LONDON'S GLOBAL UNIVERSITY

THE SEMANTICS OF SMART CONTRACTS USED IN BANKING AND FINANCIAL SERVICES

GABRIEL VANCA

BSc Computer Science

Supervisor:

DR CHRISTOPHER D. CLACK

Department of Computer Science

University College London

April 29, 2018

Abstract

The project takes an in-depth view of the idea of digitalised legal contracting and it creates a new viable model for semantic analysis of high-value, long-duration, highly-regulated, standardised smart contracts that can work to automate various procedures in banking and financial services, making them faster, cheaper to implement, easier to use and more accessible to customers. It analyses various deontic components such as rights, prohibitions and obligations, how these combine with the operational aspects of a legal contract and how they are related to temporal environments such as absolute, relative, recurrent, discrete and continuous time.

The dissertation reviews existing techniques and logic systems for representing smart contracts. It brings forward new formalism, new types of logical operators and constructs the basis of a concrete system of rules for correctly built smart contracts. The project places the foundations of template-based contract design.

The project provides a proof-of-concept application for Petri-net visualisations of key components of the new model and for simulation of various contractual scenarios. The proof-of-concept is used to test and validate the model and a critical evaluation of the model is provided, alongside future research directions.

Acknowledgements

I would like to express my gratitude to Dr Christopher D. Clack without whose continued support and guidance, this dissertation would not have been possible. I am extremely grateful for his invaluable help with the research and the project.

Contents

1 Introduction

1.1 Research Motivation

Despite finding ourselves in the Digital Age, financial institutions are saddled with out of date IT systems, millions of faxes are being sent each year and trades are handled manually many times. These opaque methods lock up capital and slow down processes, costing the industry high sums of money and slowing growth. In this context where reliance on physical documents has lead to costly delays, high overhead costs, difficulty to achieve financial regulatory compliance, increased exposures to errors and fraud [1], the financial industry has been looking for innovative ways to trade and operate.

The industry's interest was recently captivated by smart contracts and the possibility of running them on secure, anonymous and immutable distributed ledgers. Smart contracts will greatly help most financial sectors and operations. In investment banking, clients would benefit from settlements cycles reduced from 20-48 days to 6-10 days. In retail banking, costs of loans and mortgages would decrease by hundreds of pounds. In insurance, customers would likely see lower premiums due to automated processing of claims and increased fraud prevention mechanisms. [1]

This dissertation looks at how smart contracts can be used in banking and financial services and specifically at the semantic model they operate on.

1.2 Smart Contracts and the Semantic Challenges

The term 'smart contract' has been given various meanings: from software agents to the way legal contracts can be implemented in software [2].[1] Looking at the legal text for financial contracts, one notices that it has considerable complexities that have to be represented using a formal semantic model that is unambiguous, makes the smart contract computable and enables the analysis of the semantic representation. [3]

Clack et al [4] point out that legal documentation for financial contracts contains both operational aspects (the actions to be performed) and non-operational (the timing of various actions, the rights, obli-

[1] Section 2.1 contains a wider discussion on the terminology of "smart contracts".

gations, prohibitions of the parties etc). The way these aspects are intertwined shows that there is a need for a formal model that can combine various semantic aspects.

This has been a long-time interest of the research community, with a key paper on the topic of semantic models for smart-contracting by Ronald Lee dating back from 1988[5]. The current paper will analyse in depth Lee's model as well as look into further research conducted on the topic of semantic models for smart contracting.

A supporting software program will also be provided with this project to enable graphical representations of the logical model under the form of Petri nets.

Clack [3] notes the benefits of formal analysis of the legal documentation: highlighting inconsistencies, ambiguities, missing cases and generating 'test cases' for the testing smart contract code and simulating real-life situations, without the high cost of legal fees.

1.3 Research Aim and Objectives

The main aim of this project is to investigate a viable semantic system for the construction of high-value, standardised smart contracts for financial services. The project has the following objectives:

1. Review existing techniques and logics for representing smart contracts in finance.
2. Provide an in-depth critical analysis of an existing semantic model by Lee [5]
3. Create a new model for semantic analysis of smart contracts, as an extension to Lee's model.
4. Create a proof-of-concept application for Petri-net visualisations of key components of the new model.
5. Use the proof-of-concept to test and validate aspects of the model, including a critical evaluation of the new model.

1.4 Report Structure

The structure of the current thesis is as follows:

Chapter 2 - Literature Review

This chapter looks more in-depth at what smart contracts are and how they work. It takes into account smart contract enforceability and it analyses the research into various logical aspects and

how to combine different types of logics into a single formal semantic model.

Chapter 3 - The Syntax of Lee's Formalism

This is a presentation of Lee's formalism and the syntax used in his papers on smart contracting [5, 6, 7, 8] which will place the basis for the semantic system analysis work conducted in Chapter 4.

Chapter 4 - Critical Analysis of Lee's Model

An in-depth analysis of Lee's semantic model for smart contracting and later proposed modifications based on his work.

Chapter 5 - Design and Implementation of the Revised Semantic Model

This chapter provides proposals on improving Lee's existing semantic model and formalism for smart-contracting and introduces a few new and notable concepts in order to construct a more robust and future-proof model.

Chapter 6 - Testing and Validation

This chapter briefly details the importance of validating the semantic model and it looks at how various validations of the proposed approach can be made.

Chapter 7 - Summary and Conclusions

The final chapter provides a critical assessment of the proposed approach and it discusses future research that needs directions for this topic.

2 | Literature Review

2.1 Smart Contracts

Terminology

To harmonise the various meanings given to the term "smart contract", ever since the term was first used by Nick Szabo in 1996 [9, 10], and the terminology conflicts that arise from the interaction between the disciplines of Computer Science, Law and Banking, Clack et al [4] give the following portmanteau definition adopted herein:

> *A smart contract is an automatable and enforceable agreement. Automatable by a computer, although some parts may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code.*

The reason this definition was chosen is that it is broad and abstract enough to cover the multitude of meanings given to the term, including the two most common, as outlined by J. Stark [2]:

1. Software agents that typically run on a shared ledger, but are not necessarily linked to an actual legal agreement. Definitions of these type live usually in the Blockchain community, but there is no consensus on one exact definition, each being slightly different. Stark calls the definition imperfect and misleading as it emphasizes a single narrow use case and, therefore, renames these software agents as "smart contract code".
2. A contract that constitutes a legal agreement complemented or replaced by smart contract code. This is a specific use of the software agents and this definition is more common amongst those working in the financial and legal sectors. Stark calls these "smart legal agreements".

Enforceability

The two essential requirements of smart contracts presented in the definition given by Clack et al [4] are automation and enforceability. Traditional enforcement of legal contracts includes dispute resolution methods such as arbitration or recourse to courts of law in case of wrong-performance and non-performance of contractual obligations or in the case of illegality. However, this might not always be

ideal as such procedures can be lengthy and costly. Non-traditional enforcement is implemented at the digital level. Where smart contracts are implemented on distributed ledgers, the result is a "tamper-proof" contract in which, theoretically, non-performance or wrong-performance becomes impossible.

However, Clack [4, 3] points out that pure tamper-proof execution is not adequate for high-value, long-duration, highly-regulated financial contracts. This is because tamper-proof code once launched cannot be changed. It is however common for financial agreements to support discretion and flexibility in performance such as deferred payments or to permit a payment holiday. In a tamper-proof system, all these possible variations would need to be envisaged and coded in advance which would bar discretion and flexibility. Furthermore, even if all theoretical variations of the agreement could be encoded in the legal prose and in code, the solution still fail because when authoring the legal prose, the parties cannot take into consideration all future law and regulatory changes.

Therefore Clack et al [3, 4] rightly point out that the semantic model for smart contracts has to allow for "executive override" provisions and also for human input, especially as objects and actions in the physical world are unlikely to be under the control of the smart contract code. Clack suggests stopping the contract, changing the term of the agreement, and starting a new contract from the state where the old one stopped as a possible solution to the problem of enforcing high-value, long-duration, highly-regulated banking and financial contracts. This has to be, however, supported by the semantic model.

2.2 Semantic Modelling and Related Challenges

2.2.1 Types of Semantics Analysis and Non-Separability

Clack [11] talks about three fundamental types of semantic analysis for smart contracts. The first one is temporal semantics which covers the time aspects of the agreement. Deontic semantics is the second one and it covers the aspects related to the rights, obligations and prohibitions of the involved contractual parties. The last one, operational semantics, refers to actions, even to those which cannot be or the lawyer does not want to encode due to enforceability reasons. These types of semantic analysis have previously been developed and investigated separately, but Clack observes that they interact and are rarely separable. For example, a temporal aspect may be often linked to an operational one. In the phrase "upon reasonable demand", "upon" is temporal and "demand" is operational. All three aspects can be combined and tightly integrated within a single phrase.

Clack et al note that even though legal contracts are operationally bound, they have non-operational semantics as "In a contract all actions derive from an obligation or right of some form (a deontic aspect)" [11], but also that "Many actions have embedded temporal aspects and may have embedded deontic aspects" [11]. Therefore, seeing that non-operational aspects are unavoidable in a contract by the very nature of legal contracts and that they are inseparable due to their very tight integration, it becomes apparent that there is a need for such a semantic model that allows combining these three types of semantic analysis.

2.2.2 Requirements and the Choice for a Semantic Model

Hvitved [12] reaches the same conclusion as Clack [11] when talking about the need to combine the three types of semantic analysis: deontic, temporal and operational and defines a set of 16 requirements that a semantic model must support based on various aspects found in contracts:

(R1) Contract model, contract language, and a formal semantics.
(R2) Contract participants.
(R3) (Conditional) commitments.
(R4) Absolute temporal constraints.
(R5) Relative temporal constraints.
(R6) Reparation clauses.
(R7) Instantaneous and continuous actions.
(R8) Potentially infinite and repetitive contracts.
(R9) Time-varying, external dependencies.
(R10) History-sensitive commitments.
(R11) In-place expressions.
(R12) Parametrised contracts.
(R13) Isomorphic encoding.
(R14) Run-time monitoring.
(R15) Blame assignment.
(R16) Amenability to (compositional) analysis.

It might appear that the requirement for semantic support of rights and prohibitions is absent. However, Hvitved refers to permissions and prohibitions as types of commitments (requirement R3), based on Von Wright's [13] theory where permissions and prohibitions are defined as types of obligations. In theory, a prohibition is defined as an obligation of not doing something. On the other side, a permission is defined in two ways: Von Wrights defines it as the lack of an obligation of not doing something, whilst Giordano [14] defines it as an obligation for the other party that becomes active once the party that holds the permission invokes said right (i.e. the right of one party is an obligation for the other party to respect the first party's right and facilitate the permitted action).

Hvitved concludes that well-established formalism such as deontic logics, temporal logics and timed

automata are inadequate for modelling all details of contracts on their own. Consequently, he looks at several new models and languages for specifying contracts that have been proposed and, after in-depth analysis, rates them accordingly with the previously mentioned requirements.

	Lee	Goo	Bou	Pey	Mol	Mil	And	Pri
R1							✓	✓
R2	✓		✓			✓	✓	
R3	✓	✓	✓	✓	✓	✓	✓	✓
R4	✓	✓	✓	✓	(✓)	✓	✓	
R5	✓			✓	✓	✓	✓	✓
R6	✓		✓		✓	✓	✓	✓
R7	✓							
R8	✓			✓	✓		✓	✓
R9				✓			✓	
R10							✓	
R11		✓		✓			✓	
R12	✓			✓			✓	
R13	✓		✓	(✓)		✓		
R14	✓	✓	✓		✓	✓	✓	✓
R15	(✓)		(✓)					
R16	✓			✓	✓	✓	✓	✓

Figure 2.1: Hvitved's formalisms comparison matrix (contract formalisms horizontally, requirements vertically) [12]. The compared formalisms are the following: Lee is Ronald Lee's formalism [5], Goo is Goodchild et al. [15], Bou is Boulmakoul and Salle [16], Pey is Peyton Jones and Eber [17], Mol is Molina-Jimenez et al [18], Mil is Milosevic et al. [19, 20, 21], And is Andersen et al. [22] and Pri is Prisacariu and Schneider [23, 24]

Hvitved [12] notes that most approaches lack detail, formal semantics, neglect formal mathematical underpinnings or provide incomplete mathematical models and semantics. Priscariu et al. [23, 24] and Andersen et al. [22] are the only ones that do not have those issues, but they lack empirical evidence that their systems adequately capture contracts of the intended domains, as well as other very important features such as support for instantaneous and continuous actions.

