbj, (deontic_to_verb stmtType), vrb, obj,
        ↪  date)]
104
105      || Convert deontic type to modal verb
106      || deontic_to_verb :: t_statement_type -> m_denotic_verbs
107      deontic_to_verb    Obligation  = M_SHALL
108      deontic_to_verb    Permission  = M_MAY
109      deontic_to_verb    Prohibition = M_SHANT
110
111
```

```
112 abstype m_contract
113 with
114    translate_contract :: maybe t_contract -> m_contract
115    showm_contract     :: m_contract -> [char]
116
117 m_contract_type   ::= M_Contract ([m_statement], [m_conditional_statement],
    ↪  [m_definition], [m_conditional_definition])
118 m_contract         == m_contract_type
119
120  || Translate parsed cola contract into combined syntax
121 translate_contract parsed_cola_contract = M_Contract (m_translate_contract
    ↪  parsed_cola_contract)
122  || Show combined syntax contract
123 cola_to_python con = translate_contract (parse con)
124
125 showm_contract (M_Contract (stmts, cond_stmts, defin, cond_defins))
126    = "con = Contract()\n\n"
127        ++ (lay (map (print_stmt_call.print_statement) stmts))
128        ++ (lay (map (print_stmt_call.print_condition_stmt) cond_stmts))
129        ++ (lay (map (print_def_call.print_definition) defin))
130        ++ (lay (map (print_def_call.print_conditionitional_definition)
    ↪  cond_defins))
131        ++ "con.interaciveSimulation()"
132
133 print_stmt_call :: [char] -> [char]
134 print_stmt_call stmt = "con.statement(" ++ stmt ++ ")\n"
135 print_def_call def = "con.definition(" ++ def ++ ")\n"
136 print_temporal_expression :: t_date -> [char]
137 print_temporal_expression date = "TemporalExpression('ON','" ++ (show date) ++ "')"
138
139 print_test TRUE = "True"
140 print_test FALSE = "False"
141
142 print_modal_verb M_SHALL = "SHALL"
143 print_modal_verb M_SHANT = "SHANT"
144 print_modal_verb M_MAY = "MAY"
145
146 print_statement (M_TemporalActionStatement(m_test, t_subject, m_modal_verb, t_verb,
    ↪  t_object, date))
147     = "TemporalStatement('" ++ (show t_subject) ++ "', '" ++ (print_modal_verb
    ↪  m_modal_verb)
148         ++ "', '" ++ (show t_verb) ++ "', '" ++ (show t_object) ++ "', " ++
    ↪  (print_temporal_expression date) ++ ", valid=" ++  (print_test m_test) ++ ")"
149
150 print_condition_stmt (M_ConditionalStatement(cond, stmt))
151    = "ConditionalStatement(condition=" ++ (print_condition cond) ++ ", statement="
    ↪  ++ (print_statement stmt) ++ ")"
152
```

```
153 print_condition (M_StatementCondition(M_TemporalActionStatement(m_test, t_subject,
    ↪ m_modal_verb, t_verb, t_object, date)))
154    = "StatementCondition(statement=" ++ (print_statement
    ↪ (M_TemporalActionStatement(m_test, t_subject, m_modal_verb, t_verb, t_object,
    ↪ date)))
155       ++ ", test=" ++ (print_test m_test) ++ ")"
156
157 print_condition (M_TemporalActionCondition(m_test, t_subject, t_verb_status,
    ↪ t_object, date))
158    = "TemporalActionCondition('" ++ (show t_subject) ++ "', '" ++ (show
    ↪ t_verb_status) ++ "' , '"
159       ++ (show t_object) ++ "', " ++ (print_temporal_expression date) ++ ", test="
    ↪ ++ (print_test m_test) ++")"
160
161 print_condition (M_ExpressionCondition(m_test, sbj1, t_verb_status, t_comparison,
    ↪ sbj2))
162    = "ExpressionCondition(BooleanExpression('" ++ (show sbj1) ++ "', '" ++ (show
    ↪ t_verb_status)
163       ++ "', '" ++ (show t_comparison) ++ "', '" ++ (show sbj2) ++ "'), test=" ++
    ↪ (print_test m_test) ++ ")"
164
165 print_condition (M_AndCondition(conds))
166    = "AndCondition(conditions=[" ++ (concat (map ((++",\n\t\t").print_condition)
    ↪ conds)) ++ "])"
167
168 print_condition (M_OrCondition(conds))
169    = "OrCondition(conditions=[" ++ (concat (map ((++",\n\t\t").print_condition)
    ↪ conds)) ++ "])"
170
171
172 print_definition (M_IsDefinition(sbj1, sbj2))       = "IsDefinition('" ++ (show
    ↪ sbj1) ++ "', '" ++ (show sbj2) ++"')"
173 print_definition (M_EqualsDefinition(obj, numexp))  = "EqualsDefinition('" ++ (show
    ↪ obj) ++ "', '" ++ (print_expression numexp) ++"')"
174 print_definition (M_DefAnd(def))                    = "[" ++ (concat (map
    ↪ ((++",\n\t\t").print_definition) def)) ++ "]"
175
176 print_expression (NumericalExprObject obj)  =   (show obj)
177 print_expression (NumericalExprNum  t_num)  =   (show t_num)
178 print_expression (NumericalExprExpr exp1 op exp2) = (print_expression exp1) ++ " "
    ↪ ++ (show op) ++ " " ++ (print_expression exp2)
179 print_expression (NoNumericalExpr) = ""
180
181 print_conditionitional_definition (M_ConditionalDefinition(cond, defs))
182     = "ConditionalDefinition(condition=" ++ (print_condition cond) ++ ",
    ↪ definitions=" ++ (print_definition defs) ++ ")\n"
183
184 || Simple Test Contracts
185 || Simple Statement
```

```
186 test1 = "[1] It is the case that Dominic shall deliver a Report on the 11 September
    ↪  2023."
187 || Simple Forbidden Statement
188 test2 = "[1] It is not the case that Dominic shall deliver a Report on the 11
    ↪  September 2023."
189 || Simple Conditional Statement
190 test3 = "IF [1] it is the case that Dominic delivered a Report on the 11 September
    ↪  2023 THEN [2] it is not the case that Dominic shall deliver a Report on the 11
    ↪  September 2023."
191 test4 = "IF [1] It is not the case that Dominic delivered a Report on the 11
    ↪  September 2023 THEN [2] it is the case that UCL may deliver a Punishment on the
    ↪  11 September 2023 AND [3] it is the case that Dominic shall deliver a Report on
    ↪  the 12 September 2023."
192 || And Condition
193 test5 = "IF [1] it is the case that PartyA shall pay AmountA on the 11 September
    ↪  2023 AND [2] it is the case that PartyB shall pay AmountB on the 11 September
    ↪  2023 THEN [3] it is not the case that PartyA shall pay AmountA on the 11
    ↪  September 2023."
194 test6 = "IF [1] it is the case that PartyA shall pay AmountA on the 11 September
    ↪  2023 OR [2] it is the case that PartyB shall pay AmountB on the 11 September
    ↪  2023 THEN [3] it is not the case that PartyA shall pay AmountA on the 11
    ↪  September 2023"
195 || Seperate Statements
196 test7 = "[1] It is the case that Dominic shall deliver a Report on the 11 September
    ↪  2023."
197             ++ "<AND> [2] It is not the case that Dominic may deliver a Report on
    ↪  the 11 September 2023."
198
199 || ISDA Contract
200 isda_orig = "IF [1] it is the case that PartyA shall pay AmountA on the 01 January
    ↪  1970 AND [2] it is the case that PartyB shall pay AmountB on the 01 January
    ↪  1970 THEN [3] it is not the case that PartyA shall pay AmountA on the 01
    ↪  January 1970 AND [4] it is not the case that PartyB shall pay AmountB on the 01
    ↪  January 1970 "
201               ++ "<AND> [5] it is the case that ExcessParty shall pay the excess
    ↪  amount of currency on the 01 January 1970 "
202               ++ "<AND> IF [6] it is the case that PartyA paid more than PartyB
    ↪  THEN [7] ExcessParty IS PartyA AND [8] the excess amount of currency EQUALS
    ↪  AmountA MINUS AmountB "
203               ++ "<AND> IF [9] it is the case that PartyB paid more than PartyA
    ↪  THEN [10] ExcessParty IS PartyB AND [11] the excess amount of currency EQUALS
    ↪  AmountB MINUS AmountA."
204
205 || Modified ISDA Contract
```

```
206 isda_modified = "IF [1] it is the case that PartyA shall pay AmountA on the 01
    ↪ January 1970 AND [2] it is the case that PartyB shall pay AmountB on the 01
    ↪ January 1970 THEN [3] it is not the case that PartyA shall pay AmountA on the
    ↪ 01 January 1970 AND [4] it is not the case that PartyB shall pay AmountB on the
    ↪ 01 January 1970 AND [5] it is the case that ExcessParty shall pay the excess
    ↪ amount of currency on the 01 January 1970 "
207                 ++ "<AND> IF [6] it is the case that PartyA paid more than PartyB
    ↪ THEN [7] ExcessParty IS PartyA AND [8] the excess amount of currency EQUALS
    ↪ AmountA MINUS AmountB "
208                 ++ "<AND> IF [9] it is the case that PartyB paid more than PartyA
    ↪ THEN [10] ExcessParty IS PartyB AND [11] the excess amount of currency EQUALS
    ↪ AmountB MINUS AmountA."
209
210 || Bike Delivery Contract
211 bike_orig = "IF [1a] it is the case that Alice paid 100 pounds on the 1 April 2021
    ↪ OR [1b] it is the case that Alice paid 120 pounds on the 1 April 2021 THEN [2]
    ↪ it is the case that Bob must deliver a bicycle on the 5 April 2021"
212             ++ " <AND> [3a] it is the case that Bob may deliver a receipt on the
    ↪ 5 April 2021 AND [3b] it is the case that Bob is forbidden to charge a
    ↪ delivery_fee on the 5 April 2021."
213
214 || Modified Bike Delivery Contract
215  biked_modified = "IF [1a] it is the case that Alice paid 100 pounds on the 1 April
    ↪ 2021 OR [1b] it is the case that Alice paid 120 pounds on the 1 April 2021 THEN
    ↪ [2] it is the case that Bob must deliver a bicycle on the 5 April 2021 AND [3a]
    ↪ it is the case that Bob may deliver a receipt on the 5 April 2021 AND [3b] it
    ↪ is the case that Bob is forbidden to charge a delivery_fee on the 5 April
    ↪ 2021."
216
217 || Bike Delivery Contract with Sanction
218 bike_sanction = "IF [1a] it is the case that Alice paid 100 pounds on the 1 April
    ↪ 2021 OR [1b] it is the case that Alice paid 120 pounds on the 1 April 2021 THEN
    ↪ [2] it is the case that Bob must deliver a bicycle on the 5 April 2021 AND [3a]
    ↪ it is the case that Bob is forbidden to charge a delivery_fee on the 5 April
    ↪ 2021 "
219                  ++ "<AND> IF [6] it is the case that Bob delivered a bicycle on
    ↪ the 5 April 2021 AND [7] it is not the case that Bob charged a delivery_fee on
    ↪ the 5 April 2021 THEN [8] it is the case that Bob may deliver a receipt on the
    ↪ 5 April 2021 "
220                  ++ "<AND> [4] It is the case that Alice may charge 120 pounds on
    ↪ the 1 April 2021 IF [5] it is not the case that Bob delivered a bicycle on the
    ↪ 5 April 2021."
221
222 || Guarantor Agreement Contract
223 guarantor = "[1] It is the case that Landlord shall deliver a Property on the 02
    ↪ April 2021 "
```

```
224             ++ "<AND> IF [2] it is the case that Landlord delivered a
    ↪ demandOfTenantPayment on the 02 March 2021 AND [3] it is not the case that
    ↪ Tenant paid AmountA on the 2 March 2021 THEN [4] it is the case that the
    ↪ Landlord may deliver a demandOfGuarantorPayment on the 03 March 2021 "
225             ++ "<AND> IF [3] it is the case that Landlord delivered a
    ↪ demandOfTenantPayment on the 02 March 2021 AND [4] it is not the case that
    ↪ Tenant paid AmountA on the 2 March 2021 AND [5] it is the case that Landlord
    ↪ delivered a demandOfGuarantorPayment on the 03 March 2021 THEN [6] it is the
    ↪ case that Guarantor shall pay AmountA on the 03 March 2021 "
226             ++ "<AND> IF [7] it is not the case that Tenant paid AmountB on the 10
    ↪ March 2022 THEN [8] it is the case that the Guarantor shall pay AmountB on the
    ↪ 11 March 2023 "
227             ++ "<AND> IF [9] it is the case that HousingBenefitScheme paid AmountC
    ↪ on the 02 March 2021 AND [10] it is the case that LocalAuthority delivered a
    ↪ overpaymentClaim on the 02 March 2022 THEN [11] it is the case that Guarantor
    ↪ shall pay AmountC on the 01 January 1970."
228
229 python_test1 = cola_to_python test1
230 python_test2 = cola_to_python test2
231 python_test3 = cola_to_python test3
232 python_test4 = cola_to_python test4
233 python_test5 = cola_to_python test5
234 python_test6 = cola_to_python test6
235 python_test7 = cola_to_python test7
236 python_isda_orig = cola_to_python isda_orig
237 python_isda_modified = cola_to_python isda_modified
238 python_bike_orig = cola_to_python bike_orig
239 python_bike_modified = cola_to_python biked_modified
240 python_bike_sanction = cola_to_python bike_sanction
241 python_guarantor = cola_to_python guarantor
```

Listing 13: Miranda Code for CoLa contract translation.

A.1.3 Lexon Compiler