Lee's model [6] captures most of the aspects required by contracts. Though Lee's model is not what Hvitved calls "a silver bullet" for smart contracts semantic modelling due to its lack in terms of formalism, it offers the support to build a semantic model to solve its inherent shortcomings. Therefore, this paper will take an in-depth analysis of Lee's semantic model and propose ways to solve its semantic shortcomings.

2.2.3 Deontic and Temporal Aspects in Electronic Contracting

2.2.3.1 Deontic Aspects

In a contract, actions do not simply occur. They are obligatory, permitted or prohibited. These "deontic" concepts were first proposed by Von Wright in his 1968 paper "An Essay in Deontic Logic and the General Theory of Action" [13] which is now widely recognised as the foundation paper for deontic logic. Von Wright introduced the "obligatory" operator: O. Let there be an action noted Φ. $O \quad \Phi$ means that action Φ is obligatory or an "obligation".

Permission (noted P) is defined in terms of obligation as:

$$P\Phi \leftrightarrow \neg O \neg \Phi \tag{2.1}$$

which means that to be permitted Φ is not to be obliged not to do Φ. [1]

Similarly, the concept of prohibition, or impermissibility, (noted F) is defined:

$$F\Phi \leftrightarrow O \neg \Phi \tag{2.2}$$

which means that to be prohibited (or forbidden) Φ is to be obliged to not do Φ.

2.2.3.2 Temporal Aspects

Linington et al. [19] note that temporal constraints are essential to most legal contracts. They are used to express either global conditions (e.g. start date of the contract) or conditions that refer to various deontic modalities. Such temporal aspects are typically expressed in one of the following two forms: deadlines and intervals. Deadlines might be used for discharging obligations (e.g. credit repayment should be made in 25 days after the receipt of a credit statement) whilst intervals might be used (not exclusively) for permitting a behaviour (e.g. the times of the week when certain transfer fees apply).

The most basic specification of time is an absolute time point (e.g. "09.00, 1 September 2018"). Time points may also be specified relative to some other time points (e.g. "the payment deadline is 8 days after the opening of the bank account"). These temporal aspects can also be less trivial. For instance, a contract might specify recurring intervals (e.g. "each Monday") or combined recurring intervals (e.g."between 9 AM to 4 PM during Weekdays"). Finally, temporal expression can take quite a complex

[1] This definition is considered inadequate by Giordano [14] and by Linington et al. [19]. See Subsection 2.2.3.3 for a discussion on that.

temporal form. For instance "There must be no more than three occurrences of downtime within any one-week period"). This is known as a sliding time-window. [19]

2.2.3.3 Obligations, Permissions, Prohibitions and Associated Temporal Aspects

Linington et al. [19] note that obligations deal with expected behaviours whilst permissions and prohibitions deal with possible behaviours. Furthermore, they note that this makes a fundamental difference in the comparative difficulty of checking that requirements are satisfied.

An obligation requires X to engage in some behaviour and is discharged if the required behaviour is observed.[2] However, it might not be so obvious identifying when a violation has occurred as there might be different levels of urgency. This means that in practice the situation is only straightforward if there is a defined deadline or temporal ordering requirement on the performance of a contractual obligation.

A prohibition on X engaging in an action role is satisfied so long as no such action is observed. Any counter-example is sufficient to show a violation.

Finally, a permission for X to perform an action role is associated with the moment of the action, but this might take indefinitely long to observe if X does not choose to exercise his rights. Furthermore, observation of an attempt to perform the action failing in a way that indicates the permission has been withheld is evidence of a violation of that said right. However, the action might fail for other reasons, so without a clear indication of why the failure took place, there is no evidence of a violation. This involves some very practical issues: e.g. "*At what point ... does declaring a resource busy every time an attempt is made to use it amounts to a failure to honour the permission to use it?*".[19]

2.2.3.4 Classification of Obligations from a Temporal Perspective

Giordano et al [14] raise a very important point that is overlooked in other papers on contractual deontic logic. They define different types of obligations: achievement, contrary-to-duty, maintenance, punctual, preemptive and non-preemptive. As it will be revealed in Chapter 4, this is an important point to make because different types of obligations need to be implemented in distinctive ways.

Achievement Obligations - Achievement obligations require a given condition to occur at least once before a deadline. An example of that is a customer having to repay a specified portion of his credit

[2] In Subsection 2.2.3.4, there is a discussion on the claim made by Giordano et al [14] that the situation is actually more complex and that Linington et al.'s definition is not always the case.

within 25 days of receipt of his monthly credit statement. The action of issuing the credit statement generates (or "invokes") an obligation to pay within a deadline. The obligation persists until it is fulfilled, cancelled or, in some cases, violated. Just because an obligation was violated that does not mean that it stops being enforced. If not fulfilled or cancelled, the obligation persists until the deadline. If the obligation is cancelled, the persistence becomes blocked. An achievement obligation neither fulfilled nor violated is "pending".

Achievement obligations can be preemptive or non-preemptive. A preemptive obligation can be fulfilled any time before the deadline, even before the obligation being invoked. A non-preemptive obligation has to be fulfilled in the time-span between being invoked and the deadline.[3]

Maintenance Obligations - Maintenance obligations require a condition to be fulfilled during all time instances before a deadline. For example, after opening a flexible deposit account with a bank, the customers must keep a balance of over £1,000 until the account is closed. A maintenance obligation persists until the deadline is reached, cancelled or, in some cases, violated.

Contrary-to-duty Obligations - A contrary-to-duty obligation is a new obligation that is caused by a previous obligation being violated. E.g. An obligation says that a customer has to repay a loan within 65 days of lending. The deadline passes and the obligation is violated. The achievement obligation to pay the sum persists and a contrary-to-duty obligation that says that the sum has to be paid with a fine within two working days is invoked. The fulfilment of the contrary-to-duty obligation to pay with a fine compensates the earlier violation. However, the initial obligation might still need to be fulfilled.

Punctual Obligations - Punctual obligations involve instantaneous actions. E.g. If a system receives a message, it must immediately acknowledge it. However, Giordano et al fail to consider that legally there can be no expectation that something is done instantaneous, so the obliged action must start almost immediately after the obligation has been invoked. However, if the action takes a long amount of time to be completed, there can be no expectation that the action will also be finalised shortly after the obligation has been invoked. So punctual obligations are actually achievement obligations with one deadline for fulfilling the obliged action and possibly with a second deadline for starting the obliged action. Different implementations have to be made depending on the expected duration of that said action as starting to fulfil an action and falling short of meeting the deadline is not the same as not even attempting to fulfil that action.

[3]This is looked in more depth in Section 4.4

3 | The Syntax of Lee's Formalism

Ronald M. Lee [5] presents a logic model for smart contracting that combines various types of logics such as deontic logic for the representation of contractual rights, obligations and prohibitions; temporal logic for time modelling; and operational logic for actions encapsulated in the contract.

3.1 Petri Nets. Graphical and Syntactical Representations

Lee [5] introduces the concept of a graphical way of visualising the logical content of electronic contracts. He frequently uses Petri nets as time is mostly relative in contracts and such sequential, concurrent and recurrent relationships can be easily represented through a place-transition graph in which decisions leading to the movement across states (places) can be represented as transitions. This project adopts the usage of Petri nets as they are an efficient way of visualising contracts. As they are used in most other papers [14, 19, 23] covered in Chapter 2, it can be said that Petri nets have become a de facto standard for representing smart contract formalism. "A Petri net combines the features of PERT diagrams and decision trees in representing sequence/concurrency and choice in a common formalism"[5].

3.1.1 Petri Net Graphs

A Petri net graph is a 3-tuple (S,T,W) [25] where:

1. S is a finite set of places (some of them being start or end places) - or"states" [5]
2. T is a finite set of transitions - also referred as "gates" in this paper due to the similarity in the way they function, by 'opening' in response to a condition being fulfilled
3. S and T are disjoint, meaning that no object can be both a place and a transition
4. W is a multiset of arcs $W : (SxT) \cup (TxS) \rightarrow \mathbb{N}$ where each arc can connect a place and a transition and gets assigned a non-negative integer arc multiplicity (or weight) - weights are many times omitted in Lee's Petri nets as well as in this paper

Petri nets are convenient for understanding the timing of events. Looking at the Petri net below and beginning at the "Start" state, there is a sequential progress until reaching the "Finish" state. Transition 1 causes concurrent succession to two states: S1 and S2, In order to progress, transition 2 or 3 has to be fired ("gate 2 or 3 has to open"), but this is dependent on the attached condition. For example, "X:A"

(read as "X does A") means that the party X has to complete the action A. Once X does A, the contract progresses to state S3 whilst being at the same time in S2. If Y does B as well, the contract progresses to S4 whilst remaining also in S3. Upon both S3 and S4 being reached, transition 4 is fired and the subsequent succession makes the contract reach "Finish". [1]

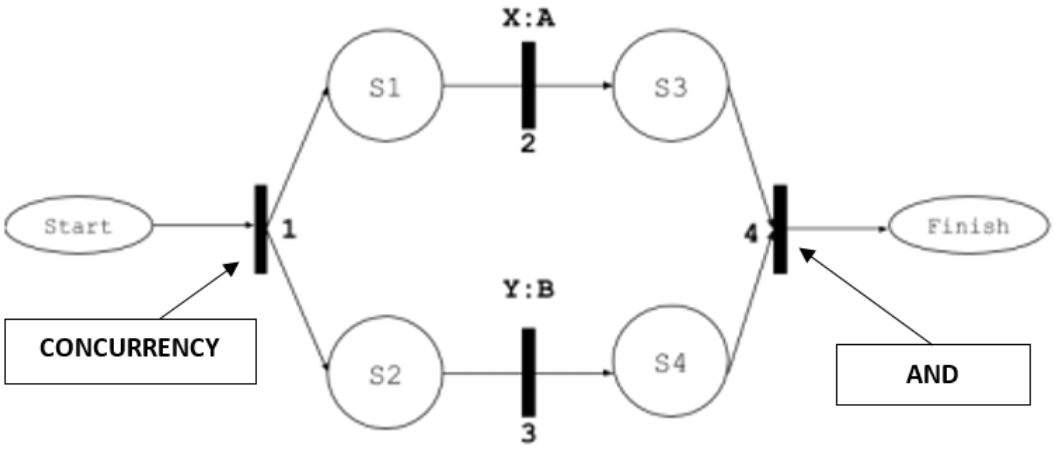


Figure 3.1: Example Contract in Petri Net Notation - Concurrency and And

3.1.2 T-Calculus - The Syntactical Representation of Petri Nets

As conventional proposition and predicate logics are not applicable to a dynamic aspect, Lee starts from Von Wright's T-Calculus model (1965) [26] and extends to allow the equational representation of a Petri net. The basic operator in Von Wright's approach is the logical connective T which is a type of asymmetrical conjunction and has the following form: $\Phi \quad T \quad \Theta$ which is read as Φ 'and next' Θ where Φ and Θ are propositions. This indicates that Θ is the state that follows Φ. [5, 26, 27]

Lee uses T calculus to give a logical representation to the state-transition diagram [5]. For the previous given Petri net, the corresponding T calculus expression is the following: $S_{\text{start}}T_1((S_1T_2S_3) \wedge (S_2T_3S_4))T_4S_{\text{finish}}$. He gives the following interpretation for Petri nets: state nodes correspond to elementary propositions or their negations; transition nodes correspond to T connectives; arcs branching out of a state node correspond to mutually exclusive disjunction; arcs branching out of a transition node correspond to a conjunction (concurrency).

[1] This example uses immediate transitions (represented as filled rectangles), differing from timed transitions (represented as empty rectangles), have no attached time, hence the attached actions have no deadlines. This example is, therefore, a theoretical one as, in contracting, all transitions must be timed except very rare exceptions (see also Section 4.4).