```rust
// ./src/main.rs
fn main() {
        let args: Vec<String> = std::env::args().collect();
        let unparsed_file = std::fs::read_to_string(&args[1]).expect("cannot read file");
        match ast::parse(unparsed_file){
                Ok(lexons)=>{
                        let vm=vm::LexonVM::new(lexons);
                        println!("\nResulting Ethereum Solidity code:\n");
                        let sol=vm.solidity();
                        println!("{}",sol);
                        check_sol(sol);
                        println!("\nResulting Aeternity Sophia code:\n");
                        let sop=vm.sophia();
                        println!("{}",sop);
                        check_sop(sop);
            // ====================================================
                        // Modification by Dominic Kloecker
                        // Write the JSON AST to a file
                        let json = vm.get_json();
                        println!("{}", json);
                        let directory = "./json_ast";
                        let filename = format!("{}/{}.txt", directory, "contract_ast");
                        // Create the directory if it does not exist
                        if !Path::new(directory).exists() {
                                fs::create_dir_all(directory).expect("Failed to create directory");
                        }
                        // Write the fromatted_content to the file
                        let fromatted_content = format!("{}{}", json, "\n");
                        fs::write(filename, &fromatted_content).expect("Unable to write to file");
            // ====================================================
                },
                Err(error)=>{
                        println!("error: {:?}",error);
                }
        };
}
```

Listing 14: Rust code of modified Lexon compiler [4], outputting Lexons AST in JSON format using inbuilt Lexon get_json() method.

A.1.4 Lexon Syntax Translator

```python
import json
from pprint import pprint
from typing import List, Dict, Tuple, Optional

from Condition import Condition, StateCondition, AndCondition, ExpressionCondition
from Contract import Contract
from Definition import IsDefinition, EqualsDefinition, ConditionalDefinition
from Expression import NumericExpression
from State import State
from Statement import Statement, TwoSubjects, ConditionalStatement


def make_groups(lexons: List[Dict]):
    """Splot lexon list into groups to keep track of subject.
    Statmenets split by AND are done by the same subject while seqeunce indicates
    a new subject."""
    groups = []
    group = []
    for lexon in lexons:
        match lexon["stmt"]:
            case "Sequence":
                groups.append(group)
                group = []
            case {"And": _}:
                pass
            case _:
                group.append(lexon)

    groups.append(group)
    return groups

def convert_conditions(condition: Dict) -> Tuple[Condition, Condition]:
    """convert the conditions and return the true and false conditions."""
    condition_operators = condition["ops"]
    expressions = condition["exprs"]

    current_expression = expressions.pop(0)
    if "Cmp" in current_expression:
        result, test = convert_comparison(current_expression["Cmp"]), True
    else:
        result, test = convert_is_expression(current_expression["Is"])

    true_condition = ExpressionCondition(result, test=test)
    false_condition = ExpressionCondition(result, test=test)

    if not condition_operators:
```

```
47          return true_condition, false_condition
48
49      if condition_operators[0] == "And":
50          true_conditions = [true_condition]
51          false_conditions = [false_condition]
52
53          while expressions:
54              expr = expressions.pop(0)
55
56              if "Cmp" in expr:
57                  result, test = convert_comparison(expr["Cmp"]), True
58              else:
59                  result, test = convert_is_expression(expr["Is"])
60
61              true_conditions.append(ExpressionCondition(result, test=test))
62              false_conditions.append(ExpressionCondition(result, test=test))
63
64          return AndCondition(true_conditions), AndCondition(false_conditions)  #
            ↪ type: ignore
65
66      return true_condition, false_condition
67
68
69  def convert_is_expression(is_expr: Dict) -> tuple[NumericExpression, bool]:
70      """convert 'is' expression and return a numeric expression."""
71      first_operand = is_expr[0]
72      operator = is_expr[1]
73      test = True
74      if operator == "isnot":
75          test = False
76          operator = "is"
77
78      second_operand = convert_expression(is_expr[2])
79
80      return NumericExpression(first_operand, operator, second_operand), test
81
82
83  def convert_comparison(comparison: Dict) -> NumericExpression:
84      """convert comparison and return a numeric expression."""
85      operator = comparison["op"]
86      first_operand = convert_expression(comparison["exp1"])
87      second_operand = convert_expression(comparison["exp2"])
88
89      return NumericExpression(first_operand, operator, second_operand)
90
91
92  def convert_expression(expr: Dict) -> str:
93      """convert general expression and return its string representation."""
94      operators = expr["ops"]
```

```
95      result = convert_terms(expr["terms"].pop(0))
96
97      for op in operators:
98          result = f"{result} {op} {convert_terms(expr['terms'].pop(0))}"
99
100     return result
101
102
103 def convert_terms(term: Dict) -> str:
104     """convert terms and return its string representation."""
105     factors = term["factors"]
106     return convert_factors(factors.pop(0))
107
108
109 def convert_factors(factors: Dict) -> str:
110     """convert factors and return its string representation."""
111     if not factors:
112         return ""
113
114     if "Sym" in factors:
115         return factors["Sym"]["sym"]
116     if "sym" in factors:
117         return factors["sym"]
118     if "Num" in factors:
119         return ""
120
121     # For other types of factors
122     key, value = list(factors.items())[0]
123     if key == "Time":
124         return f"{value}"
125     if key == "Remainder":
126         return f"{value} {key}"
127
128     return f"{value} {key}"
129
130
131 def convert_payment_statement(pay_stmt: Dict) -> Tuple[str, str, str]:
132     """convert payment statement and return pay-to, pay-from, and amount
        ↪  details."""
133     pay_from = convert_factors(pay_stmt["from"]) if pay_stmt["from"] else ""
134     amount_expression = pay_stmt["exp"]
135     pay_to = convert_factors(pay_stmt["to"]) if pay_stmt["to"] else "the other
        ↪  party"
136
137     amount = convert_expression(amount_expression)
138
139     return pay_to, pay_from, amount
140
141
```

```
142 def convert_return_statement(ret_stmt: Dict) -> Tuple[str, str]:
143     """convert payment statement and return pay-to, pay-from, and amount
        ↪  details."""
144     terms = ret_stmt[0][0]["terms"]
145     return_what = convert_factors(terms[0]["factors"].pop(0))
146 
147     return_to = ret_stmt[1]["sym"] if ret_stmt[1] else "the other party"
148 
149     return return_what, return_to
150 
151 
152 def convert_operation(operations: Dict) -> str:
153     """convert operations and return its string representation."""
154     terms = operations[1]["terms"]
155     factor_operators = operations[1]["ops"]
156     factors = terms[0]["factors"]
157 
158     result = convert_factors(factors.pop(0))
159     object = operations[0]
160 
161     while factor_operators:
162         op = factor_operators.pop(0)
163         result = f"{result} {op} {convert_factors(factors.pop(0))}"
164 
165     return f"{result}"
166 
167 
168 def convert_lexon_group(lexons: List[Dict], con: Contract,
169                         pre_condition: Optional[Condition] = None, prev_subject="",
170                         modal_verb="Shall", overall_condition: Optional[Condition] =
                            ↪  None,
171                         fullfillment_conditions: Optional[Condition] = None) -> None:
172     prev_subject = prev_subject
173     pre_condition = pre_condition
174 
175     if fullfillment_conditions == None:
176         fullfillment_conditions = []
177 
178     while lexons:
179         # Pop the next lexon from the list and unpack it
180         lexon = lexons.pop(0)
181         stmt = lexon["stmt"]
182         varname = lexon["varnames"]
183 
184         # Reset the statement and definition
185         statement = None
186         definition = None
187 
188         # Pattern match the statement and convert it to appropriate element
```

```
189             match stmt:
190                 case "Sequence":
191                     continue
192 
193                 case "And":
194                     continue
195 
196                 case {"Definition": _}:
197                     continue
198 
199                 case {"Increase": _}:
200                     if prev_subject != "":
201                         subject = prev_subject
202                     else:
203                         subject = varname[0]
204 
205                     operation = convert_operation(stmt["Increase"])
206                     subject2 = stmt["Increase"][0]
207                     statement = TwoSubjects(subject, modal_verb, "Increase", subject2,
                        ↪  "by", operation)
208 
209                 case {"Decrease": _}:
210                     if prev_subject != "":
211                         subject = prev_subject
212                     else:
213                         subject = varname[0]
214 
215                     subject2 = stmt["Decrease"][0]
216                     operation = convert_operation(stmt["Decrease"])
217                     statement = TwoSubjects(subject, modal_verb, "Decrease", subject2,
                        ↪  "by", operation)
218 
219                 case {"Pay": _}:
220                     if prev_subject != "":
221                         subject = prev_subject
222                     else:
223                         subject = stmt["Pay"]["who"]["sym"]
224                         prev_subject = subject
225 
226                     pay_to, pay_from, what = convert_payment_statement(stmt["Pay"])
227                     if pay_from != "" or None:
228                         statement = TwoSubjects(subject, modal_verb, "Pay", what,
                            ↪  "from" + pay_from + " to", pay_to)
229                     else:
230                         statement = TwoSubjects(subject, modal_verb, "Pay", what, "to",
                            ↪  pay_to)
231 
232                 case {"Return": _}:
233                     if prev_subject != "":
```

```
234                     subject = prev_subject
235                 else:
236                     subject = varname[0]
237                     prev_subject = subject
238 
239                 return_what, return_to = convert_return_statement(stmt["Return"])
240                 statement = TwoSubjects(subject, modal_verb, "Return", return_what,
                    ↪  "to", return_to)
241 
242             case {"Certify": _}:
243                 if prev_subject != "":
244                     subject = prev_subject
245                 else:
246                     subject = varname[0]
247                     prev_subject = subject
248 
249                 # Combine all the strings in the list (Might be nested lists so
                    ↪  flatten it first)
250                 # and then join them with a space
251                 object = " ".join([item for sublist in stmt["Certify"] for item in
                    ↪  sublist])
252                 # object = stmt["Certify"]
253                 statement = Statement(subject, modal_verb, "certify", object)
254 
255             case {"Fix": _}:
256                 if prev_subject != "":
257                     subject1 = prev_subject
258                     object = varname[-1]
259                 else:
260                     subject1 = varname[0]
261                     object = varname[-1]
262                     prev_subject = subject1
263 
264                 statement = Statement(subject1, modal_verb, "Fix", object)
265 
266             case "Appoint":
267                 if prev_subject != "":
268                     subject1 = prev_subject
269                     object = varname[0]
270                 else:
271                     subject1 = varname[0]
272                     object = varname[1]
273                     prev_subject = subject1
274 
275                 statement = Statement(subject1, modal_verb, "Appoint", object)
276 
277             case {"Be": _}:
278                 subject = varname[0]
279 
```

```
280                 if stmt["Be"]["expression"]:
281                     expressions = stmt["Be"]["expression"]
282                     expression_string = convert_expression(expressions)
283 
284                     object = expression_string
285                     definition = EqualsDefinition(subject, object)
286                 else:
287                     expressions = stmt["Be"]["def"]
288                     definition = IsDefinition(subject, expressions)
289 
290             case {"If": _}:
291                 conditions = stmt["If"]
292                 condition_expressions = conditions["cond"]
293 
294                 on_true_cond, on_false_cond =
                    ↪ convert_conditions(condition_expressions)
295 
296                 # Recursive call to convert nested statements
297                 if pre_condition is not None:
298                     on_true_comb = AndCondition.combined([on_true_cond,
                        ↪ pre_condition])
299                     on_false_comb = AndCondition.combined([on_false_cond,
                        ↪ pre_condition])
300                 else:
301                     on_true_comb = on_true_cond
302                     on_false_comb = on_false_cond
303 
304                 on_true = stmt["If"]["ontrue"]
305                 convert_lexon_group(lexons=on_true, con=con,
                    ↪ pre_condition=on_true_comb, modal_verb=modal_verb,
306                                     prev_subject=prev_subject,
                                        ↪ overall_condition=overall_condition)
307                 on_false = stmt["If"]["onfalse"]
308                 # Recursive call to convert nested statements
309                 convert_lexon_group(lexons=on_false, con=con,
                    ↪ pre_condition=on_false_comb, modal_verb=modal_verb,
310                                     prev_subject=prev_subject,
                                        ↪ overall_condition=overall_condition)
311 
312             case {"May": _}:
313                 subject = stmt["May"][0]
314                 nested_statemtents = stmt["May"][1]
315                 # Recursive call to convert nested statements
316                 convert_lexon_group(nested_statemtents, con,
                    ↪ pre_condition=pre_condition, prev_subject=subject,
317                                     modal_verb="May",
                                        ↪ overall_condition=overall_condition)
318 
319             # General Catch all
```

```
320             case _:
321                 pass
322
323         condition = None
324         if pre_condition and overall_condition:
325             condition = AndCondition.combined([pre_condition, overall_condition])
326         elif overall_condition:
327             condition = overall_condition
328
329         if statement:
330             if condition:
331                 statement = ConditionalStatement(condition, statement)
332                 pre_condition = statement.statement().fulfillmentCondition()
333                 fullfillment_conditions.append(pre_condition)
334             else:
335                 pre_condition = statement.fulfillmentCondition()
336                 fullfillment_conditions.append(pre_condition)
337
338             con.statement(statement)
339
340         elif definition:
341             if condition:
342                 definition = ConditionalDefinition(condition, [definition])
343
344             con.definition(definition)
345
346         modal_verb = "Shall"
347
348
349 def convert_all_lexon_groups(lexon_groups: List[List[Dict]], contract: Contract,
350                              prev_condition: Optional[Condition] = None,
                             ↪  overall_condition: Optional[Condition] = None,
351                              fullfillment_conditions: List[Condition] = []):
352     """convert all the lexon groups."""
353     for lexon_group in lexon_groups:
354         convert_lexon_group(lexon_group, contract, prev_condition,
         ↪  overall_condition=overall_condition,
355                             fullfillment_conditions=fullfillment_conditions)
356
357
358 class LexonTranslator:
359     def __init__(self, filename) -> None:
360         self._filename = filename
361         self._json = {}
362         # self._contract = Con
363
364     def read_json_from_file(self) -> None:
365         with open(self._filename, "r") as f:
366             self._json = json.load(f)[0]
```

```
367
368     def pprint_json(self):
369         """Pretty print the JSON"""
370         pprint(self._json)
371
372     def _terms_stmts(self):
373         """Return the Statement JSONS"""
374         return self._json["term_stmts"].copy()
375
376     def _term_clasuses(self):
377         """Return the Clauses JSONS"""
378         return self._json["term_chpts"].copy()
379
380     def _get_recitals(self) -> list[dict]:
381         """Return all recitals"""
382         return self._terms_stmts()
383
384     def _get_clauses(self) -> list[dict]:
385         """Return all clauses"""
386         return self._term_clasuses()
387
388     def _convert(self) -> None:
389         """Convert the Contract JSON into equivalent Petri Net"""
390         # # Get Recitals
391         recitals = self._get_recitals()
392
393         state = State(label="Contract Active", condition=None)
394         state_cond = StateCondition(state=state)
395
396         groups = make_groups(recitals)
397         con = Contract()
398         con.state(state)
399         recital_fullfillment = []
400         test = convert_all_lexon_groups(groups, con, overall_condition=state_cond,
401                                         fullfillment_conditions=recital_fullfillment)
402         # recitals_met = AndCondition.combined(recital_fullfillment)
403         if recital_fullfillment:
404             recitals_met = AndCondition(recital_fullfillment)
405             recitals_state = State(label="Recitals Met", condition=recitals_met)
406             con.state(recitals_state)
407         else:
408             recitals_state = State(label="Recitals Met", condition=state_cond)
409             con.state(recitals_state)
410
411         recital_cond = StateCondition(state=recitals_state)
412
413         clauses = self._get_clauses()
414         for clause in clauses:
415             name = clause["name"]
```

```
416             clause_state = State(label="Clause Invoked: " + name, condition=None)
417             clause_state_cond = StateCondition(state=clause_state)
418             clause_stmts = clause["statements"]
419             clause_groups = make_groups(clause_stmts)
420             combined_condition = AndCondition.combined([recital_cond,
                ↪  clause_state_cond])
421             convert_all_lexon_groups(clause_groups, con,
                ↪  overall_condition=combined_condition)
422             con.state(clause_state)
423
424         con.interaciveSimulation()
425
426     def _simulate_contract(self):
427         """Simulate Contract"""
428         self.read_json_from_file()
429         self._convert()
```

Listing 15: Python implementation of Lexon contract translation.

A.1.5 Contract Model

Due to space limitations, the Python code of the contract model is not included in the report. The code, along with all other code used for this project, is available in the following GitHub repository GitHub.

A.2 Functionality Tests

This section contains contract conversion and simulation test results for a variety of contracts written in Lexon and CoLa. Performance simulations contain snapshots of the contractual state after a set of events. Due to space limitations, only snapshots of some simulations have been included.
Although every effort has been made to make Petri Nets as visible as possible, it is suggested to read the simulations on a digital device which allows for zooming in.

A.2.1 CoLa Contracts

A.2.1.1 ISDA Master Agreement

Original ISDA master agreement from Listing 3.

Conditions

ID	Status	Condition
C1	UNKNOWN	"ExcessParty" paid SomeCurrency "ExcessAmount" ON (1,1,1970)
C2	UNKNOWN	"PartyA" SHALL pay SomeCurrency "AmountA" ON (1,1,1970)
C3	UNKNOWN	"PartyA" paid morethan "PartyB"
C4	UNKNOWN	"PartyB" SHALL pay SomeCurrency "AmountB" ON (1,1,1970)
C5	UNKNOWN	"PartyB" paid morethan "PartyA"

Statements

ID	Status	Statement	Statement Type & Conditions
S1	TRUE	"ExcessParty" SHALL pay SomeCurrency "ExcessAmount" ON (1,1,1970)	Enabling <- Uncoditional
S2	UNKNOWN	"PartyA" SHALL pay SomeCurrency "AmountA" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)
S3	UNKNOWN	"PartyB" SHALL pay SomeCurrency "AmountB" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)

Definitions

ID	Status	Definition	Conditions
D1	UNKNOWN	("ExcessParty" IS "PartyB") AND (SomeCurrency "ExcessAmount" EQUALS OtherObject "AmountB" MINUS OtherObject "AmountA")	(C5, True)
D2	UNKNOWN	("ExcessParty" IS "PartyA") AND (SomeCurrency "ExcessAmount" EQUALS OtherObject "AmountA" MINUS OtherObject "AmountB")	(C3, True)

Figure A.1: Screenshot of tabular summary: ISDA master agreement (Listing 3)

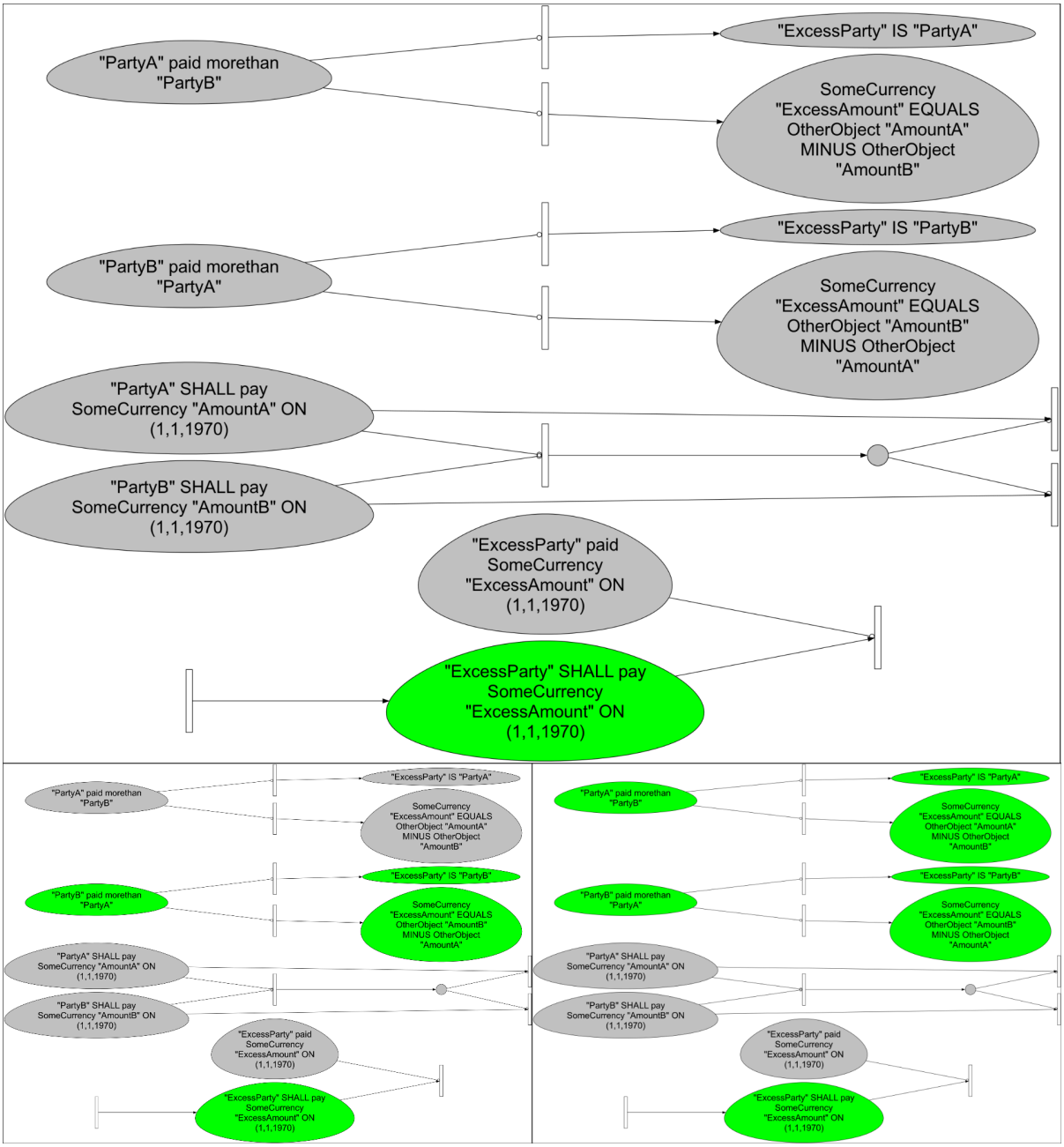


Figure A.2: Performance Simulation snapshots of ISDA master agreement, given illogical scenario stating that PartyB paid more than partyA and PartyA paid more than PartyB ([(C5, True), (C3, True)])

A.2.1.2 Modified ISDA Master Agreement

Conditions		
ID	Status	Condition
C1	UNKNOWN	"ExcessParty" paid SomeCurrency "ExcessAmount" ON (1,1,1970)
C2	UNKNOWN	"PartyA" SHALL pay SomeCurrency "AmountA" ON (1,1,1970)
C3	UNKNOWN	"PartyA" paid morethan "PartyB"
C4	UNKNOWN	"PartyB" SHALL pay SomeCurrency "AmountB" ON (1,1,1970)
C5	UNKNOWN	"PartyB" paid morethan "PartyA"

Statements			
ID	Status	Statement	Statement Type & Conditions
S1	TRUE	"ExcessParty" SHALL pay SomeCurrency "ExcessAmount" ON (1,1,1970)	Enabling <- Uncoditional
S2	UNKNOWN	"PartyA" SHALL pay SomeCurrency "AmountA" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)
S3	UNKNOWN	"PartyB" SHALL pay SomeCurrency "AmountB" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)

Statements			
ID	Status	Statement	Statement Type & Conditions
S1	UNKNOWN	"PartyA" SHALL pay SomeCurrency "AmountA" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)
S2	UNKNOWN	"PartyB" SHALL pay SomeCurrency "AmountB" ON (1,1,1970)	Voiding <- (C2, True) & (C4, True)
S3	UNKNOWN	"ExcessParty" SHALL pay SomeCurrency "ExcessAmount" ON (1,1,1970)	Enabling <- (C2, True) & (C4, True)

Figure A.3: Screenshot of tabular summary: Modified ISDA master agreement (Lisiting 8).

Figure A.4: Performance Simulation snapshots of modified ISDA Master Agreement showing voidance of obligations. Scenario [(C4, True), (C2, True)]

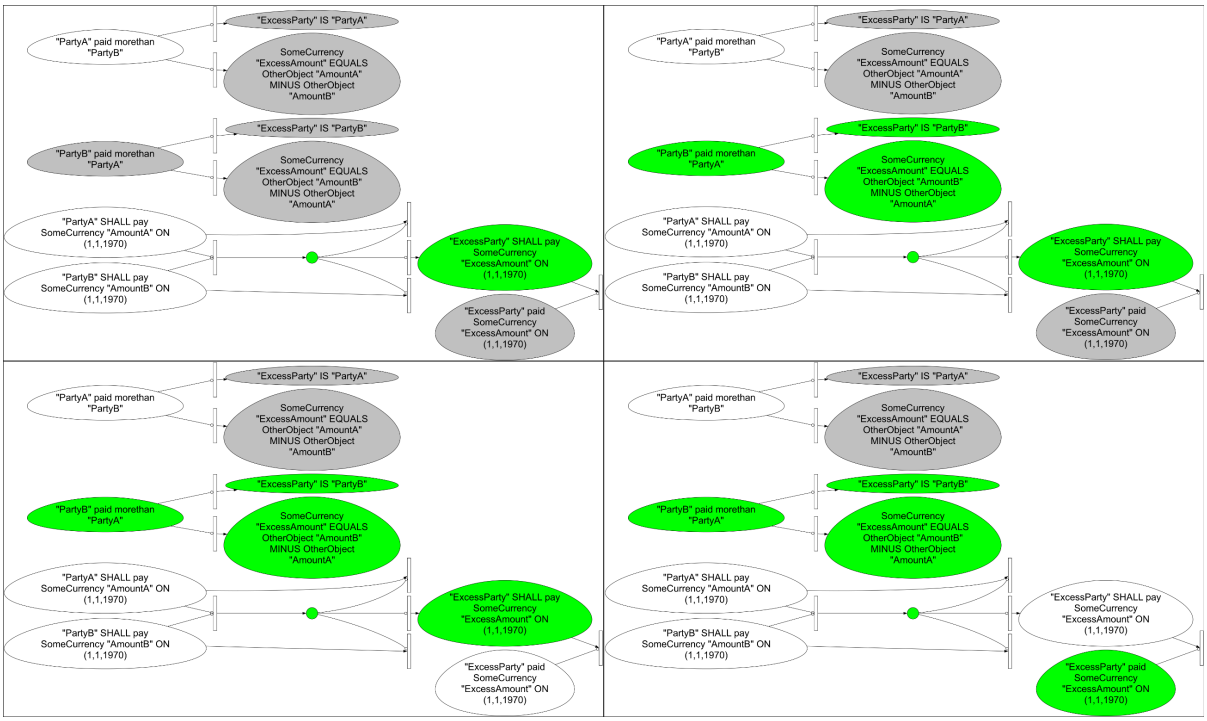


Figure A.5: Continuation of modified ISDA master agreement simulation in Figure A.4, showing completed performance of the contract. Scenario [(C4, True), (C2, True), (C3, False), (C5, True), (C1, False), (C1, True)]

A.2.1.3 Bike Delivery

Simple delivery agreement from Listing 9

Conditions

ID	Status	Condition
C1	UNKNOWN	Alice paid Pounds 100 ON (1,4,2021)
C2	UNKNOWN	Alice paid Pounds 120 ON (1,4,2021)
C3	UNKNOWN	Bob charged OtherObject delivery_fee ON (5,4,2021)
C4	UNKNOWN	Bob delivered OtherObject bicycle ON (5,4,2021)
C5	UNKNOWN	Bob delivered OtherObject receipt ON (5,4,2021)

Statements

ID	Status	Statement	Statement Type & Conditions
S1	TRUE	Bob MAY deliver OtherObject receipt ON (5,4,2021)	Enabling <- Uncoditional
S2	TRUE	Bob SHANT charge OtherObject delivery_fee ON (5,4,2021)	Enabling <- Uncoditional
S3	UNKNOWN	Bob SHALL deliver OtherObject bicycle ON (5,4,2021)	Enabling <- (C1, True) | (C2, True)

Figure A.6: Screenshot of tabular summary: Bike delivery agreement (Listing 9)

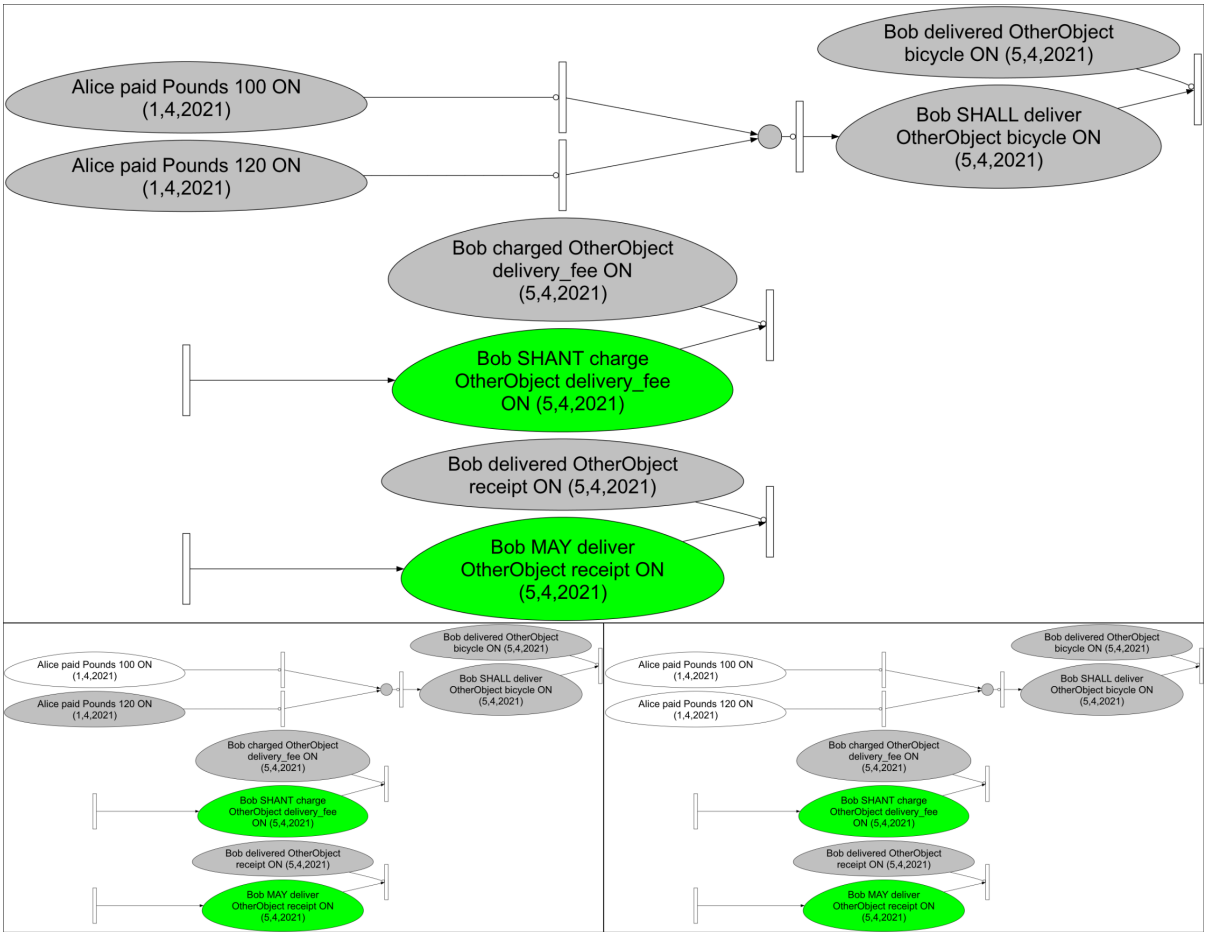


Figure A.7: Performance Simulation of bike delivery agreement, with no payment occurring. Scenario [(C1, False), (C2, False)]

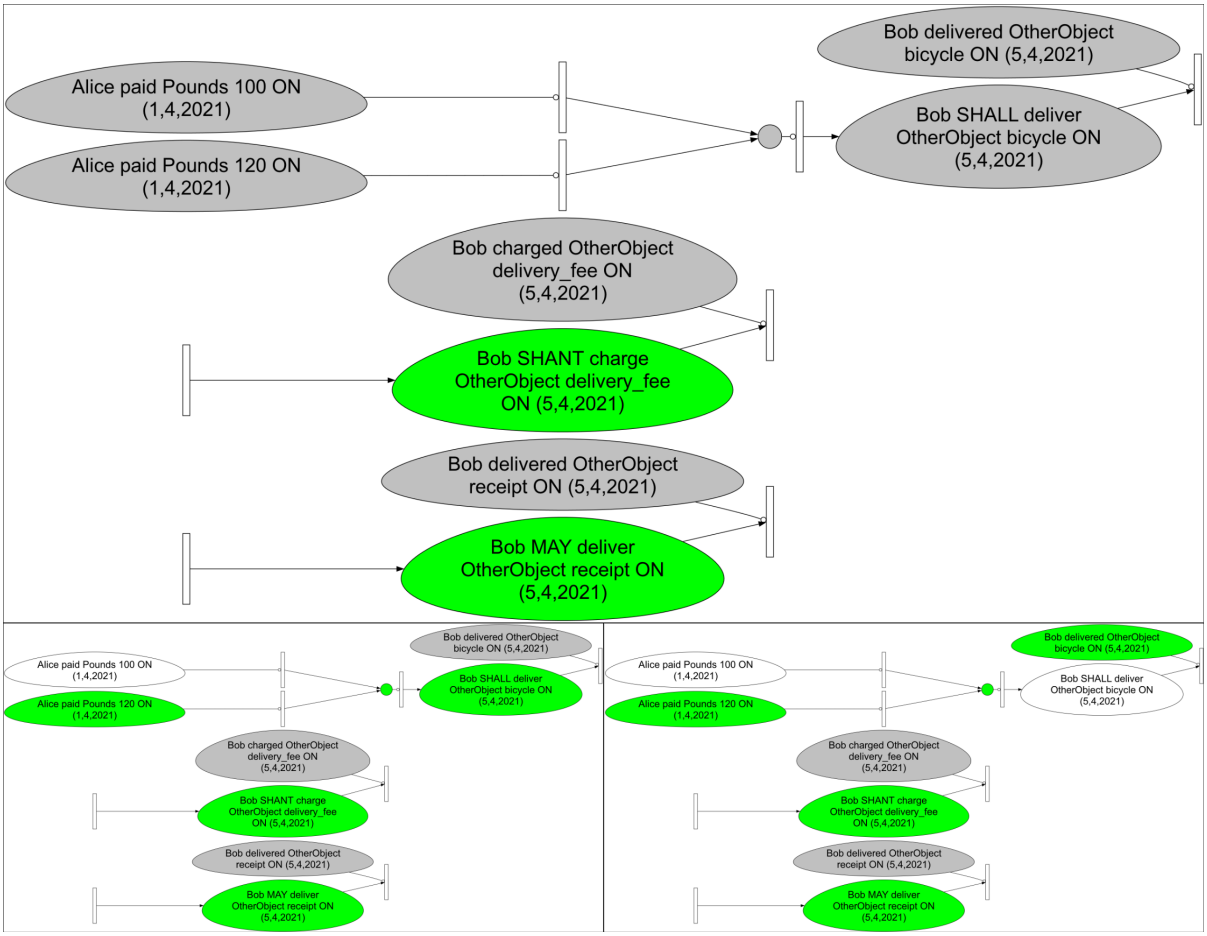
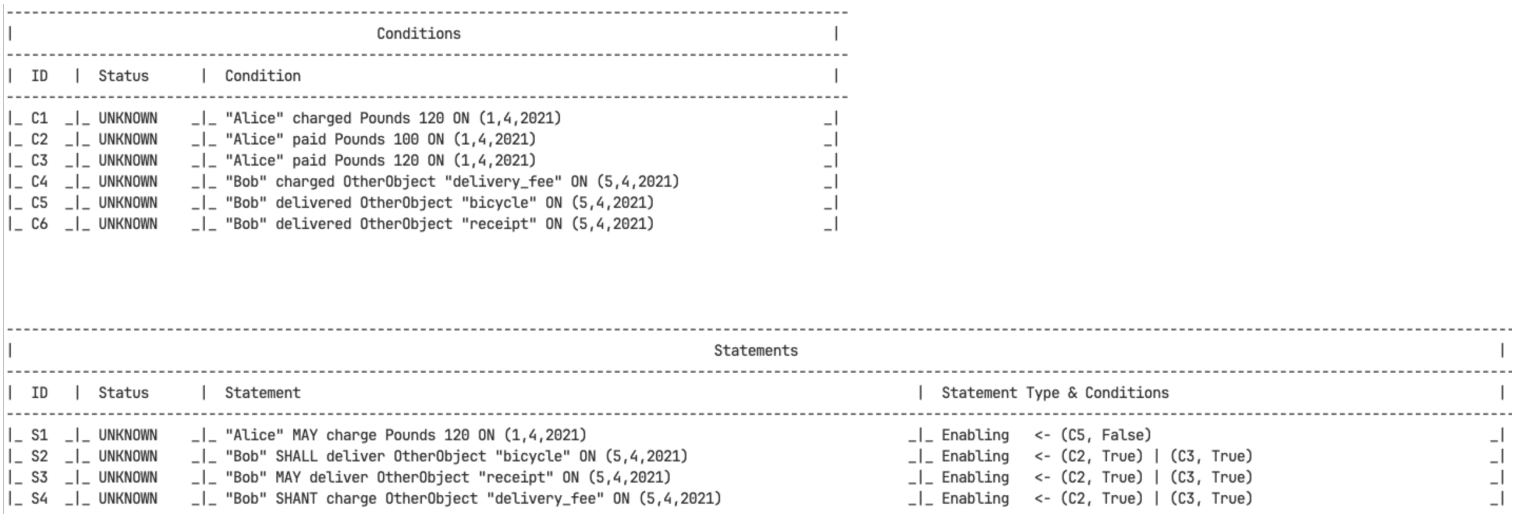


Figure A.8: Performance Simulation of bike delivery agreement, payment occurring and bike delivered. Scenario [(C1, False), (C2, True), (C4, True)]

A.2.1.4 Bike Delivery with Sanction