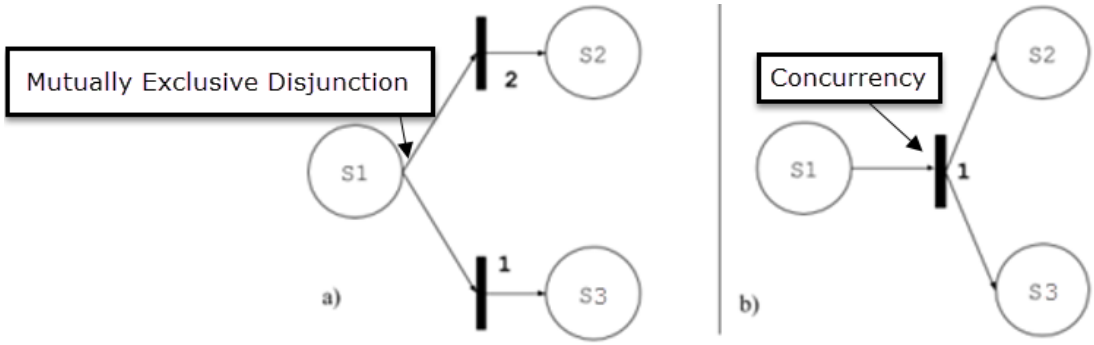


Figure 3.2: Petri Net Formalism: Mutually Exclusive Disjunction and Concurrency

3.2 The Trans-Assertion Notation

Lee [6] introduces the internal Petri net representation. Here, he extends the definition of the Petri nets and adds the concept of "trans-assertions" [2]. A trans-assertion is a more complex element containing two sets of states, the transition between them and the two sets of arcs connecting each element of each set with the transition. It will be seen in Section 4.2.1 that this concept makes a lot of sense as it usually forms the most basic type of deontic elements.

As an example, let there be a set of contractual obligations saying the following: "*A will pay B a sum of £10,000,000 by 30/08/2018. In exchange, B agrees to transfer Building BC to A on 10/09/2018.*" from a contract signed on the 01/08/2018. Using Lee's trans-assertions notation, the set of contractual obligations would look like this:

$$trans([s(Start, 01-Aug-18)], [s(1,\_1)], A : rb(30-Aug-18) : Pay(B, 10\quad 000\quad 000))$$

$$trans([s(1,\_1)], [s(2,\_2)], B : rb(10-Sep-18) : Transfer(Building\quad BC))$$

Lee associates a time of the event marked with an underscore followed by an identifier number (e.g. _1) to every state. Also, he associates an action (e.g. "Pay"), the deadline of action (e.g. "rb(30-Aug-18)" means "before 30/08/2018") and the author of the action (usually one of the parties) to every transition. The 3-tuple (Author, Time, Action) forms the condition for the transition to fire (e.g. for progress from state 1 to state 2 in the case in which the transfer of the building is completed in time). The 5-tuple (initial state, destination state, transition, arc from initial state to transition, arc from transition to destination state) forms the trans-assertion which, in this example, constitute obligations to pay the sum of money and to transfer the building. These obligations are however incomplete as there is nothing to specify what happens if any of the two parties do not fulfil them.

[2] Hence the notation is also known as the "trans-assertions" notation

4 | Critical Analysis of Lee's Model

As seen in Clack [11] and Hvitved [12] and as discussed in Chapter 2, there is a need for a semantic model that can encompass the non-operational aspects of a contract. The two main non-operational aspects of concern are those regarding the commitments of various parties and those relating to the time aspects (timespans, deadlines) in which operations and commitments are represented.

Lee [5] proposes a formalism and a semantic system which combines the deontic and temporal logics. Based on Hvitved's formalism comparison, Lee's semantic system encompasses most of the aspects required by contracts, does not neglect formal mathematical underpinnings and captures contracts appropriately. However, Lee's system does lack detail and lacks in terms of the definition of formal semantics and language. Further, this chapter will provide an in-depth analysis of Lee's semantic model and try to identify its shortcomings. Solutions for many of the identified issues are presented in Chapter 5.

4.1 Temporal Aspects

Contracts usually include a given order of actions. Sometimes, those actions might be sequential, whilst others may be concurrent. It is also not unusual in the financial sector that certain actions may be contingent on natural events, the performance of a third party or the occurrence of financial or economic events (e.g. The Bank of England raising interest rates or the pound sterling plummeting); as such, their timing is not absolute but relative. The lack of support for relative timing, as discussed later in this chapter, is one of the key issues with Lee's formalism. Another aspect to be considered is the initiation of penalties which can be incurred when not meeting a deadline. Furthermore, whilst some actions might have to be done at certain points in time (discrete time), other contractual clauses apply the concept of a continuous time, discussed later in this section. All these temporal relationships are key to the logic of a contract.

4.1.1 Rescher and Urquhart (RU) Calculus. Lee's RU Calculus Adaptation

As noted in Chapter 3, both T-calculus and graphical Petri nets can represent whether events happen subsequently or concurrently. They are, therefore, extremely useful in representing the relative temporal relationships contracts contain. However, modelling contracts often requires actions to be taken at fixed

points in time or within fixed deadlines. Lee calls these "absolute time frames" giving the Gregorian calendar as an example of such a time frame. He decides to makes use of the Rescher and Urquhart (RU) calculus [28], based on the logic operator R with R_t meaning "realised at time t".

Lee notes two issues about RU calculus: first, it assumes that components used to describe time are equal, which is not true in practice as months and years have various lengths; second, it can make use of discrete time, but cannot be used to represent continuous time. Furthermore, as discussed in Subsection 4.1.2, the RU calculus does not permit the formal representation of complex expressions, such as "action X should occur Y days after action Z has occurred".

Lee makes a few adaptations to the RU Calculus. First, he uses 01-January-0000 as the zero point with respect to which all absolute references are made. This is done for convenience and assumes contracts will not contain BC dates. Secondly, he also proposes that the day becomes the basic unit for time referencing as days are always the same length (24 hours) and it is also the most commonly used unit in business. Thirdly, to solve the issues of continuous time, Lee introduces the concept of time-spans and the notation span(D, D') meaning the period of time between D and D'. The 'realised at time' R_t operator is replaced by two new operators in order to fully capture the temporal needs of the contract:

- RD - "realised during". $RD_s\Phi$ means that Φ occurs at least once during timespan s.

$$RD_s\Phi \leftrightarrow (\exists \quad t \in s) \rightarrow R_t\Phi$$

- RT -"realised throughout". $RT_s\Phi$ means that Φ is true at every point in time during timespan s.

$$RT_s\Phi \leftrightarrow (\forall \quad t \in s) \rightarrow R_t\Phi$$

Besides these two new operators, Lee introduces two convenience operators:

- RB - "realised before". $RB_d\Phi$ means that Φ is true at least once during the timespan that starts "arbitrarily" and ends at date d. This simply means that Φ occurs at least once before the date d.

$$RB_d\Phi \leftrightarrow (\exists \quad t < d) \rightarrow R_t\Phi$$

$$where \quad "<" \quad means \quad "precedes"$$

- RW - "realised within". $RW_{i,d}\Phi$ means that Φ is true at least once during the timespan between a date d and another date that is the result of adding the interval i to the date d. This simply means that Φ has occurred within time i of the date d.

$$RW_{i,d}\Phi \leftrightarrow RD_{d,d+i}\Phi$$

4.1.2 The Issues with Lee's Adaptation of the RU Calculus

Further, there will be a discussion on the major temporal issues in Lee's and Pithadia's models [5, 27] and adaptations of the RU Calculus. Solutions for these issues are presented in Chapter 5.

4.1.2.1 The Basic Unit of Time

The first issue with Lee's adaptation is the fact that he standardises the unit of time as being the day. Whilst this might be sufficient for some contracts (e.g. a loan contract), there would be many contracts for which a day is too long to be the basic unit. An example for that is the UK Faster Payments System (FPS) [29], a payments-clearing scheme that reduces the payment times between different banks' customer accounts from the three working days with the older BACS system to just a two hours. [1]

An obvious solution might be replacing the day with the minute as the basic unit of time. However, some contracts refer to even smaller time units. Examples include web hosting companies that guarantee 99.99% uptime, financial institutions that manage online transactions and payments, DDNS routing providers that guarantee speeds at the precision of milliseconds. Simply looking for the smallest possible time unit and using that as the basic time unit for all contracts is not a good solution as encoding a contract that processes tasks within milliseconds in the same way as a contract where events would occur once every few years (e.g. bonds investment contracts). It is, therefore, necessary to look for a time reference mechanism that adapts to the type of contract and its requirements.

4.1.2.2 Recurrency and Relativeness of Time

Another issue is the way recurrency is handled by Lee's RU calculus adaptation. One clause in a contract might say that a payment has to be made each month of 2018. That clause enters into effect every single month in 2018: a form of finite recurrency. Lee [5] handles finite recurrency by using a separate trans-assertion for the each month's payment (e.g. Sum X to be paid in January. Sum X to be paid in February. ... Sum X to be paid in December.). This is similar to having a contract that has a separate identical clause for each month. If the contract is a multi-annual one, it is easy to imagine how complex the contract representation would become. An example contract between two banks for fast payments between customers' accounts such as FPS [29] would have to handle the following:

1. An unknown number of payments. This is unsupported by Lee's model.

[1] Other examples include derivatives swapping agreements in which processing has to be done once every few seconds.

2. A very high number of customers and of payments. If the contract has 100 clauses in regards to how each transaction is processed and a million payments will be processed over the length of the contract, the digital contract would need to have a hundred million clauses. This is not scalable.
3. Relative timing. Lee's model only supports dates and deadlines being defined from before the beginning of the contract. This would be impractical for contracts such as the ones that manage payments; the parties are unlikely to be aware at the signing of the contract when their customers will be initiating payments.

To solve these issues, a model could use the recurrency capabilities of the T calculus[2] and introduce relative time operators that adapt to the triggering of events rather than being defined in the contract.

4.1.2.3 Specific Time Elements

Another issue with Lee's adaptation of the RU calculus is the lack of support for specific time elements. For example, a payment processing contract might ask that payments made over the weekend to be processed on the Monday of next week. Loan contracts might ask that a repayment is made by the first Friday of each Month. A long-term insurance contract might ask that a payment is made by the end of the first week of April each year. Lee offers no support for the use of such time elements that can be quite frequent in some types of contracts.

4.1.2.4 RD and RB Operators

The RD (realised during) operator is one of the two fundamental operators in Lee's adaptation. Nonetheless, there is a range of temporal specifications that can't be represented using it. The operator only allows representing an action that takes place at least once during a certain time span which might not be sufficient in some cases. For example, there might be a need for an action to take place at least 9 times; exactly 7 times; at least 5 times but not more than 8 times; or it does not need to happen but if it does it can only happen 3 times. None of these examples can be represented using the RD operator. Either a more complex operator is needed or a standard procedure to represent those cases using the RD operator have to be adopted. [3]

Another issue, also identified by Pithadia [27], is the RB (realised before) operator which has an "arbitrary start date". This is confusing and it can create unexpected issues (e.g. an obligation might be

[2] Also supported by Petri nets
[3] Chapter 5 introduces the idea of templates which can be used to define such mechanism.

considered to be in place before the start of the contract). Pithadia replaces this arbitrary date with the zero start date in Lee's RU Calculus (01-January-00). This, however, is still not a good solution as it is not grounded in the semantics of the contract nor does it solve the issue of enforcing an obligation before the start of the contract (e.g. an action done before a regulation came into place). It might be attractive to use the start date of the contract as the base date for the RB operator. This means that the RB operator would not have a universal base date, but one dependent on each contract. This would essentially transform RB into an RD operator, eliminating, therefore, its very purpose. It is also worth noting that RB is problematic as it does not take into consideration the concepts of preemptive and non-preemptive obligations as presented in Section 2.2.3.4 and discussed in Section 4.4. It would be adequate, therefore, the completely remove this operator from the semantic system.