```
IF  [1a] it is the case that Alice paid 100 pounds on the 1 April 2021
        OR
    [1b] it is the case that Alice paid 120 pounds on the 1 April 2021
THEN[2]  it is the case that Bob must deliver a bicycle on the 5 April 2021
        AND
    [3a] it is the case that Bob may deliver a receipt on the 5 April 2021
        AND
    [3b] it is the case that Bob is forbidden to charge a delivery_fee on the 5
    ↪  April 2021 "
C-AND
    [4] It is the case that Alice may charge 120 pounds on the 1 April 2021
IF  [5] it is not the case that Bob delivered a bicycle on the 5 April 2021.
```

Listing 16: CoLa contract of bike delivery with sanction.



Conditions		
ID	Status	Condition
C1	UNKNOWN	"Alice" charged Pounds 120 ON (1,4,2021)
C2	UNKNOWN	"Alice" paid Pounds 100 ON (1,4,2021)
C3	UNKNOWN	"Alice" paid Pounds 120 ON (1,4,2021)
C4	UNKNOWN	"Bob" charged OtherObject "delivery_fee" ON (5,4,2021)
C5	UNKNOWN	"Bob" delivered OtherObject "bicycle" ON (5,4,2021)
C6	UNKNOWN	"Bob" delivered OtherObject "receipt" ON (5,4,2021)

Statements			
ID	Status	Statement	Statement Type & Conditions
S1	UNKNOWN	"Alice" MAY charge Pounds 120 ON (1,4,2021)	Enabling <- (C5, False)
S2	UNKNOWN	"Bob" SHALL deliver OtherObject "bicycle" ON (5,4,2021)	Enabling <- (C2, True) | (C3, True)
S3	UNKNOWN	"Bob" MAY deliver OtherObject "receipt" ON (5,4,2021)	Enabling <- (C2, True) | (C3, True)
S4	UNKNOWN	"Bob" SHANT charge OtherObject "delivery_fee" ON (5,4,2021)	Enabling <- (C2, True) | (C3, True)

Figure A.9: Screenshot of tabular summary: Bike delivery with sanction

Figure A.10: Performance simulation of bike delivery with sanctions, no payment occurring. Scenario [(C2, False), (C3, False)]

Figure A.11: Performance simulation of bike delivery with sanctions, bike not delivered after payment is made. Scenario [(C2, False), (C3, True), (C5, False), (C1, True)]

Figure A.12: Performance simulation of bike delivery with sanctions, bike delivered. Scenario [(C2, True), (C5, True), (C6, True)]

A.2.2 Lexon Contracts

A.2.2.1 Simple Escrow Agreement

```
LEX Paid Escrow.
LEXON: 0.2.20
COMMENT: 3.f - an escrow that is controlled by a third party for a fee.