4.2 Deontic Aspects

4.2.1 Lee and Pithadia's models

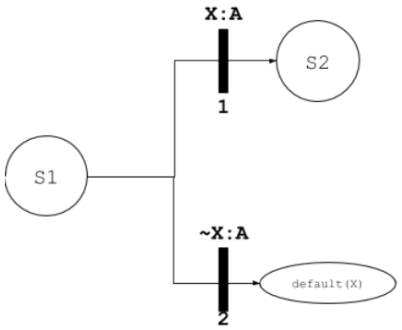
The purpose of a contract is to capture commitments made by parties. Whilst temporal aspects are important for placing the commitments into a time frame, it is also necessary to define and analyse the nature of such commitments. Lee [5] builds on Von Wright's work [13] and defines three types of commitments: obligations, rights (or permissions) and prohibitions. Lee bases his model on the one suggested by Anderson [30] which relates deontic logic to the alethic model logic using the definition:

$$O\Phi \leftrightarrow \Box(\neg\Phi \rightarrow S)$$

i.e. If one party is obliged to do Φ, then it is necessary that not doing Φ will lead to sanction S. In contracts, sanctions are sometimes stated explicitly in penalty clauses. If sanctions are not stated, they might be determined by means such as negotiation, arbitration or court suits. As many times there is such an expectation and, therefore, penalty clauses are omitted, Lee drops the necessity operator and adopts a more relaxed definition.

$$O\Phi \leftrightarrow (\neg\Phi \rightarrow S)$$

For the representation of commitments in electronic contracting, Lee bases his model on the idea that breach of commitments leads to sanctions and replaces the propositional constant S with the notation $default(X)$ where X is the party that defaulted on the contract. An action is therefore obligatory only if breaching the obligation (not doing the action) leads to a default state.



$$trans(S1, S2, X : A),$$

$$trans(S1, default(X), \neg X : A)$$

Figure 4.1: Definition of an obligation in Petri net representation and trans-assertion formalism; S1 and S2 are states, X is the party obliged and A is the obligatory action

Lee's model lacks any formal definitions for prohibitions and permissions, but, by building on Lee's model, Pithadia gives the following definitions:

Prohibitions :	*Permissions* :
$trans(S1, S2, \neg X : A),$	$trans(S1, S2, X : A),$
$trans(S2, default(X), X : A)$	$trans(S1, S2, \neg X : A)$

4.2.2 Issues with Lee's model and Pithadia's extensions

This subsection will be looking at the major deontic issues in Lee's model and Pithadia's extensions [5, 27] and adaptations of Von Wright's T-calculus [13] and of Anderson's work [30]. Solutions for some of these issues are presented in Chapter 5.

There are two obvious issues with Pithadia's extensions. First, the initial states for the pair of trans-assertions that define a prohibition are different. Similarly, the final state for the first trans-assertion in the same definition is identical with the initial state of the second.[4] Secondly, the final states for the pair of trans-assertions defining a permission are identical. This implies that the permitted action has no effect which is not necessarily true. Furthermore, it implies that the permission only holds between S1 and S2 and then gets discharged. This is unlikely as most permissions occur over a period of time, perhaps the entire contract, and allow the permission to be invoked many times.

4.2.2.1 Obligations and Prohibitions

Pithadia [27] states: "It is important to note that ... the deontic logic is "implemented" by means of expressing the obligation, permission or prohibition directly into the contract using a pair of Trans-

[4] This might, however, simply be a typing error in the definition that has been copied multiple times throughout the dissertation.

Assertion statements" [27]. This is based on Lee's definition of an obligation as being a pair of trans-assertions. Neither Lee nor Pithadia test this concept of paired trans-assertions which can be challenged by starting from the discussion on Linington et al. [19] in Subsection 2.2.3.3 which explains that obligations deal with expected behaviours whilst permissions and prohibitions deal with possible behaviours. This makes a considerable difference in the way we define these types of commitments.

On one side, an obligation requires X to engage in some behaviour Φ within a deadline and once Φ is observed to have been completed, the obligation is discharged. If Φ is not completed within the deadline, there is a default[5]. On the other side, a prohibition Θ on X is satisfied as long as no action which involves X doing Θ is observed. If such an observation is made, then there is a default, but if no observation is made, the state does not change. The prohibition might not be permanent and might be discharged after some while, but the prohibition is not discharged because X has not done the action Θ. In the case of an obligation, the state changes as soon as Φ is completed. Therefore, Pithadia's definition is defective as it has no time attached: $trans(S1, S2, \neg X : \Theta)$ means "If Θ is false even for a moment then the prohibition is discharged.".

Pithadia uses the definition with the meaning "If Θ remains false over the whole duration in which the prohibition is active (meaning the action Θ has not been done at any time during the whole duration in which the prohibition is active) then the prohibition is discharged." Also, Pithadia fails to take into consideration that, in electronic contracting, prohibitions tend to refer to continuous time, opposite to obligations which tend to refer to discrete time[6]. It is also worth noting that most prohibitions in a contract are usually permanent[7]. Of course, in some cases, prohibitions might only be in place for a limited amount of time and then they might be discharged. This means that the definition of a prohibition has to account for the fact that a prohibition might be composed of either one or two trans-assertions, depending on whether the prohibition is permanent or temporary.

The progress of a contract is never bound by the fulfilment of a prohibition, even though violating that prohibition might block that said progress. As such, it is important that in a Petri net representation of a contract, a prohibition should be on a concurrent branch. This makes sense as prohibitions are generally represented by a state and a transition waiting (maybe forever) to fire upon the prohibition

[5] In Lee and Pithadia's formalism, not fulfilling a commitment leads to default. However, defaulting on a high-value financial contract might not happen simply due to any small commitment not being respected so the definition needs to be relaxed.

[6] A discussion on this is made in Chapter 5.

[7] They can be triggered at the beginning of the contract or somewhere on the duration of the contract, but they tend to be permanent.

being violated. It would not be desired that the contract would pause there. Therefore, for a contract to be correct, every single prohibition should be asynchronous (in a concurrent Petri net branch).

An important point to be raised is that certain prohibitions remain in place after the apparent end of a contract. For example, an employment contract might specify that after the end of a person's employment in a certain company, that person is not allowed to work in a competing company for the following 5 years (known as restrictive covenants or post-termination restrictions [31, 32]). In this case, the clause(s) that represent that prohibition remain(s) in place even though the employment itself has been terminated. From a logical perspective, this means that, even though the activity within the company has terminated, the employment contract itself has not terminated and is simply in a different state - one where the employee has no obligations to work and the employer none to pay him. Other such covenants might include prohibiting the person leaving employment from ever disclosing certain information about activity in that company. Again, from a logical perspective, that means that the contract will only end when that person is no longer able to disclose such information, therefore making the contract permanent during the life of that person.

Another noticeable issue with Pithadia's prohibition definition is that many prohibitions tend to be non-exhaustive which means that even though a prohibition has been breached and a sanction has been triggered, that prohibition still remains in place. This concept is not supported by Pithadia as he immediately discharges a prohibition as soon as that prohibition has been violated.

4.2.2.2 Rights

The discussion on permissions (rights) is partially similar to the one on prohibitions, but it also adds some specific complexities as identified by Linington et al [19]. Two trans-assertions are not always needed as many contractual rights tend to be permanent in which case there is no need for a discharging trans-assertion. Also, rights tend to be non-exhaustive and, yet, Pithadia discharges rights as soon as those rights have been fulfilled. Furthermore, there is no time reference in Pithadia's trans-assertions. As such, the meaning of the second trans-assertion ($trans(S1, S2, \neg X : A)$) is "If Φ is false even once (i.e. if the permission Φ has not been fulfilled even for one moment) then the right is discharged." Obviously, this cannot be the intended meaning; permissions, just like obligations and prohibitions, only make sense when there is a time reference attached to them.

The permissions problem becomes more complex when looking at the observations made by Linington et al [19]. They explain that there is a difference between a right being executed (fulfilled or completed)

and a right being invoked. Invoking a right means that the party that is bound by the right has to facilitate, actively or passively, the fulfilment of that right by the party which holds the right.[8]

Therefore, a right might be invoked but never fulfilled. Pithadia does not include a sanctioning mechanism when it comes to rights. This is a mistake because it is important why that invoked right was not executed. An invoked right might fail for various reasons, but one of them is because that right was withheld by another party (usually by the party bound to facilitate the fulfilment of that right). Linington et al [19] talk about permissions as actually being an obligation for the other party to grant (fulfil) that right once the right is invoked. Therefore, if the right is withheld, the withholding party is breaking the permission commitment. This idea is impossible to be represented using Lee and Pithadia's model as they only take into account whether the right has been executed, but not if the right has been invoked. A more complex definition of permissions might be necessary in order to encompass this capability.

4.3 Isomorphism and Canonical Form

There are two problems with applying Lee and Pithadia's models in order to convert legal prose to formalised smart contracts or the other way around: lack of isomorphism and lack of canonical form. The issues arise from the fact that the model is not prescriptive and various concepts can be expressed in different ways. Isomorphism is necessary in order to enable lawyers to validate the semantics whilst a canonical form is necessary in analysis in order to identify similar contracts or clauses.

An example of expressing clauses in various ways through the semantic model can be found by looking at the classification of obligations made by Giordano et al [14] and discussed in Subsection 2.2.3.4. The maintenance obligations require a condition to be true at all times within a timespan[9]. This is a form of obligation based on continuous time, different from other types of obligations which rely on discrete time. The obligations that make use of continuous time (X is obliged to make sure Φ is true at all times) can also be represented as prohibitions (X is prohibited from doing $\neg\Phi$[10]). It can be easily seen how the difference between an obligation and a prohibition is ambiguous. This is a problem and it has already been noted in Subsection 4.2.2.1 that obligations and prohibitions might need different ways of being expressed (i.e. obligations needing two trans-assertions whilst a prohibition needs one or two). The problem of the semantic model not being formal enough is the very reason which gives birth to the previously mentioned issues.

[8] Executing a right means that the bounded party has facilitated the fulfilment of that right.
[9] E.g. A customer having to keep a minimum sum of money in his bank account at all times
[10] E.g. the customer is prohibited from taking out of the account more money than it is permitted.

The lack of canonical form refers to the fact that a legal contract can be represented as numerous different Petri nets.[11] The isomorphism problem, introduced by Clack [3], refers to the fact that the structure of the formal descriptions might be different than the structure of the legal text. E.g. a single legal clause might be represented by two trans-assertions that might be separated visually in a Petri net (and vice versa: two similar clauses in the legal text might be represented as a single trans-assertions).

Due to the isomorphism problem, even if a canonical form is given to the formalism, the generated legal prose might have different variants. Even if for a computer all versions might mean the same thing, that might not be true in a court of law. One, therefore, has to ask the question of whether it is wise to enable legal prose to be generated from a Petri net and whether that should be allowed in a court of law. With the current issues of the system, it is probably better not allow translation to legal prose. This will need further research and it is likely to end up being a decision taken by regulatory bodies.

4.4 Trans-Assertion Notation Issues. Conjunctions and Disjunctions

$$trans([A_1,...,A_m],[B_1,...,B_n],Event)$$

Lee uses various forms of trans-assertion notation. Initially, he defines the notation in the above way which means that if Event occurs, then there will be a transition from states $A_1 - A_m$ to the states $B_1 - B_n$. This definition is ambiguous as Lee does not explain if the contract has to be in all the states $A_1 - A_m$ in order to check for the event or if just one state is enough. Pithadia also makes use of trans-assertions with multiple initial states but does not formalise the meaning of the notation. In the case of Petri nets, Mansour [33] explains that a transition after a set of initial states has the meaning of synchronous joining (also known as "aggregation", "branch unification" or "conjunction")and it means that all states $A_1 - A_m$ must have been reached before the transition can be triggered. As this also seems to be the meaning intended in Pithadia's paper, this meaning is adopted herein. [12]

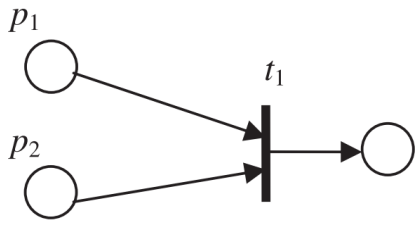


Figure 4.2: Petri Net Synchronization

[11] Just as two mathematical functions might give identical results for the same inputs, yet be defined differently internally.
[12] See also Figure 3.1 and Subsection 3.1.1.

A transition with multiple final states has the meaning of concurrency [33] (also knows as "transition conjunction" or "branch splitting"). [13] It is important to differentiate multiple arcs coming into a gate from multiple arcs coming into a state as that does not aggregate the branches [14]. However, having multiple arcs coming out of state is a valid operator called "state disjunction" by Lee or "mutually exclusive disjunction" by Mansour [33] meaning that after one of the successive gates is opened, no other successive gate can be opened [15]).