"Payer" is a person.
"Payee" is a person.
"Arbiter" is a person.
"Fee" is an amount.

The Payer pays an Amount into escrow ,
appoints the Payee ,
appoints the Arbiter ,
appoints the Broker,
and also fixes the Fee .

CLAUSE: Pay Out.
The Payer may pay the escrow to the Payee.
The Arbiter may pay from escrow the Fee to themselves,
and afterwards pay the remainder of the escrow to the Payee.

CLAUSE: Pay Back.
The Payee may pay the escrow to the Payer.
The Arbiter may pay from escrow the Fee to themselves,
and afterwards return the remainder of the escrow to the Payer.

```

Listing 17: Lexon contract, Escrow controlled by a third party [5]

Conditions	
ID	Status	Condition
C1	UNKNOWN	Arbiter paid Escrow Remainder to Payee
C2	TRUE	Arbiter paid Fee fromescrow to Themselves
C3	TRUE	Arbiter returned Escrow Remainder to Payer
C4	TRUE	Clause Invoked: Pay_Back
C5	UNKNOWN	Clause Invoked: Pay_Out
C6	TRUE	Contract Active
C7	UNKNOWN	Payee paid Escrow to Payer
C8	TRUE	Payer appointed Arbiter
C9	TRUE	Payer appointed Payee
C10	TRUE	Payer fixed Fee
C11	TRUE	Payer paid Amount to Escrow
C12	UNKNOWN	Payer paid Escrow to Payee
C13	UNKNOWN	Recitals Met

Statements			
ID	Status	Statement	Statement Type & Conditions
S1	FALSE	Payer SHALL pay Amount to Escrow	Enabling <- (C6, True)
S2	FALSE	Payer SHALL appoint Payee	Enabling <- (C11, True) & (C6, True)
S3	FALSE	Payer SHALL appoint Arbiter	Enabling <- (C9, True) & (C6, True)
S4	FALSE	Payer SHALL fix Fee	Enabling <- (C8, True) & (C6, True)
S5	UNKNOWN	Payer MAY pay Escrow to Payee	Enabling <- (C13, True) & (C5, True)
S6	FALSE	Arbiter MAY pay Fee fromescrow to Themselves	Enabling <- (C13, True) & (C5, True)
S7	UNKNOWN	Arbiter SHALL pay Escrow Remainder to Payee	Enabling <- (C2, True) & (C13, True) & (C5, True)
S8	TRUE	Payee MAY pay Escrow to Payer	Enabling <- (C13, True) & (C4, True)
S9	FALSE	Arbiter MAY pay Fee fromescrow to Themselves	Enabling <- (C13, True) & (C4, True)
S10	FALSE	Arbiter SHALL return Escrow Remainder to Payer	Enabling <- (C2, True) & (C13, True) & (C4, True)

Figure A.13: Screenshot of tabular summary: Simple escrow agreement (Listing 17)

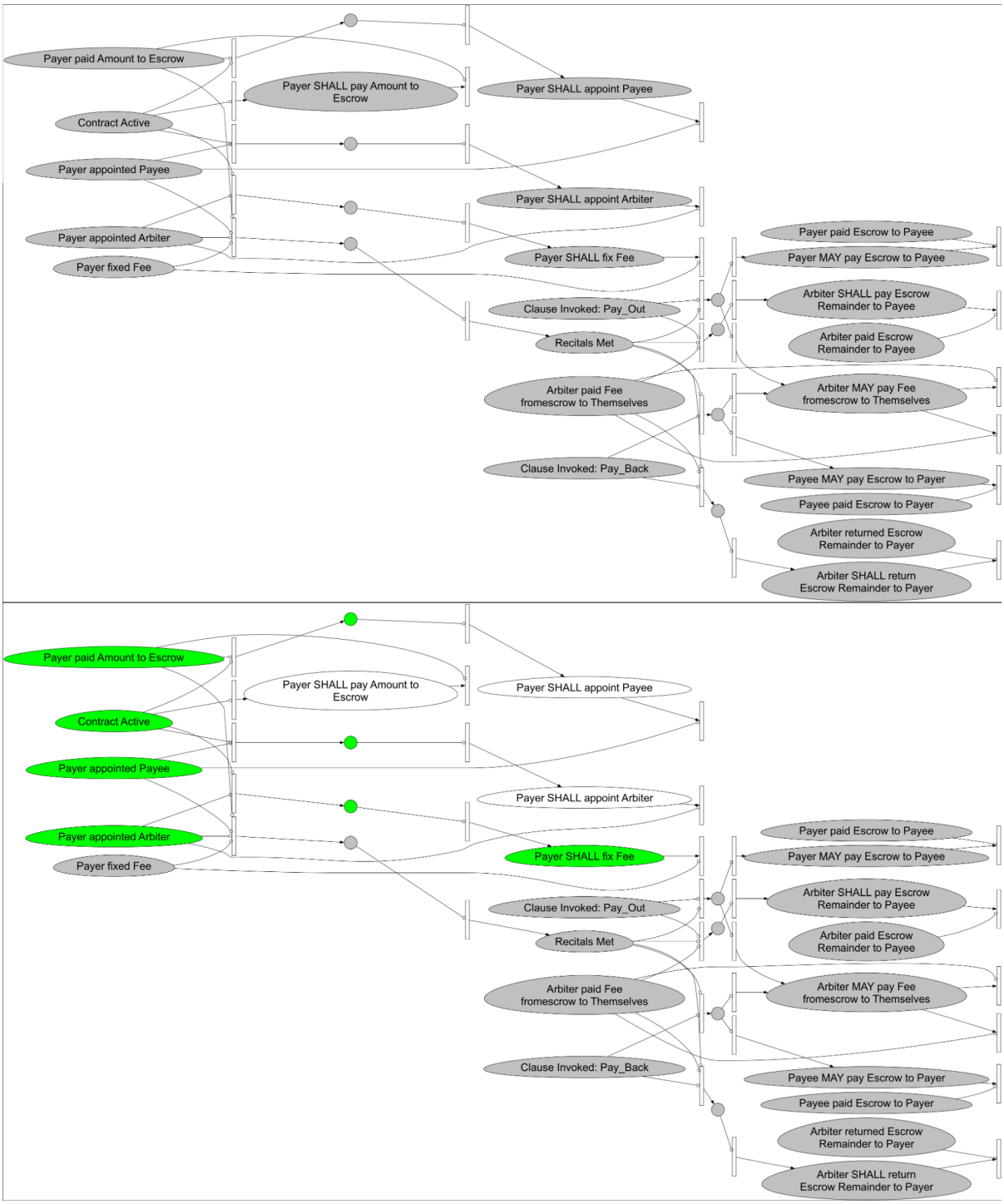


Figure A.14: Performance simulation of simple escrow agreement. Scenario [(C6, True), (C11, True), (C9, True), (C8, True)]

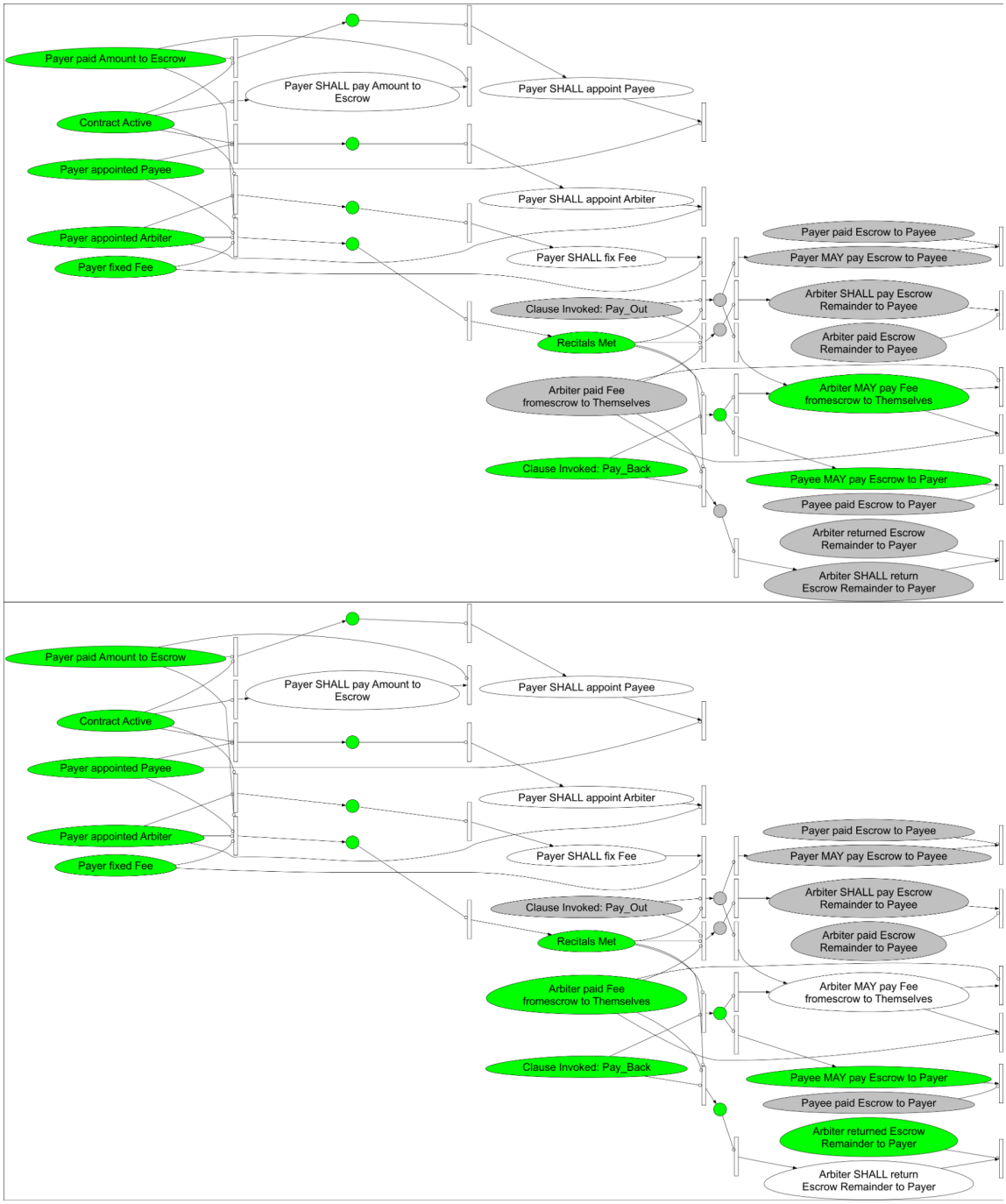


Figure A.15: Performance simulation of simple escrow agreement, Recitals met and Pay Back clause invoked. Scenario [(C6, True), (C11, True), (C9, True), (C8, True), (C10, True), (C4, True), (C2, True), (C3, True)]

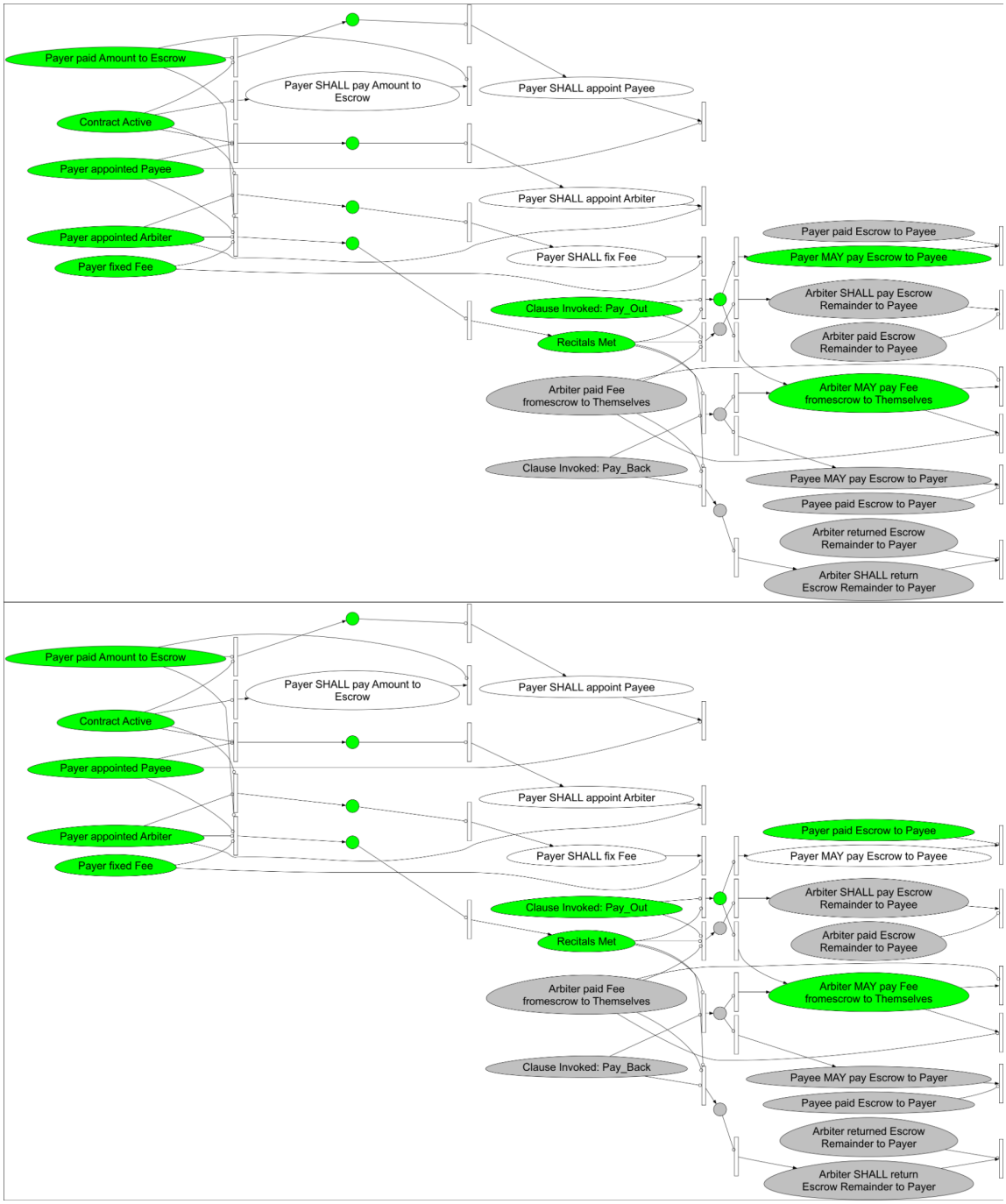
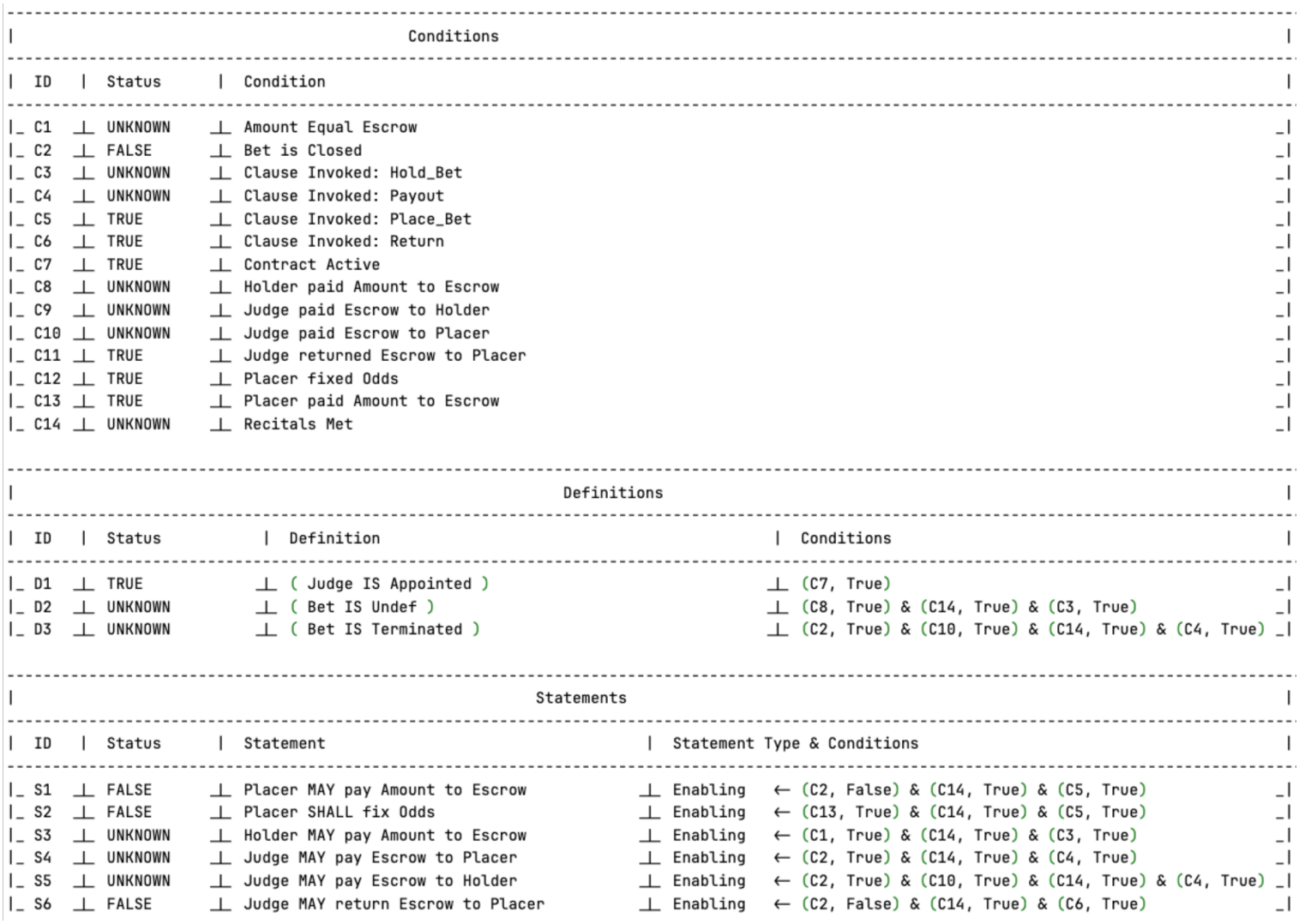


Figure A.16: Performance simulation of simple escrow agreement, Recitals met and Pay Out clause invoked. Scenario [(C6, True), (C11, True), (C9, True), (C8, True), (C10, True), (C5, True), (C12, True)]

A.2.2.2 Returnable Bet

```
LEX Returnable Bet.
LEXON: 0.2.12
COMMENT: 6.c - a bet between two parties, with odds
TERMS:
"Placer" is a person.
"Holder" is a person.
"Judge" is a person.
"Closed" is a binary.
"Bet" is this contract.
The Judge is appointed.

CLAUSE: Place Bet.
If the Bet is not Closed then
the Placer may pay an Amount into escrow,
and also fix the Odds.

CLAUSE: Hold Bet.
If the Amount is equal to the escrow times the Odds then
the Holder may pay the Amount into escrow,
and then the Bet is deemed Closed.

CLAUSE: Payout.
The Judge may If the Bet is Closed then pay the escrow to the Placer.
The Judge may If the Bet is Closed then pay the escrow to the Holder.
In any case, afterwards the Bet is terminated.

CLAUSE: Return.
The Judge may
if the Bet is not Closed then
return the escrow to the Placer.
```

Listing 18: Lexon contract, returnable bet between two parties with odds [3].



Conditions

ID	Status	Condition
C1	UNKNOWN	Amount Equal Escrow
C2	FALSE	Bet is Closed
C3	UNKNOWN	Clause Invoked: Hold_Bet
C4	UNKNOWN	Clause Invoked: Payout
C5	TRUE	Clause Invoked: Place_Bet
C6	TRUE	Clause Invoked: Return
C7	TRUE	Contract Active
C8	UNKNOWN	Holder paid Amount to Escrow
C9	UNKNOWN	Judge paid Escrow to Holder
C10	UNKNOWN	Judge paid Escrow to Placer
C11	TRUE	Judge returned Escrow to Placer
C12	TRUE	Placer fixed Odds
C13	TRUE	Placer paid Amount to Escrow
C14	UNKNOWN	Recitals Met

Definitions

ID	Status	Definition	Conditions
D1	TRUE	(Judge IS Appointed)	(C7, True)
D2	UNKNOWN	(Bet IS Undef)	(C8, True) & (C14, True) & (C3, True)
D3	UNKNOWN	(Bet IS Terminated)	(C2, True) & (C10, True) & (C14, True) & (C4, True)

Statements

ID	Status	Statement	Statement Type & Conditions
S1	FALSE	Placer MAY pay Amount to Escrow	Enabling ← (C2, False) & (C14, True) & (C5, True)
S2	FALSE	Placer SHALL fix Odds	Enabling ← (C13, True) & (C14, True) & (C5, True)
S3	UNKNOWN	Holder MAY pay Amount to Escrow	Enabling ← (C1, True) & (C14, True) & (C3, True)
S4	UNKNOWN	Judge MAY pay Escrow to Placer	Enabling ← (C2, True) & (C14, True) & (C4, True)
S5	UNKNOWN	Judge MAY pay Escrow to Holder	Enabling ← (C2, True) & (C10, True) & (C14, True) & (C4, True)
S6	FALSE	Judge MAY return Escrow to Placer	Enabling ← (C2, False) & (C14, True) & (C6, True)

Figure A.17: Screenshot of tabular summary: Returnable Bet between two parties (Listing 18).

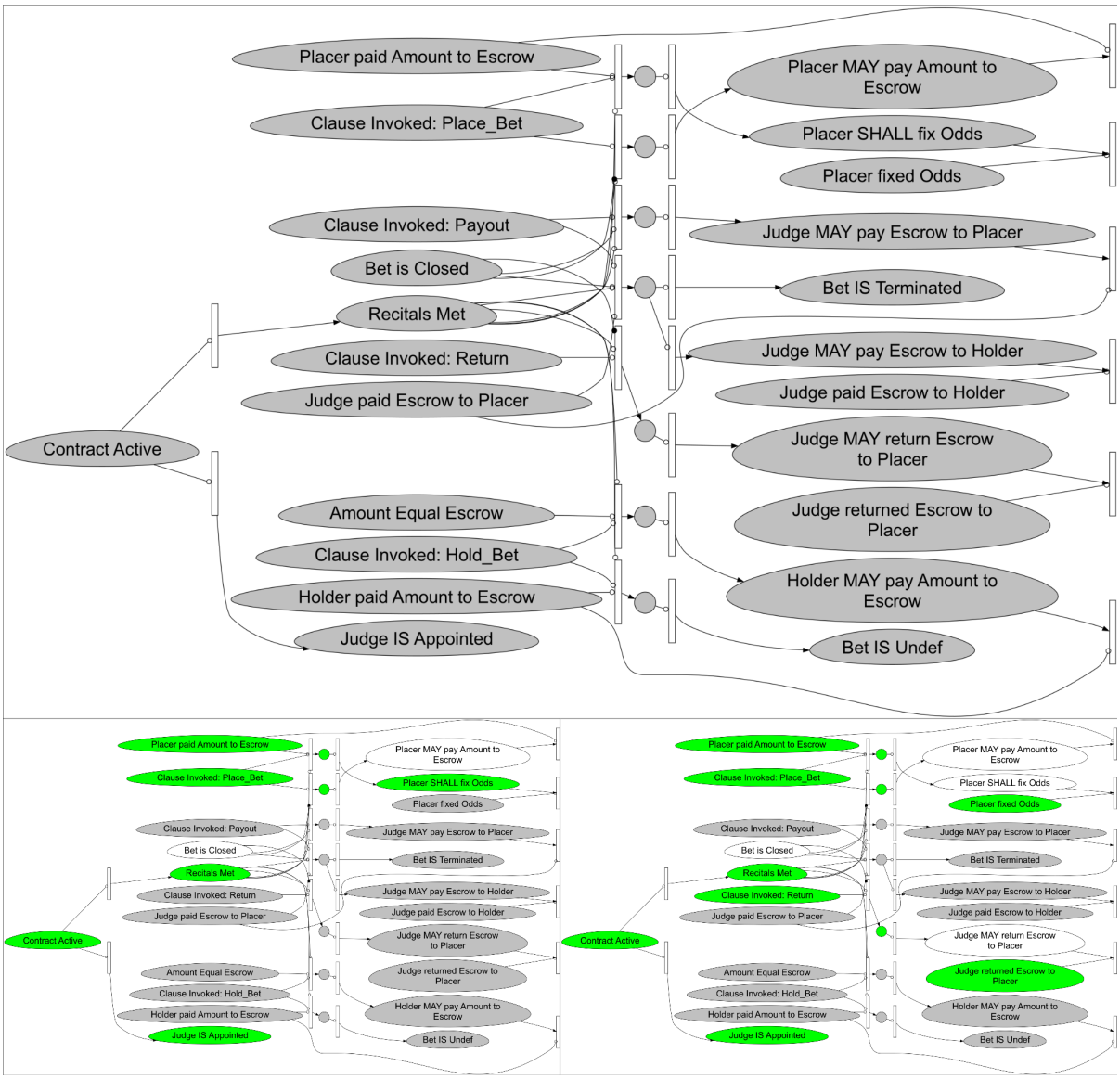


Figure A.18: Performance simulation of returnable bet. Scenario [(C7, True), (C5, True), (C2, False), (C13, True), (C12, True), (C6, True), (C11, True)]

A.2.2.3 UCC Financing Statement

```
LEX UCC Financing Statement.
LEXON: 0.2.12

"Financing Statement" is this contract.
"File Number" is data.
"Initial Statement Date" is a time.
"Filer" is a person.
"Debtor" is a person.
"Secured Party" is a person.
"Filing Office" is a person.
"Collateral" is data.
"Digital Asset Collateral" is an amount.
"Reminder Fee" is an amount.
"Continuation Window Start" is a time.
"Continuation Statement Date" is a time.
"Continuation Statement Filing Number" is data.
"Lapse Date" is a time.
"Default" is a binary.
"Continuation Statement" is a binary.
"Termination Statement" is a binary.
"Termination StatementTime" is a time.
"Notification Statement" is a text.
"Default" is a binary.

The Filer fixes the Filing Office,
fixes the Debtor,
fixes the Secured Party,
and fixes the Collateral.

Clause: Certify.
The Filing Office may certify the File Number.

Clause: Set File Date.
The Filing Office may fix the Initial Statement Date as the current time.

Clause: Set Lapse.
The Filing Office may fix the Lapse Date as now.

Clause: Pay Escrow In.
The Debtor may pay the Digital Asset Collateral into escrow.

Clause: Fail to Pay.
The Secured Party may fix Default as true.