Lee's [ ] notation for describing a set has been replaced in Pithadia's paper by a nested bracket notation. This is harder to understand for humans and Lee's notation seems to be a better alternative. Nonetheless, Lee's notation suffers from the fact that each contractual component[16] have to be defined within the one line trans-assertion assignment. This means that the parameters list of each state repeats every single time it is referred which is unnecessary and makes the notation difficult to understand for humans. It is also prone to errors as parameters might get modified. A solution could be to define each state and their name at the beginning of the contract rather than while writing the trans-assertions as this would avoid carrying the list of parameters every time.

Lee also uses a timing variable for each state e.g. $s(1,T1)$. As the timing variable T1 is only used internally by the computer when processing and offers no additional information when inputting data, it might be a good idea to remove the variable from the trans-assertion notation, the formalism adjusted so that the time when a state is reached can be accessed in a different way.

One last thing to note about Lee's trans-assertion notation is that time is not always specified which is an issue as it is not clear what no time reference means. Naturally, one can expect that it would mean either the entire duration of the contract or the timespan between the date the commitment has been activated and the end of the contract. The first case might involve that the timespan for a commitment starts before that commitment was triggered in which case obligation is said to be preemptive; otherwise, it is non-preemptive (discussed in Subsection 2.2.3.4). It is difficult to associate a similar meaning to permissions as a 'preemptive permission' would not make much sense. Considering this ambiguity and as discussed in the previous sections, there is a strong case for imposing a semantic rule such that each trans-assertion is to specify its time reference with the exception of trans-assertions which are specifically constructed for aggregation or concurrency and have no event or action attached to them.

[13] Having multiple initial states and multiple final states in one trans-assertions is also possible and has the meaning that it aggregates the incoming branches and then it starts new concurrent branches.

[14] In fact, multiple arcs coming into a state does not change the progress of the branches in any way as it is not an operator

[15] Except if the state is reached again in the case it is part of a cycle

[16] State, transitions, time reference, time-spans and actions the fundamental components in Lee's formalism

5 | Design and Implementation of Revised Semantic Model

This chapter provides solutions for many of the issues identified in Chapter 4 in order to obtain a formal semantic model for semantic analysis of smart contracts that meets the goals set out in Chapter 1. The issues tacked are temporal aspects (e.g. absolute and relative timing, discrete and continuous time-spans) deontic definitions and formalism, the introductions of templates and system parameters.

5.1 Deontic and Temporal Aspects

As seen in Chapter 4, there are two sets of issues with the way Lee's model encompasses temporal aspects. The first one is related to time references and the second one to time-spans. When referring to the deontic aspect of Lee's model, one noticed a lack of formalism of the types of commitments[1] Lee makes use of which is caused by the lack of formalism of time aspect.

5.1.1 Time References and Operators

As shown in Subsection 4.1.2.2, Lee's model has a major issue which makes it impossible to be used in many financial contracts: Lee's model is absolute time-based. Moreover, the absolute time model is deficient as it uses the day as basic time unit (see Subsection 4.1.2.1), and does not support specific time elements[2] (see Subsection 4.1.2.3). To solve these issues, the project introduces a new absolute time format and the following concepts: absolute, relative and recurrent time references and time operators.

An absolute time reference describes an absolute point in time (e.g. 01/07/2018) and is the only time reference used by Lee. The following notation will be introduced for describing an absolute time reference: $Year-Month-DayatHour:Minute:Second.Milisecond$. [3] [4] Using all the parameters is not necessary and missing a parameter will default that parameter to its minimum value (i.e. 1 for days and months and 0 for all the rest). [5] The year must always be specified. Further examples of this syntax

[1] Obligations, rights, prohibitions
[2] E.g. The first Monday after an event
[3] This format can easily be extended to support nanoseconds if need be, specifications of time zones etc.
[4] The months can be expressed either in words or in numbers.
[5] E.g. $2018-Jan-01$ is equivalent with 2018 as well as with $2018-01-01 \quad at \quad 00:00:00.000$.

being used in practice can be seen in Appendix A.3. This solves two issues: no longer needing to have a basic time unit [6] and no longer having the day as the smallest time unit.

Relative time is the second time reference type being introduced. In a contract many events happen in response to others, hence the use of Petri-net diagrams, which are relative by nature, for visually representing contracts. However, Lee's model uses the relative property of the diagrams in a purely conditional fashion (e.g. if action A is done, then do action B). As such, time references for relative actions are impossible to be accurately represented.[7] Adding the concept of relative time is solving this issue, but absolute time references are not adequate for representing relative time; as such, one more concept is introduced: time operators.

To enable the use of the relative time references, one can make use of two relative time operators: After (AFT) and Before (BFR). Given an absolute time reference and a time period called "differential time period", these operators can be used to represent relative time references. This might sound no different from Lee's RB operator. In practice, the BFR operator can be used in conjunction with pre-determined time[8][9] which would provide a similar result as the RB operator. However, the operator's real strength is when used with non-pre-determined time. For example, AFT can be used in conjunction with a state S2 and differential time period of 2 hours with the result being 2 hours after the state S2 was reached. The functioning of this conjuncture is based on a notion introduced by Lee [5] who says that each state registers the time it was reached.[10] Here a modification is added to the property: each state registers the *last* time it was reached. This is important to clarify as a state can be reached multiple times as part of a cycle. By remembering the time a certain state was reached, the operations introduced can now be used in order to enable the use of relative timing.

There are still contractual formats that cannot be represented even with these operators. For example, let there be a contract for short-term credits which offers 0% interest rate if the credit is repaid by the end of the month or within 14 days of the credit being offered to the customer, whichever is the earliest. Formats such as 'whichever is the earliest' or 'whichever is the latest' are impossible to represent with the current format. As such, two additional relative time operators are added: Earliest (EST) and Latest (LST) which take two time references and calculate the earliest or the latest, as per case.

[6] It would be impractical to have to specify the number of milliseconds each time

[7] E.g. processing a payment within 2 hours after the payment was initiated is impossible if the time of the payment initiation is not known at the time the contract is signed

[8] Pre-determined time refers to any absolute time references that have been established at the start of a contract; e.g. the end date of a contract is a predetermined absolute time reference.

[9] E.g. using BFR with the date 10/05/2020 and a period of 2 days, it results in 8/05/2020

[10] Lee never makes use of this property in the formalism.

The third type of time reference is recurrent time. If a certain payment has to be made each month for 7 years, it would be impractical to have 70 different clauses for each time reference. A time reference that can encompass multiple absolute time references is needed: the Schedule operator (SCH). SCH is built on the idea of a well known Unix/Linux tool called Crontab [34, 35] which is used to schedule various actions at recurrent times [11]. SCH is building on this tool and extending it, adding support for more precision through the introduction of years, seconds and milliseconds. This operator solves two issues: lack of reccurency and lack of support for specific time references.

5.1.2 Discrete and Continuous Time-Spans

In Section 4.3 discusses how Giordano et al's maintenance obligation [14] can also be represented as a prohibition. The analysis of obligations from a temporal perspective identifies two main types of commitment: achievement obligations which are fulfilled once an action has been taken (within a deadline and maybe more than one time) and maintenance obligations which require a condition to be true during all instants within a timespan. On the other side, when talking about semantic formalism, Linington et al. [19] describe obligations as requiring a party to "engage in some piece of behaviour, involving participating in particular action roles, and is discharged if the required behaviour is observed" [19]. They also define a prohibition as requiring a party not to engage in some behaviour and that the prohibition is satisfied as long as the prohibited action is not observed.

Lee's model suffers from an important shortcoming. Though Lee uses the concept of discrete and continuous time-spans through R-operators [12] [13] [14], he does not formalise the definitions of these concepts. This subsection will define them in order to enable their usage in the formalism being brought forward.

A discrete time-span is a finite and countable totally-ordered set of distinct time units [15] through which a time variable jumps from one unit to the next and any measurement is done a finite number of times between two units.

A continuous time-span is an infinite and uncountable totally-ordered set of time units where between any two units there is an infinite number of units and in which a variable has a certain value for an unmeasurably small amount of time.

[11] E.g. data backups every Tuesday at midday or each 13th day of every February and November
[12] Specifically RD for discrete time and RT for continuous time
[13] From now on referred to as time-span operators
[14] Not to be confused with time operators such as LST or SCH
[15] E.g. weeks, days, hours, milliseconds

An achievement obligation makes use of discrete time as it requires an active behaviour and an action taken. Once the action is observed, the obligation is considered executed and discharged. A maintenance obligation is different as X is obliged to make sure Φ is true at all times [16], no active behaviour and no action is needed. Continuous time is, however, not computable: a computer can check if Φ has become false very often, maybe every nanosecond, but mathematically and according to the formalism introduced that is still not continuous as that implies that between every two consecutive time units, there is an infinite amount of time units [17]. The solution for this problem is that in the case maintenance obligations, there should not be a continuous checking of Φ, but waiting for $\neg\Phi$ to become true and trigger an event when that happens; in other words, waiting for an event that is prohibited. As such, an obligation in the legal prose is implemented, in Computer Science, as a prohibition.

5.1.3 Deontic Definitions

Based on the observations previously made in this chapter, in Chapter 4 and on Linington et al.'s definitions [19], the following formalism for obligations and prohibitions is defined:

Obligations. An obligation is a pair of trans-assertions representing a commitment that requires an action to be executed by a party a finite number of times in a given discrete time-span. [18]

$$\begin{aligned} &trans([A_1,...,A_m],[B_1],E:T),\\ &trans([A_1,...,A_m],[B_2],\neg E:T) \end{aligned} \tag{5.1}$$

where $[A_1,...,A_m]$ is the list of initial states, B_1 is the final state in case of obligation fulfilment and B_2 in case of obligation failure; E is the event of party X completing action A. The mandatory timespan T solves the problem of preemptive and non-preemptive obligations. It should be noted that an obligation may have multiple initial states meaning that the obligation is only enforced if all states are reached.

Prohibitions. A prohibition is a singular trans-assertion representing a commitment that forbids an action from being taken at any time during a continuous time-span that starts with the invocation of the prohibition and ends either at the end of the legal agreement or earlier if discharged as such.

$$trans([A_1,...,A_m],[B_1],E:T),$$

Prohibition discharging mechanism:

$$trans([A_1,...,A_m],[B_2],\neg E:T)$$

[16] Therefore, it makes use of continuous time

[17] This is actually why, in Computer Science, there is no such thing as real continuous time, just approximate representations

[18] Lee was making use of default(X) for the second branch. However, breaking a simple obligation does not necessarily lead to a default on the whole contract (e.g. just issuing a warning).

Once B_1 has been reached, the prohibition is no longer enforced. If the prohibition is meant to remain in place, this becomes a problem and the solution is the introduction of templates later in the chapter.

As prohibitions might never be violated or it might take a very long time to be violated, it is wrong to implement prohibitions in succession to other mechanisms (e.g. obligations). Due to the very nature of prohibitions, they should be implemented concurrently via the conjunction gate operator as shown in Section 3.1.1.

Permissions . A permission is a group of three trans-assertions: one represents the right of a party to invoke a permission and the remaining two represent a commitment from the other party to facilitate the fulfilment of the right; the group can be paired with a permission-release mechanism which discharges the permission at the end of a given continuous timespan.

$$\begin{aligned} &trans([A_1,...,A_m],[B_1],E:T),\\ &trans([B_1],[C_1],F:W),\\ &trans([B_1],[C_2],\neg F:W) \end{aligned} \tag{5.2}$$

Permission discharging mechanism:

$$trans([A_1,...,A_m],[B_2],\neg E:T)$$

where $[A_1,...,A_m]$ is the list of initial states, B_1 is the state meaning the permission has been invoked and B_2 meaning it was discharged due to not being invoked in the given continuous timespan; C_1 is reached when the invoked right is fulfilled and C_2 if it is not fulfilled; E is the invoke action event and F is the fullfill action event; the continuous timespan T represents the time in which the right is available and W is the discrete timespan in which the invoked right has to be fulfilled.