Clause: Take Possession.
If Default is true then the Filing Office may pay the Digital Asset Collateral to
↪  the Secured Party.
```

```
47
48  Clause: File Continuation.
49  The Secured Party may fix the Continuation Statement as true.
50
51  Clause: Set Continuation Lapse.
52  If the Continuation Statement is true then the Filing Office may fix the
    ↪  Continuation Statement Date.
53
54  Clause: File Termination.
55  The Secured Party may fix Termination Statement as true,
56  and fix the Termination StatementTime as the current time.
57
58  Clause: Release Escrow.
59  If the Termination Statement is true then The Filing Office may pay the Digital
    ↪  Asset Collateral to the Debtor.
60
61  Clause: Release ReminderFee.
62  If the Termination Statement is true then The Filing Office may pay the Reminder
    ↪  Fee to the Secured Party.
```

Listing 19: Lexon Contract, UCC Financing Statement [3].

Conditions

ID	Status	Condition
C1	UNKNOWN	Clause Invoked: Certify
C2	UNKNOWN	Clause Invoked: Fail_to_Pay
C3	UNKNOWN	Clause Invoked: File_Continuation
C4	UNKNOWN	Clause Invoked: File_Termination
C5	UNKNOWN	Clause Invoked: Pay_Escrow_In
C6	UNKNOWN	Clause Invoked: Release_Escrow
C7	UNKNOWN	Clause Invoked: Release_ReminderFee
C8	UNKNOWN	Clause Invoked: Set_Continuation_Lapse
C9	UNKNOWN	Clause Invoked: Set_File_Date
C10	UNKNOWN	Clause Invoked: Set_Lapse
C11	UNKNOWN	Clause Invoked: Take_Possession
C12	UNKNOWN	Continuation_Statement is True
C13	UNKNOWN	Contract Active
C14	UNKNOWN	Debtor paid Digital_asset_collateral to Escrow
C15	UNKNOWN	Default is True
C16	UNKNOWN	Filer fixed Collateral
C17	UNKNOWN	Filer fixed Debtor
C18	UNKNOWN	Filer fixed Filing_Office
C19	UNKNOWN	Filer fixed Secured_Party
C20	UNKNOWN	Filing_office certified File_Number
C21	UNKNOWN	Filing_office fixed Continuation_Statement_Date
C22	UNKNOWN	Filing_office fixed Initial_Statement_Date
C23	UNKNOWN	Filing_office fixed Lapse_Date
C24	UNKNOWN	Filing_office paid Digital_asset_collateral to Debtor
C25	UNKNOWN	Filing_office paid Digital_asset_collateral to Secured_party
C26	UNKNOWN	Filing_office paid Reminder_fee to Secured_party
C27	UNKNOWN	Recitals Met
C28	UNKNOWN	Secured_party fixed Continuation_Statement
C29	UNKNOWN	Secured_party fixed Default
C30	UNKNOWN	Secured_party fixed Termination_Statement
C31	UNKNOWN	Secured_party fixed Termination_StatementTime
C32	UNKNOWN	Termination_Statement is True

Statement

ID	Status	Statement	Statement Type & Conditions
S1	UNKNOWN	Filer SHALL fix Filing_Office	Enabling ← (C13, True)
S2	UNKNOWN	Filer SHALL fix Debtor	Enabling ← (C18, True) & (C13, True)
S3	UNKNOWN	Filer SHALL fix Secured_Party	Enabling ← (C17, True) & (C13, True)
S4	UNKNOWN	Filer SHALL fix Collateral	Enabling ← (C19, True) & (C13, True)
S5	UNKNOWN	Filing_office MAY certify File_Number	Enabling ← (C27, True) & (C1, True)
S6	UNKNOWN	Filing_office MAY fix Initial_Statement_Date	Enabling ← (C27, True) & (C9, True)
S7	UNKNOWN	Filing_office MAY fix Lapse_Date	Enabling ← (C27, True) & (C10, True)
S8	UNKNOWN	Debtor MAY pay Digital_asset_collateral to Escrow	Enabling ← (C27, True) & (C5, True)
S9	UNKNOWN	Secured_party MAY fix Default	Enabling ← (C27, True) & (C2, True)
S10	UNKNOWN	Filing_office MAY pay Digital_asset_collateral to Secured_party	Enabling ← (C15, True) & (C27, True) & (C11, True)
S11	UNKNOWN	Secured_party MAY fix Continuation_Statement	Enabling ← (C27, True) & (C3, True)
S12	UNKNOWN	Filing_office MAY fix Continuation_Statement_Date	Enabling ← (C12, True) & (C27, True) & (C8, True)
S13	UNKNOWN	Secured_party MAY fix Termination_Statement	Enabling ← (C27, True) & (C4, True)
S14	UNKNOWN	Secured_party SHALL fix Termination_StatementTime	Enabling ← (C30, True) & (C27, True) & (C4, True)
S15	UNKNOWN	Filing_office MAY pay Digital_asset_collateral to Debtor	Enabling ← (C32, True) & (C27, True) & (C6, True)
S16	UNKNOWN	Filing_office MAY pay Reminder_fee to Secured_party	Enabling ← (C32, True) & (C27, True) & (C7, True)

Figure A.19: Screenshot of tabular summary: UCC Financing Statement (Listing 19).

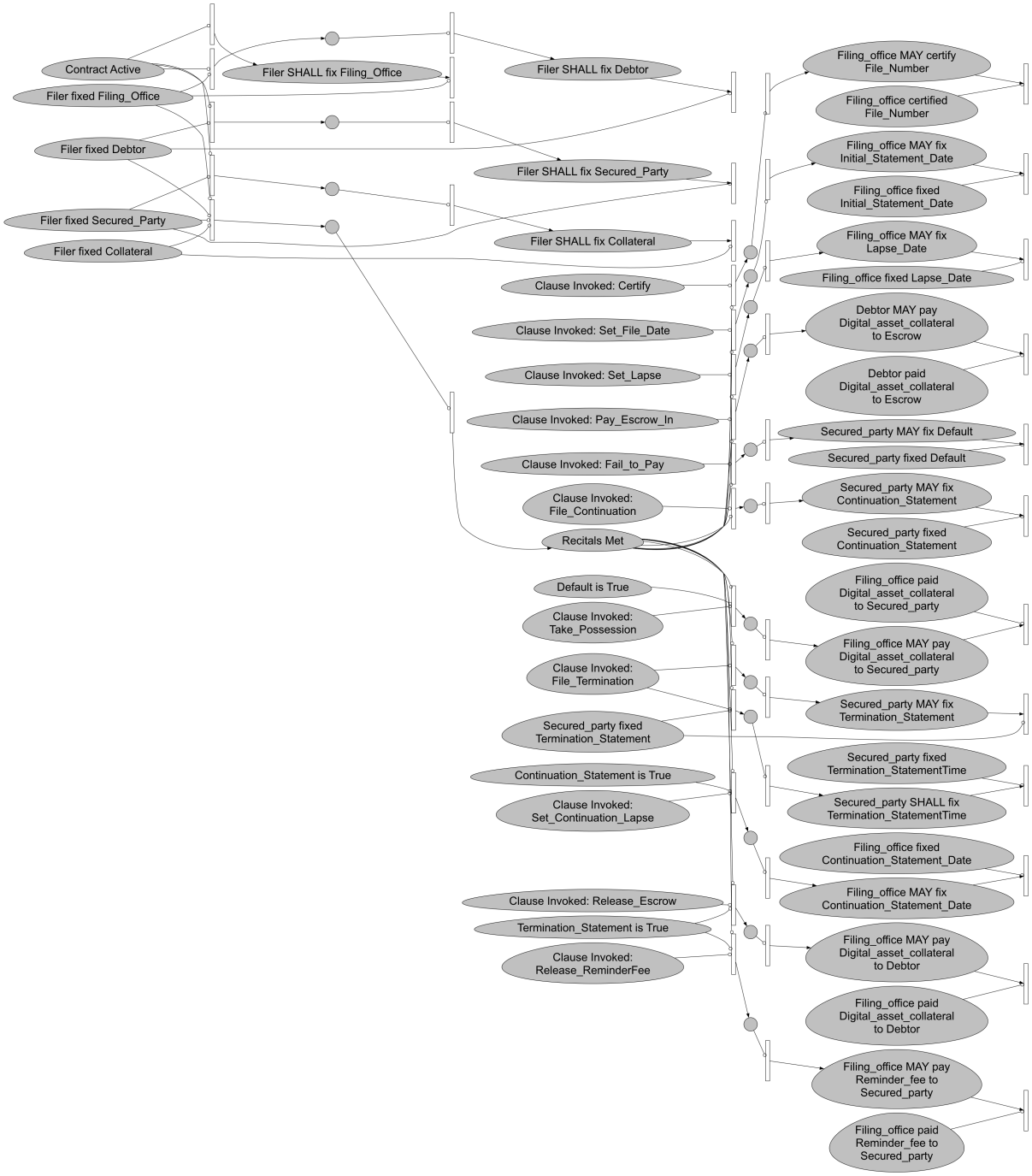


Figure A.20: Performance simulation, UCC Financing Statement with no events.

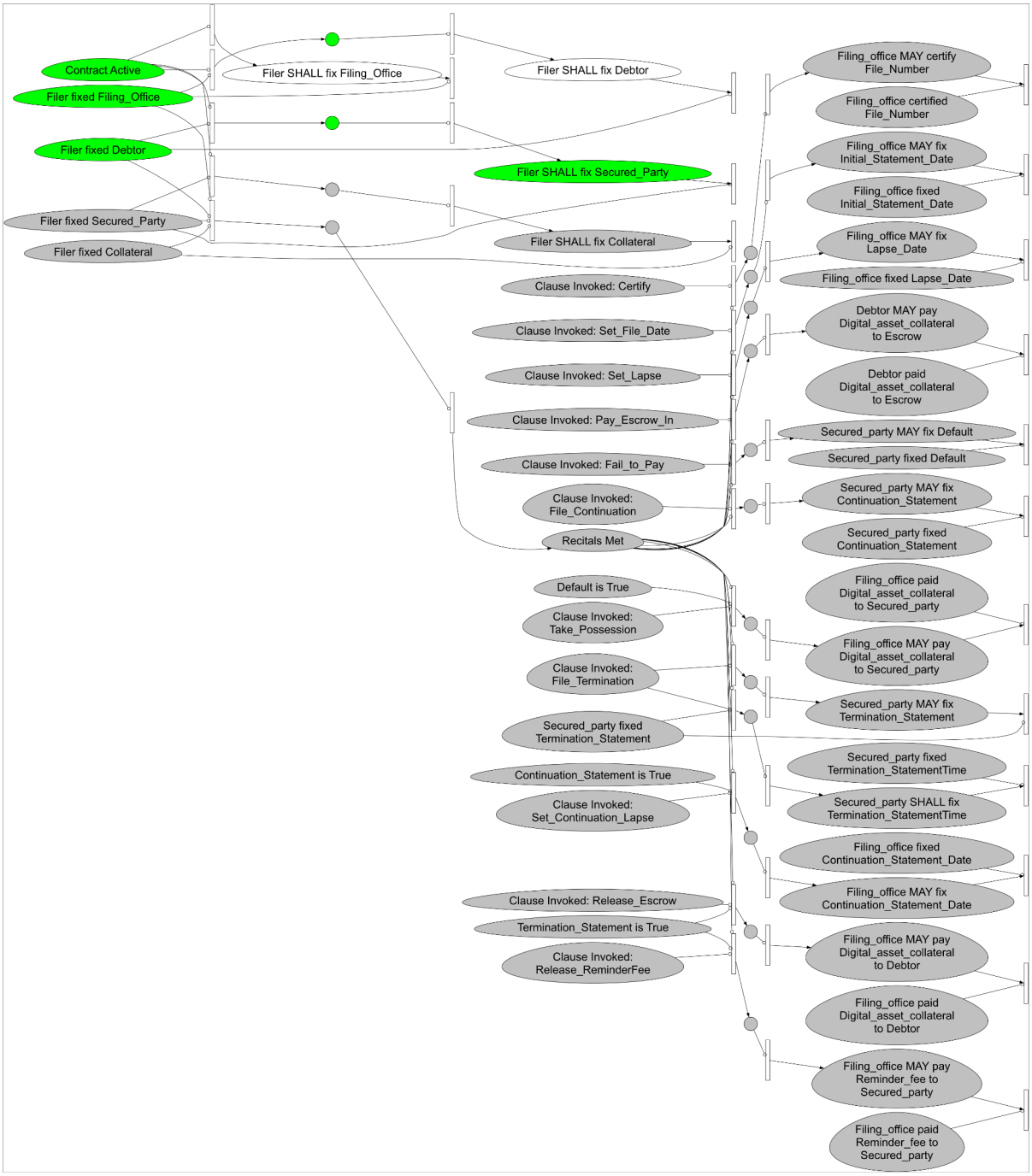


Figure A.21: Performance simulation, UCC Financing Statement. Scenario [(13, True), C18, True), (C17, True)]

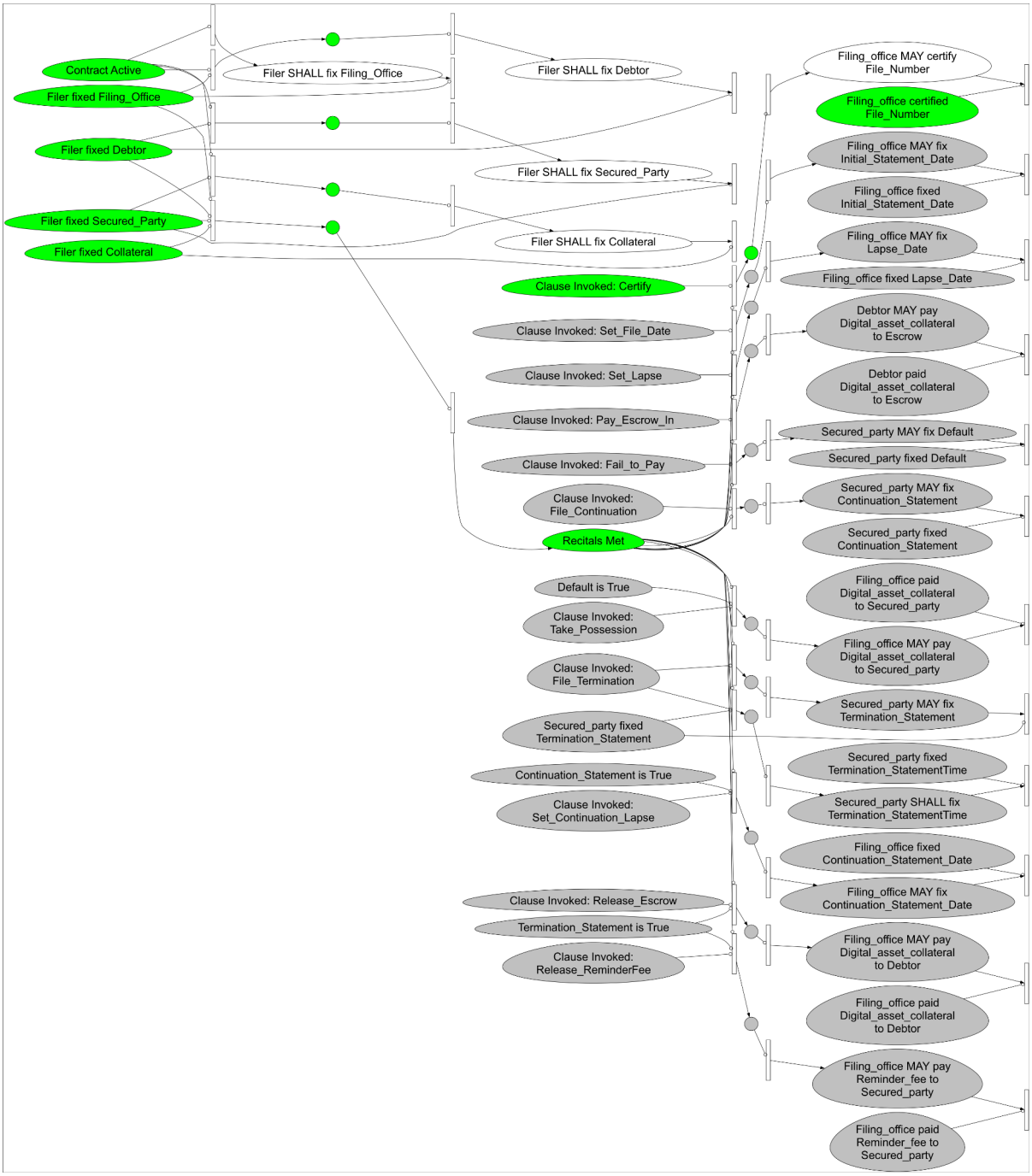


Figure A.22: Performance simulation, UCC Financing Statement. Scenario [(13, True), C18, True), (C17, True), (C16, True), (C1, True), (C28, True)]

B | User Guide

B.1 CoLa Contracts

To convert a CoLa contract into the Petri Net and start the performance simulation, the following steps should be taken.

1. Install Miranda on your local machine.
2. Download the CoLa file, available in the project GitHub repository GitHub.
3. Run the following command against any valid CoLa contract 'cola_to_python ...'
4. This will generate the Python Code to generate the Petri Net and simulate the contract. An example of this can be seen in Figure B.1.
5. Copy the generated Python code and paste it into the './Contract Model/-Cola_tests.py' file, as shown in Figure B.2, and run the programme.

```
Miranda cola_to_python isda
con = Contract()

con.statement(TemporalStatement('"ExcessParty"', 'SHALL', 'Pay', 'SomeCurrency "ExcessAmount"', TemporalExpression('ON','(1,1,1970)'), valid=True))

con.statement(ConditionalStatement(condition=AndCondition(conditions=[TemporalActionCondition('"PartyA"', 'Paid' , 'SomeCurrency "AmountA"', TemporalExpression('ON','(1,1,1970)'), test=True),
                TemporalActionCondition('"PartyB"', 'Paid' , 'SomeCurrency "AmountB"', TemporalExpression('ON','(1,1,1970)'), test=True),
                ]), statement=TemporalStatement('"PartyA"', 'SHALL', 'Pay', 'SomeCurrency "AmountA"', TemporalExpression('ON','(1,1,1970)'), valid=False)))

con.statement(ConditionalStatement(condition=AndCondition(conditions=[TemporalActionCondition('"PartyA"', 'Paid' , 'SomeCurrency "AmountA"', TemporalExpression('ON','(1,1,1970)'), test=True),
                TemporalActionCondition('"PartyB"', 'Paid' , 'SomeCurrency "AmountB"', TemporalExpression('ON','(1,1,1970)'), test=True),
                ]), statement=TemporalStatement('"PartyB"', 'SHALL', 'Pay', 'SomeCurrency "AmountB"', TemporalExpression('ON','(1,1,1970)'), valid=False)))

con.definition(ConditionalDefinition(condition=ExpressionCondition(BooleanExpression('"PartyB"', 'Paid', 'MoreThan', '"PartyA"'), test=True), definitions=[IsDefinition('"ExcessParty"', '"PartyB"'),
                EqualsDefinition('SomeCurrency "ExcessAmount"', 'OtherObject "AmountB" MINUS OtherObject "AmountA"'),
                ])
)

con.definition(ConditionalDefinition(condition=ExpressionCondition(BooleanExpression('"PartyA"', 'Paid', 'MoreThan', '"PartyB"'), test=True), definitions=[IsDefinition('"ExcessParty"', '"PartyA"'),
                EqualsDefinition('SomeCurrency "ExcessAmount"', 'OtherObject "AmountA" MINUS OtherObject "AmountB"'),
                ])
)

con.interaciveSimulation()
```

Figure B.1: To generate the Python code for Petri Net generation, run 'cola_to_python' function against any valid CoLa contract. This will result in the Python code representative of the contract.

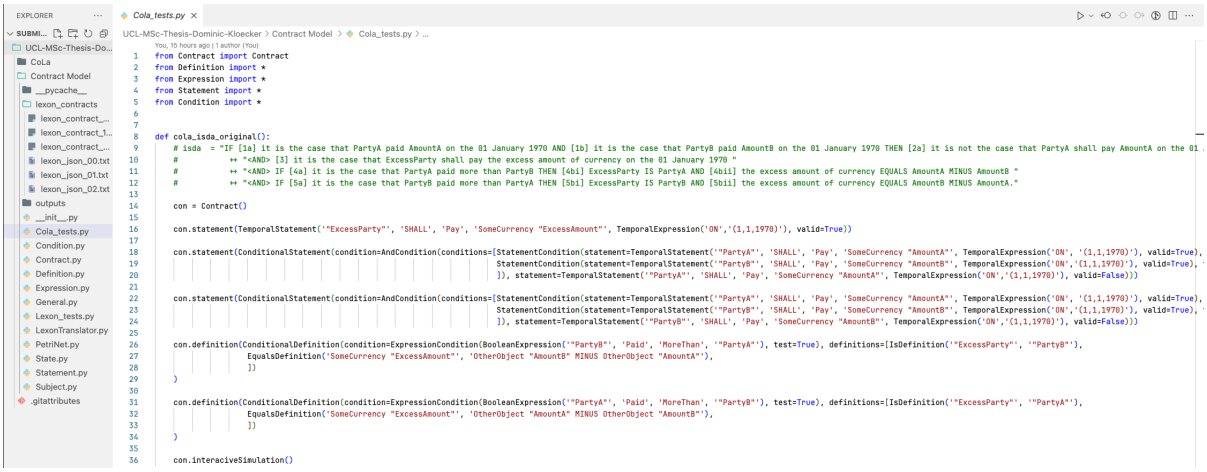


Figure B.2: Upon pasting of the Python Code generated by the Lexon syntax translation into the 'Cola_tests.py' file, generate the Petri Net by running the programme.

B.2 Lexon Contracts

To convert a Lexon contract into the Petri Net and start the performance simulation, the following steps should be taken.

1. Locally install the Lexon Compiler from the official Lexon GitLab.
2. Make the modifications shown in Listing 14 to the Lexon code in the './src/main.rs' file.
3. Once these changes are made, follow the guidelines in the Lexon README.md file to compile a Lexon contract.
4. Once a contract is compiled, its AST in JSON format will be generated inside the Lexon directory in a './json_ast/contract_ast.txt' file as shown in Figure B.3.
5. Copy the generated file into the Python directory of the contract model in the './lexon_contracts' directory.
6. Read and process the AST using the LexonTranslator class as shown in Figure B.4.

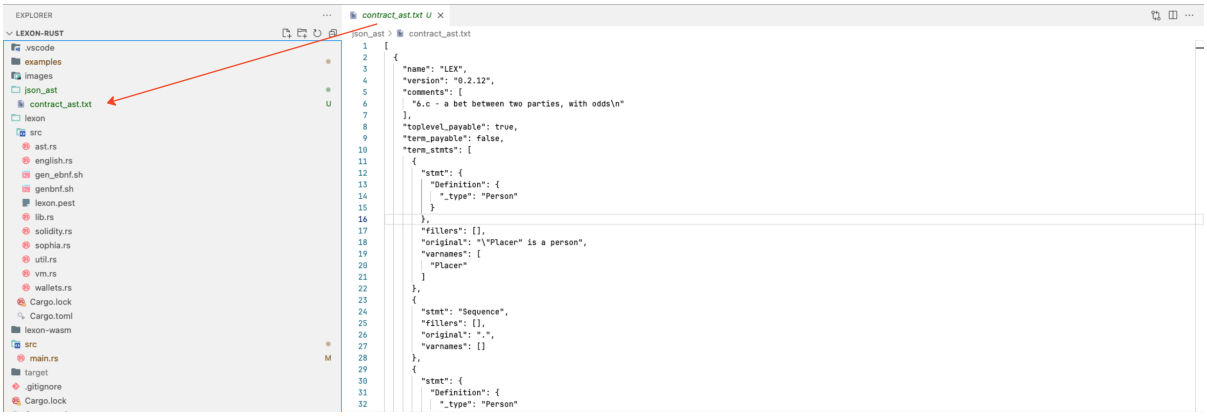


Figure B.3: Generated Lexon AST file

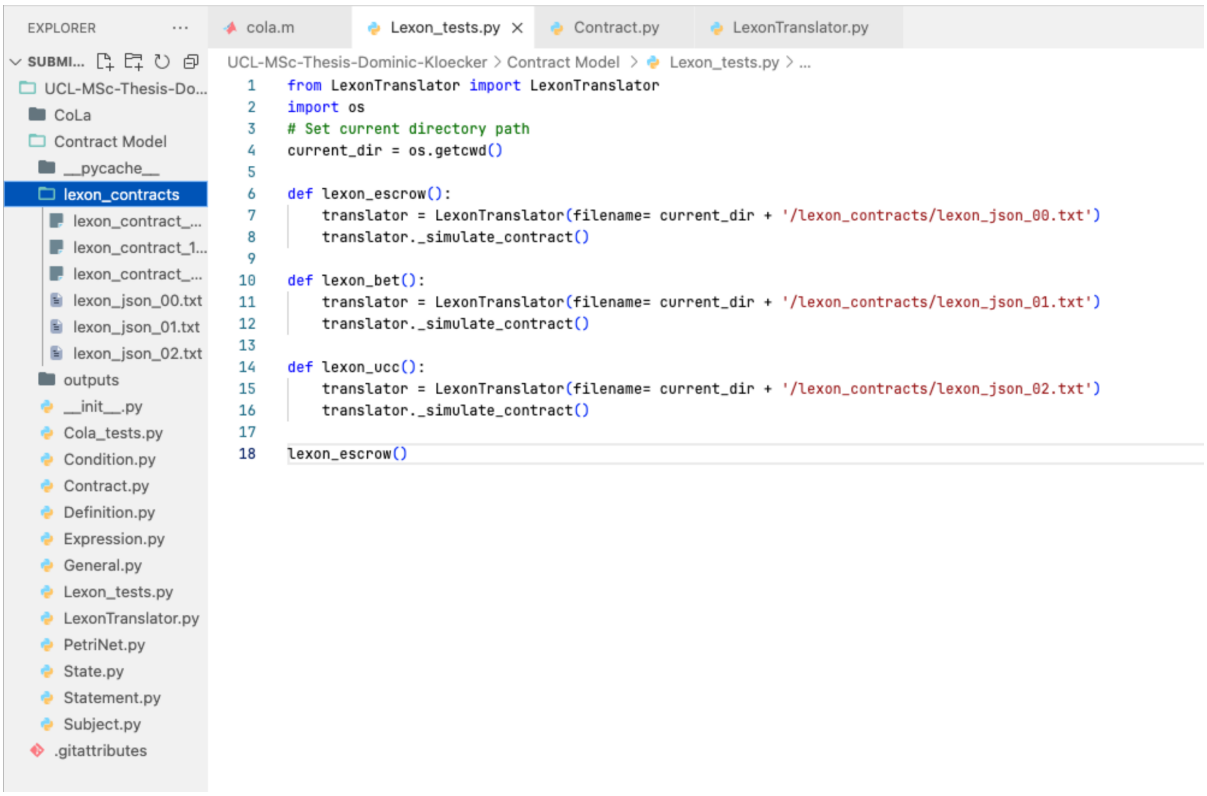


Figure B.4: Paste the AST file generated by the compiler into '.Contract Model/lexon_contracts' directory. You can then generate the Petri Net and run the simulation by referencing the text file similarly to the examples shown in this image.