The new permission definition solves the issues raised in Subsection 4.2.2.2 regarding the fact that the old definition was ignoring the invocation of rights. The difference is caused by the fact that a permission for a party to do something raises an obligation for some other party to allow and/or facilitate the fulfilment of the permitted action.

Through the formats, notions, definitions and concepts introduced or adjusted in this section, every single temporal issue identified earlier in the paper as well as a number of deontic issues have been solved. However, not only it is necessary to define the syntax of each one of the time and time-span operators introduced, but also to redefine most of the formalism in order for it to support these additions.

5.2 System Parameters

In Lee's formalism, real-world actions can lead to the opening of gates and progress within a contract.[19] However, in a legal agreement, not all events are triggered by a party completing a real-world action. For example, let's say that the insurance agreement allows a party to make 5 insurance claims per year and cover a total of £500,000. Making an insurance claim is a physical action that takes the contract into a new state, but reaching the limit of insurance claims or the total cover of the policy is not caused by a physical action, but rather by a series of actions. This is an issue as with Lee's formalism there is no way to represent the contractual clauses that limit the number of claims to 5 per year or that provide a limited total cover. [20] There is a need for a formalism that enables an internal system counter that tracks what and how many actions have taken place, but that can also allow triggering gates when that counter reaches a certain value. Here comes in the system parameter.

The system parameter is one of the most important notions the project brings forwards. Its purpose is to tackle three types of issues: counter related (such as those mentioned previously), time-related (certain actions leading to a deadline moving forwards or backwards) and text related (such as a name changing; to be explored in detail in a subsequent paper). As such we define three types of parameters: numeric (can be used as integers or as floating point numbers), text (contain letters, numbers and any other type of characters), time (store a time reference [21]).

There is a need for a formalism that allows for storage of these parameters as well as changing and checking their value. As such, two mechanisms which will be formalised in this chapter: an event mechanism which tracks the value of a system parameter and triggers a transition of states when the parameter reaches a certain given value; and an update mechanism which applies various operations on the system parameter every time a transition is fired. [22]

[19] For example, in a legal agreement for insurance services, paying the monthly price of the insurance service an action that triggers a gate when completed and leads the contracts into a new state.

[20] Other examples can include a contract that requires monthly payments and three missed payments result in a default or a contract that enforces an action to be taken exactly 7 times.

[21] Work in the same way as discrete time components - presented later in the chapter - with the difference that they can update their values dynamically

[22] These system update operations will differ depending on the type of the system parameter. Operations will be introduced later in this chapter.

5.3 Contract Components Formalism

In Von Wright's T-calculus [26] discussed in Chapters 3 and 4, each variable has an assigned value before each equation is written. This reflects how in a graphical Petri net, each component is independent (e.g. the definition of a state is not dependent on the definition of a gate or of any arcs). In Lee's trans-assertion notation, all the components[23] are defined simultaneously within one very long expression. This is not just more difficult for humans to read, but it creates its own issues as discussed in Section 4.4: carrying the parameter list with every use, a single component might be defined multiple times[24] [25], it does not scale well with large contracts and it has no support for additional components or safety flags (more on that later in this chapter). Furthermore, Lee's formalism is incompatible with the two very important concepts that this dissertation brings forward - templates and system parameters - but also with some of the concepts introduced such as time-references, time and time-span operators.

Therefore, there is a need for a notation where each component is defined independently of others and then is easily accessible when needed [26] through an id [27], which is a short string formed from the type of the component followed by a unique number (e.g. 'STATE7' [28] can be the id of a state; abbreviations such as 'S7' are also allowed. Components will also have a name and various parameters that define their behaviour (e.g. states have a parameter that defines their type). [29] Examples of the syntax in use are provided in Appendix A.1 and formal definition of the context-free grammar of the syntax in Backus normal form (BNF) is provided in Appendix A.2.

Contract Component

$$C(name, start\quad date, end\quad date) \tag{5.3}$$

A contract component (C) is the component that represents the contract; it also gives a name to the contract and it defines the start and end dates. If the contract has no end date, (e.g. a non-disclosure

[23] In Lee's formalism states, gates, time-spans, dates are the components

[24] This also leads to possible errors in having the definitions differ between one another

[25] Re-defining the start date of a contract every time is needed would be impractical

[26] Just like in a computer program where the value of variables can be accessed any number of times by simply using the name of the variable; or in a typical legal contract where for example dates are mentioned only once and various terms describing actions are defined in a special section of the contract.

[27] Template definitions are an exception from this rule as they have to be accessed by their name and do not have an id.

[28] Not case sensitive; 'State7' or 'state7' are also valid ids

[29] The proposed formalism might sound very familiar if the reader is used to object-oriented programming as the contract component syntax is similar to that of a discrete class or, in the case of template definitions, similar with that of an abstract class.

agreement), the final parameter is 'N/A'. Each contract has exactly one contract component.

States

$$S(id, name, type) \tag{5.4}$$

The states (S), also known as places, and represent various stages of a contract. There are different types of states: start, end, default, pause and intermediary. In Lee's formalism, in most cases, there was no need to mention the type. Looking at a Petri net would usually give a clear indication of whether a node is a start node or another type. The specification of the type, besides making the formalism easier to read, works as a safety flag. When computed, this allows checking if each state respects various properties and, as such, it allows to validate the correct construction of a contract[30]. The following five states and their properties are defined.:

- START - the initial start state of a contract
  1. Each contract has exactly one start state.
  2. A contract should only reach the start state once. In other words, such a state only has outbound arcs. As such, a start state can never be part of a Petri-net cycle.
- END - one of the final states of a contract; it terminates the contract
  1. Each contract has at least one end state.
  2. The effect of reaching an end state is global: the entire contract ends immediately
  3. An end state has only inbound arcs and can never be part of a Petri-net cycle.
- DEFAULT - a state corresponding to a legal default of a party on the contract
  1. The effect of reaching a default state is global: it applies to the entire contract
- PAUSE - a state where the contract branch pauses to wait for some external human input (e.g. adding or removing clauses from the contract, modifying various parameters etc.)[31]
- INTERMEDIARY - any other state that does not have one of the previously mentioned types
  1. A start state must always be succeeded by an intermediary state. An end, default or pause state must always be preceded by an intermediary state.
  2. An intermediary state must be preceded by another state, but it does not need to be succeeded by another state as it can end a branch of the contract with no effect on the rest of the contract.

[30] E.g. A contract cannot start with a default state

[31] The pausing mechanism should be analysed and formalised in a following paper

Besides these individual properties, a property discussed in Subsection 5.1.1 says that each state registers the time it was reached last time (this property will be further referred to as 'time encapsulation'). This is important as it allowed to introduce operations and relative timing.

Parties

$$P(id, name) \tag{5.5}$$

Parties (P) are a new component introduced used for keeping track of the contractual parties, as defined in a typical legal contract, as well as various third parties.

Discrete Times

Discrete times (DT) are components that track time references. A time reference can be any of the following: a specific date (e.g. '2018-12-03'), a discrete time component, a time operator, a state (due to the time encapsulation property) or an event (events have the same time encapsulation property as states; [32]). Discrete time components have an id, a name and a time reference parameter:

$$DT(id, name, time \quad reference) \tag{5.6}$$

The purpose of the discrete time component is to eliminate the need for defining a time reference every time it needs to be used. This is not just to make the writing and reading of the contract easier, but also to eliminate errors that might arise due to the need of repeatedly redefining a time reference. Also, it reflects the structure of a legal contract where all absolute dates are be defined in a special section of the agreement (e.g. $DT(DT1, `Payment \quad deadline', 2018-09-13 \quad at \quad 08:30)$ [33]).

After introducing the time operators and the state time encapsulation notion, there is no limitation to absolute time. If after initiating a payment, the contract reaches a state S5 and the payment has to be processed within 2 hours of the initiation, this can be expressed as such: $DT(DT2, `Payment \quad processing \quad deadline', AFT(S5, 0, 0, 0, 2))$ [34] These concepts solve two major issues: they improve the use of relative and recurrent timing and eliminate the scalability problem.

The scalability problem is discussed in Chapter 4 and results from the fact that, in Lee's formalism,

[32] they will be defined later in this section

[33] Further examples of this syntax being used in practice can be seen in Appendix A.3

[34] With the meaning 0 years, 0 months, 0 days and 2 hours after state S5 was reached; accessible by id DT2. The full syntax of time operators will be detailed further in this section.

the contract has to grow in line with the number of events in the context of a contract. By using the previous example, in order to define the 2 hours deadline for 1 million transactions, exactly 1 million time references would have been needed, one for each transaction. This newer model only requires one component: a discrete time. This is because the discrete time component is, like all other components, reusable and it recalculates the time reference needed each time is accessed. This means that the discrete time component can generate all the 1 million time references without the need to have them mentioned in part in the legal contract.

In Section 5.1.1, four relative time operators (AFT, BFR, EST, LST) and one recurrent time operator (SCH) have been introduced. Further, the syntax for these operators will be defined.

The Relative Time Operators AFT, BFR, EST, LST

$$AFT(reference \quad date, years, *months, *days, *hours, *minutes, *seconds, *milliseconds)$$

$$BFR(reference \quad date, years, *months, *days, *hours, *minutes, *seconds, *milliseconds)$$

$$EST(time-reference_1, time-reference_2)$$

$$LST(time-reference_1, time-reference_2)$$

where the '$*$' symbol has the meaning that the parameter is optional.

The AFT and BFR relative operators simply take a time reference which is the reference date for the operator, and a list of parameters used to specify the differential time period - the number of years, months etc that the new date is after or before the reference date. Similar to how date formats work, not all the parameters have to specified, but the year must always be specified. The EST and LST take two time references and calculate the succession of the two. EST becomes equivalent with the earliest time reference and LST with the latest.

The Recurrent Time Operator SCH

$$SCH(\{Y_B, Y_E, Y_S\}, *\{M_B, M_E, M_S\}, *\{DM_B, DM_E, DM_S\}, *\{DW_B, DW_E, DW_S\},$$
$$*\{H_B, H_E, H_S\}, *\{MIN_B, MIN_E, MIN_S\}, *\{S_B, S_E, S_S\}, *\{MLS_B, MLS_E, MLS_S\})$$
$$SCH([Y_1, Y_2, ..., Y_N], *[M_1, M_2, ..., M_N], *[DM_1, DM_2, ..., DM_N], *[DW_1, DW_2, ..., DW_N],$$
$$*[H_1, H_2, ..., H_N], *[MIN_1, MIN_2, ..., MIN_N], *[S_1, S_2, ..., S_N], *[MLS_1, MLS_2, ..., MLS_N])$$

where the '$*$' symbol has the meaning that the parameter is optional.

The referential operator SCH takes a list of 8 parameters describing the following: years, months, days

of the month, days of the week, hours, minutes, seconds, milliseconds. Only the year parameter is mandatory, all other being optional. The parameters are of three types: range, list and '$*$'.

The last parameter is used to describe the time elements for which the operator applies. For example $SCH([2018,2020,2027],[May,July])$ has the following meaning: 'each month of May and July of the years 2018, 2020 and 2027'. The type of the parameter is marked by the use of square brackets.

The $''*''$ has the same use as in the original Unix Contrab tool and it means 'each'. For example $SCH([2018,2022],*)$ has the following meaning: 'every month of the years 2018 and 2020'. It is vital to differentiate this from $SCH([2018,2022])$ which means 'once per year in 2018 and 2020' and from $SCH([2018,2022],*,*)$ which means 'every day of every month of the years 2018 and 2020'.

The range parameter is a 3-tuple of parameters marked by braces with the following meaning: $\{begin-range,end-range,step\}$. This is used to define a range which can generally be used as a much shorter version compared to defining a list. For example, if a mortgage contract lasts for 25 years, from 2018 to 2042, one would have the type each year using list format. However, in a legal agreement, a clause would likely say 'each year from 2018 to 2042'; that can be written as such: $SCH(2018,2042,1)$ where 1 represents the step by which the value increases. Similarly, if a contract would say 'every three months, each year from 2018 to 2042', the operator would look like this $SCH(2018,2042,1,Jan,Dec,3)$[35]

Possible Issues. It should be noted that this formalism is susceptible to two recursion problems: interdependence and self-dependence. Interdependence issues are caused by two or more discrete time operators referencing each other - e.g. there can be $DT(DT1,'DT1',AFT(DT2,1))$ and $DT(DT2,'DT2',DT1))$. This is similar to how in computer programming there would be two functions and each function needs the results of the other in order to calculate its result, as such creating what is knows as an 'infinite cycle' or 'infinite recursion'. Self-dependence is similar with the difference that the infinite recursion is caused by a discrete time referencing itself - e.g. $DT(DT3,'DT3',DT3)$. These issues are relatively easy to identify before the start of the contract. This is different than how in programming, such issues could only be identified at run-time.

Another problem with this formalism is the undefined reference issue. This is less obvious than the recursion problems. Such an example is generated when a discrete time component is dependent on

[35]Not to be confused with 'every third month from each year from 2018 to 2042' which would be written like this $SCH(2018,2042,1,[Mar])$ or simply $SCH(2018,2042,1,[3])$

a state, e.g. $DT(DT4, `DT4', AFT(S1,2))$, and it is referenced before that state is reached. As such, the time encapsulation property cannot be applied. This is similar to how in programming, there would be a reference for a variable that was defined but not yet initialled. However, in programming, this issue is quite easily identifiable, except for the cases in which that variable is meant to be defined as well at run-time. The time encapsulated by states can be defined only after the contract has started, at 'run-time', so the issue is more difficult to identify beforehand.

Time-Spans and Time-Span Operators

Building on the observations and suggestions made in Chapter 4, a property is introduced: all rights, obligations and prohibitions are enforced for a certain time-span. This differs from Lee's formalism where the mentioned deontic elements can be enforced by simply having a start date or an end date or neither. One of the issues raised was the idea of preemptiveness - when an obligation without a start date is invoked, does that mean that the obligation was enforced starting with the time of the invocation or the time at which the contract began? The issue is solved by placing all deontic elements in time-spans.

The time-span component (TS) does exactly that by allowing to define time spans using two time references and a time-span operator[36]. The time-span operators are the same basic R-operators introduced by Lee: realised-during (RD) which describes a discrete time span and realised-throughout (RT) which describes a continuous time span. There is, however, a major difference between the newly introduced time-span operators and Lee's. This comes from the fact that we have introduced time operators and time encapsulation property for states and events. As such, time-span operators are no longer limited to absolute time but can encapsulate relative and recurrent time. The syntax for time-spans and time-span operators is the following:

$$TS(id, name, \quad time-span \quad reference)$$
$$RD(starting \quad time \quad reference, \quad ending \quad time \quad reference)$$
$$RT(starting \quad time \quad reference, \quad ending \quad time \quad reference)$$

Actions

$$A(id, name, *parameter_1, \ldots, *parameter_N) \tag{5.7}$$

The action component is used to describe a real-world action. For example, in a loan borrowing agree-

[36] Not to be confused with time operators

ment, repaying the credit is an action. An action component has, besides its id and name, a list of components that can be of any length needed. It should be noted that actions in themselves are not attached to any party and can occur more than once (e.g. monthly payments). This adds scalability to the formalism design as the action might have to be performed by multiple parties in the contract and might need to occur a high number of times.

Events

Events are one of the important new components that are being introduced with this new formalism. They are of two types: discrete events (E) and system events (SYS-E).

Discrete events are used to bind parties to actions. For example, in a loan contract, repaying an instalment is an action, but a certain person paying is a discrete event triggered by a party.[37].

$$E(id, name, party, action) \tag{5.8}$$

As shown in Section 5.2, in a legal agreement not all events are triggered by a party completing a real-world action. As such introduced the notion of system parameters that count actions and track how they are completed. To enable the system parameters, the system update and system events mechanisms are needed. The system events component is built exactly with the purpose of enabling the latter mechanism. The syntax for system events is the following:

$$SYS-E(id, name, systemparameter, targetvalue) \tag{5.9}$$

Similar to how discrete events are triggered by a party completing an action, system events are triggered by the given system parameter reaching the given target value. This behaviour depends on the type of parameter. For numeric parameters, the event will trigger when it is equal or greater than the target numerical value. For time parameters, the event will trigger when the current time is the same as the target time. For text parameters, being given a target string of length N, the event will trigger when the first N characters of the parameter are identical with the target string.

It should be noted that events have the same time encapsulation property as states, therefore 'remembering' the last time they were triggered. This enables the use of time-span operators with events, just like with states (e.g. $RD(DT1, E1)$ or $AFT(E2, 0, 3)$). It should also be noted that events can be triggered

[37] Similarly, the Bank of England rising interest rates is a discrete event triggered by a non-contractual party

any number of times, therefore enabling reccurency[38].

System Parameters

$$SYS-P(id, name, initial \quad value) \tag{5.10}$$

The system parameter component (SYS-P) is used for the storage of the previously introduced system parameters. System parameters can be of three types: numeric, time and text. However, there is no parameter that determines the type of the parameters. This is because the type is determined by the initial value. An initial value should always be used in order to define the type of parameter, even if the value itself does not need to be used. System parameters have a fixed type in order to prevent non-valid operations from being applied to them (e.g. applying a text operation on a numeric parameter).

System Updates

$$SYS-U(id, name, parameter, operation) \tag{5.11}$$

The system update component (SYS-U) enables the updating of system parameter values by applying various operations on them. The defined system update operations for numeric parameters are the following: addition (ADD), subtraction (SUB), multiplication (MUL), division (DIV). Each one of these operations takes one parameter and applies the relevant mathematical operation to its value [39]. E.g. Let there $SYS-P_1$ with value 10 on which we apply $SYS-U(SYS-U_1, `add \quad 7', SYS-P_1, ADD(7))$; the operation will add 7 to the value of the parameter, the result being 17.

For text parameters, the operators are addition (ADD) and remove last characters (RMV_LST). ADD takes a string and adds it at the end of the text parameter. RMV_LST takes a number N and removes the last N characters from the text parameter.For time parameters, the operations are addition (ADD) and subtraction (SUB). They work exactly like the time operators AFT and BFR - they take a list of parameters (years to milliseconds) which represent the differential time period and add or subtract that time period to the value of the time parameter.

There is one last operation which can be used on either of the three types of system parameters and has the same effect: attribute (ATR). ATR is used when one wants to attribute a new value to the system

[38] Represented in a Petri-net as a cycle
[39] See full syntax in the BNF at Appendix A.2

parameter. This can be done in two ways: either by giving a parameter value (e.g. $ATR(2018-July)$) or by giving another system parameter (e.g. $ATR(SYS-P_2)$) in which case just the value of the other parameter is copied, but the parameters continue to act as different, independent parameters [40]. It should be noted that operations can not change the type of the parameters, not even the attribute operation.[41]

Gates

$$G(id, name, sign, *event, *time-span, *system \quad update) \tag{5.12}$$

Gates (G), also known as transitions, keep their semantic meaning as in Lee's formalism and in the Petri net concept, but with a different formalism and syntax. Gates are reusable: once they open and the state changes, they become closed again and can be reused; otherwise, the gate would remain closed the second time which would make recursion impossible (see example below). A system update can be optionally attached and runs each time the gate opens. There are two types of gates: immediate and timed.

$G(G_1, `the \quad positive \quad gate', True, E_1, TS_1)$

$G(G_2, `the \quad negative \quad gate', False, E_1, TS_1)$

$TA([S_1], [S_1, S_3], G_1)$

$TA([S_1], [S_2], G_2)$

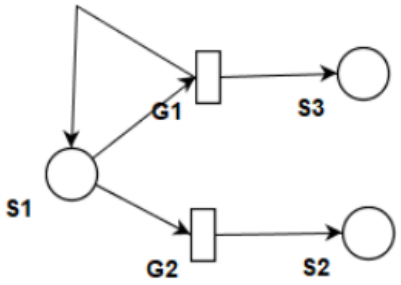


Figure 5.1: Cycle. TA is Lee's trans-assertion (to be formalised later in this section) that connects S_1 to itself and to S_3 via positive gate G_1 and to S_2 via negative gate G_2

Immediate gates are the gates that do not have an event or a time-span attached and are, in their turn, of two types: the immediate positive gate (e.g. $G(G_3, `immediate \quad positive \quad gate', True)$) which is always open and the immediate false gate (e.g. $G(G_4, `immediate \quad positive \quad gate', False)$) which can never be opened. The immediate positive gate is useful as it allows to attach system updates operations to it when there are more than one operations to run in response to an event[42]. Also, immediate positive gates are used many times to aggregate concurrent branches into one single branch (e.g. $TA([S_4, S_5], [S_6], G_3)$) or to split the branches into concurrent ones (e.g. $TA([S_7], [S_8, S_9], G_5)$).

[40] i.e. It is not a referential copy of the parameter as it is possible in computer programming

[41] Applying an operation of another type than the parameter results in an invalid format.

[42] As a gate only allows one system update, multiple immediate gates can be used, with states between them, to enable the use of those system updates without the need of attaching an event or time-span

Timed gates are the gates that have an event and a time-span attached[43][44] and they activate in one of the following two cases: either if an event was triggered or if the time-span has run out. The way in which a gate reacts is determined by its type which can be either positive or negative, depending on the attached sign: if the sign parameter is 'True' the gate is positive and if the sign is 'False' the gate is negative.

A positive timed gate represents an event happening in the given time-span; when that happens, the gate becomes open. If the time-span has not yet started (e.g. if the time-span is RD(2019,2020) but the year is 2018) then the gate will not activate under any circumstance. If the time-span has run out (e.g. if the year is now 2021), then the gate will remain closed.

Positive gates can be opened any number of times by a certain event, as such enabling reccurency[45], and an event can open any number of gates. However, a certain triggering of an event can only open a certain gate once. In other words, if the event has happened once, it will open the gate(s) to which is attached once. In order to open the gate a second time[46] the event has to happen again. In the previous example diagram E_1 is allowed to happen any number of times in the given time-span; if a triggering of an event could open a gate any number of times, when E_1 happens for the first time the contract will start running in an infinite cycle and will also create an infinite amount of contract executions on the S_3 branch.[47] The latter is because a gate with two final states has the meaning of 'concurrency' and it behaves similarly to concurrency in Computer Science: each time a gate connects to two states it starts two executions: one on the first branch and one the other. In this case, because the first branch is recurrent, it can create an infinite number of executions. The requirement of a new event triggering for each opening of a gate prevents this issue.

A negative timed gate represents an event failing to happen in the given time-span; as soon as the time-span runs out, the gate opens and the contract progresses to the following state(s). In fact, a negative gate does not make direct use of its attached event, but the event has three purposes: keep the semantic meaning, convenience[48] and as a safety flag[49] The reason why the event is not needed directly is that the

[43] Having only an event or only a time-span attached is not allowed.

[44] As most gates the dissertation discusses are timed, they are rarely named so.

[45] Represented in a Petri-net as a cycle

[46] Which is necessary in the case of a cycle

[47] The word 'infinite' is used in a purely theoretical manner as the cycle will stop when the time-span runs out or if the system running the digital contract crashes.

[48] So that the person looking at the notation understands what the gate represents.

[49] Obligations are implemented as a pair of trans-assertions. A software could check if for each obligation event there is both a positive and a negative gate, as such ensuring the contract is correctly built.

triggering of the event opens the positive gate and the contract progresses to the following state, as such the negative gate no longer being in a position to open. Therefore it is sufficient for a negative gate to only check if the timespan has expired and opened in that case. Reccurency is the reason this property exists. This becomes obvious by looking at the previous diagram: if the event would be checked for the opening of a negative gate, once the E_1 would have happened once, the contract would not have been able to progress, even if the time-span has passed. This is because the negative gate would always see the event as being triggered so it would never open.

Trans-Assertions

$$TA([initial \quad states], [destination \quad states], gate) \tag{5.13}$$

The trans-assertion component (TA) is used to encode contractual clauses and is formed of two sets of arcs: the first connects each initial state to the gate; the second connects the gate to each destination state. Having multiple incoming or outgoing arcs into a gate or a state can have the following meanings: aggregation, concurrent splitting, mutually exclusive disjunction. These are defined in Section 4.4. A contractual clause can be encoded as one or more TA's or part of a TA and a TA can represent one or more clauses or a part of a clause due to the isomorphism problem detailed in Section 4.3.

Template Definitions and Instances

The template is one of the major innovations this project brings[50]. Templates are parametrised collections of components with various purposes that can be defined once and then used in multiple contracts or more than once in the same contract. The templates are used through two components formalised in this subsection: template-definition and template-instantiation. A template-definition component is used to define the set of contract components, the parameters to the template and its name; each template type is a type in itself. A template instance[51] is needed in order to use the defined template and can be created using the name of the template definition[52] which specifies the id, name and specific parameters that the instance will be using.

[50] Will be fully formalised later in the chapter; the rest of the formalism had to be introduced first.

[51] As suggested earlier, in terms of Computer Science, one can think, for example, of a party component as a class and of a party P_1 as an instance of that class. Similarly, one can think of the template-definition component as an abstract class, the specific template definition a discrete class that inherits it and of the template instance as an instance of that template class.

[52] As the template definition is now a component.

$$TEMPLATE(name, *external-component_1, ... *external-component_N)\{internal-components\}$$

$$TEMPLATE-NAME(id, name, *component_1, ... *component_N)$$

5.4 Templates

The notion of templates was first suggested by Clack[3]. There has already been an introduction of the concept and of syntax. This section will formalise the concept. Templates are tuples of two sets, one containing references to external components and the other defining internal components, with the purpose of defining various legal mechanisms that can easily be integrated into a contract and can be used any number of times in one or more contracts. Internal components are all the components that are defined in the template definition (at the template level) and external components are all the components that are defined at the contract level (i.e. not in any template). Templates have the three properties defined below.

The first encapsulation property: The internal components of a template instance are only visible to that instance and cannot be accessed by external components. This ensures the mechanism of the template cannot be hampered by logic errors in the contract, bad intent and it keeps its information private.

The second encapsulation property: An instance of a contract can not access components that are defined at contract level besides the ones that are made available to that instance through the instantiation mechanism. This ensures that the template can be used in any contract and is not dependent on a certain external component.

The referential parameters property: All external components made available through instantiation parameters are fully available to the template and not just copied. This means that connecting a state from inside the template to a state made available as a parameter will actually connect that first state to the external state. Just as well, a system parameter changing its value externally will also make it change it internally. An event being triggered externally means it is also triggered internally.

Between the external components referenced, some of the components must obligatorily be states. This is because those states are used to connect the internal components of the template with the rest of the contract. As the model is a relational one and states are in succession one of the other, external states are

of two types, input or output states, depending on the direction of the trans-assertions: internal states succeed input states and precede output states. There is a formal requirement that input and output states should never be part of a cycle which results that a state cannot be both an input and an output state. This formal requirement is meant to prevent concurrency issues if the logic of the contract is not adapted to the one of the template and even reccurency issues if an external input/output state is part of a cycle at the contract level.

Templates can have various complexities: they can be short or very extensive. A template can have anywhere from a few components to thousands or more. It can even contain other templates in itself. A template can represent anything from a simple mechanism, such as a sanction for not fulfilling an obligation, to an entire legal agreement. This solves a huge issue: that of having to rewrite contracts every time one is needed. This is very similar to how a bank might have a separate legal agreement with each one of its maybe millions of customers. The individual contracts are nothing else than individualised legal agreement templates, each with a few slight differences (e.g. the parties have different names, the value of loans and of the interest rates might be different etc) but with the same basic functioning of the contract and its legal mechanisms.

Templates solve the canonic form problem identified in Section 4.3. The reason why contracts can be represented in so many ways (we have previously shown as obligations can be represented as prohibitions and vice-versa) and due to the lack of a canonical form. Through the formalisms introduced in the chapter, a construction of a canonical form for digital contracting has started. A following paper should follow to introduce an entire set of such canonic forms in order to achieve a formal canonic form for digital contracting.

6 | Testing and Validation

This chapter attempts to validate the semantic model, formalism and syntax presented in Chapter 5 and tests the proposed approach.

6.1 Validation of the Deontic-Temporal Formalism

The paper introduces the relation between obligations and prohibitions and the notions of discrete and continuous time-spans. Obligations are incompatible with the notion of continuous time: from a logical point of view, there cannot be an obligation that can be fulfilled at every point in time in a range of infinite points of time. On the other hand, prohibitions are incompatible with the notion of discrete time: a prohibition that covers a finite number of points in time in a range with infinite units of time does not make sense. Whilst the notion of continuous obligations exists in Law, they do not actually require a certain behaviour or action to be taken. As such, it is incompatible with the formalism introduced, with Lee's obligations[5] and with Linington 's observations [19][1]. Let there be the following examples:

- Case A: 'The lawyer is under the obligation to continue working in his client's best interests until the representation is terminated.'[36]
- Case B: 'SCIO hereby acknowledges and agrees that its primary duty and obligation hereunder is to provide Services to HGI in a continuous and uninterrupted fashion, by such means, methods or instrumentalities that will best meet the objectives of HGI.'[37]
- Case C: 'The company must keep the antivirus software on its systems updated at all times.'
- Case D: 'The insurer must be licensed by the Financial Conduct Authority (FCA) for at least 5 years for the duration of the contract.'

In case A, the 'obligation' does not require the lawyer to take an action, but it rather prohibits him from taking one against the interest of the client[2]. In case B, SCIO is obliged to provide services in an uninterrupted fashion which required no active action. Actually, SCIO is not allowed to interrupt their services. As an obligation, it would be impossible to be represented, but it would be easily represented as a prohibition. These are pseudo-obligations as it is impossible to check if an obligation is fulfilled

[1] See discussion on Subsection 2.2.3.3
[2] E.g. advising him badly

at every unit of time in a set of infinite time units. There is only one way to check if a commitment is fulfilled in a continuous way and that is by reacting to an event triggered by that obligation not being fulfilled (e.g. SCIO stopping from providing the service), as such proving the concept introduced that legal continuous obligations have the semantics of a prohibition.

In case C, the obligation is not even logically continuous. The obligation requires that once the antivirus developer has released an update, Company X must update their systems within a reasonable time frame. It is technically and logically impossible to have something updated "at all times" because as soon as an update will be released, the systems will no longer be up to date until the migration to the newer software version is completed[3]. In case D, the commitment is actually composed of an obligation and a prohibition. The obligation is that the insurer must have been holding a license from the FCA for at least 5 years at the beginning of the contract, as such referring to discrete time. The prohibition is that the insurer must not lose its license at any time during the contract, as such referring to continuous time.

6.2 Template-Based Parametrised Contract Design

Templates and system parameters change the way smart contracts are designed and help migrate to a template-based parametrised design. It is well beyond the purpose of this project to give canonic forms for all the important mechanisms in a financial contract, but this section will be introducing canonic forms for two fundamental mechanisms in order to be able to evaluate the formalism.

6.2.1 The Non-Exhaustive Prohibition (NEP) Template

Depending on how they are enforced, there are two types of prohibitions: exhaustive and non-exhaustive. Exhaustive prohibitions are the ones that are no longer enforced once violated.[4] Non-exhaustive prohibitions are the ones that remain in place even after being breached. However, it is doubtful that a contract will allow a prohibition to be violated for an infinite number of times. As such, the non-exhaustive mechanism must allow for a different behaviour when the prohibition is violated multiple times[5]. This can be tracked through system parameters. A template that encapsulates the

[3] With the migration obviously taking a certain duration and not being instantaneous.
[4] They tend to be used in cases in which not fulfilling the prohibition leads to a default.
[5] E.g. violating a prohibition two times leads to a warning whilst a third time leads to sanctions

mechanism described would be defined as such:

$TEMPLATE(\text{`}NEP', S1, S4, S5, S6, S7, SYS-P_1, E_1, TS_1)$ {

$S(S_2, \text{`}initiate \quad prohibition', INTERMEDIARY)$

$S(S_3, \text{`}prohibition \quad violated', INTERMEDIARY)$

$SYS-U(SYS-U_1, \text{`}decrease \quad with \quad 1', SYS-P_1, SUB(1))$

$SYS-E(SYS-E_1, \text{`}order \quad reached', SYS-P_1, 0)$

$G(G_2, \text{`}prohibition \quad violation', True, E_1, TS_1)$

$G(G_5, \text{`}discharge \quad prohibition', False, E_1, TS_1)$

$G(G_4, \text{`}complete \quad prohibtion \quad violation', True, SYS-E_1, TS_1, SYS-U_1)$

$G(G_3, \text{`}iterrative \quad prohibtion \quad violation', False, SYS-E_1, TS_1, SYS-U_1)$

$G(G_1, \text{`}concurrency', True)$

$TA([S_1], [S_2, S_7], G_1)$

$TA([S_2], [S_3], G_2)$

$TA([S_3], [S_2, S_4], G_3)$

$TA([S_3], [S_5], G_4)$

$TA([S_2], [S_6], G_5)$

}

An NEP instantiation looks like this: $TEMPLATE\_NEP(S_8, S_9, S_{50}, S_{62}, S_{64}, SYS-P_{12}, E_7, TS_9)$ - this is sending a set of components as parameters which contain one input state S_1 (which corresponds with S_8 at contract level), 4 output states, the maximum number of times the prohibition can be violated as $SYS-P_1$, the prohibited event E_1 and the continuous time-span TS_1 in which the prohibition is enforced. By making use of the Smart Contract Editor (SCE) software tool included with this project and documented in Appendix B, the template can be visualised as a PetriNet diagram:

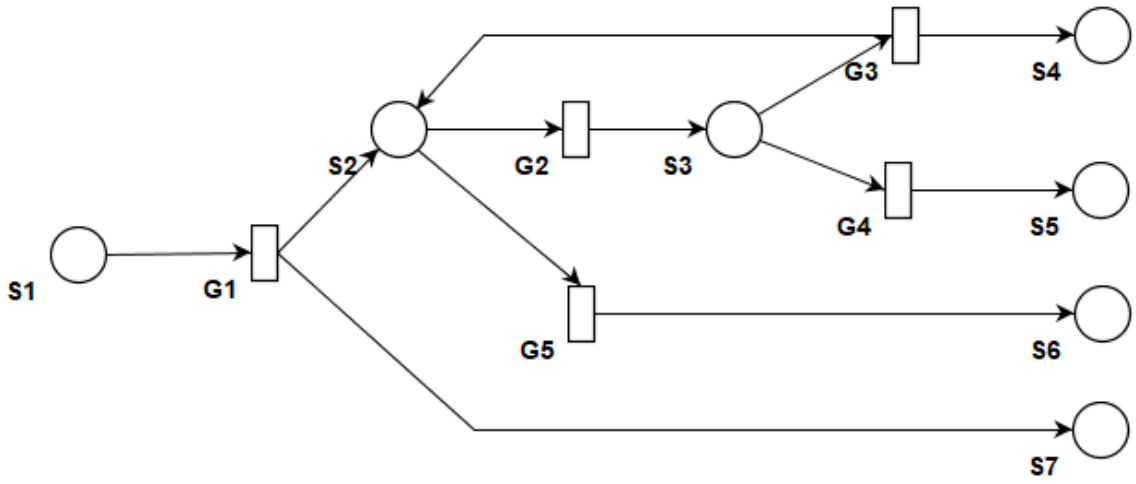


Figure 6.1: The Non-Exhaustive Prohibitions (NEP) Template

The output state S_4 leads to consequences per prohibition violation. As a single violation might not necessarily lead to sanctions or to default, this branch allows for other types of consequences, such as handling a warning to the perpetrator, and will not discharge the prohibition. S_5 leads to consequences per multiple violations and it discharges the prohibition (e.g. it can lead to a default). S_6 corresponds

to the prohibition being discharged due to TS_1 running out and S_7 simply continues the branch that was partitioned by the introduction of the prohibition. This is because, as explained in the formalism definition, prohibitions should be implemented in a concurrent branch. The immediate positive gate G_1 creates the concurrent branches. G_2 opens if the prohibited event happens within the time-span and G_5 opens if the time-span has run out. The positive gate G_2 makes use of the property that says that a single triggering of an event opens a gate a single time, as such avoiding an infinite cycle. The negative gate G_5 makes use of the property of ignoring the fact the event was triggered before, avoiding as such blocking the branch once the time-span runs out. G_3 opens if the numerical system parameter $SYS-P_1$ has not yet reached the value of 0 whilst G_4 opens when 0 is reached. $SYS-P_1$ acts therefore as a countdown through the SUB(1) operator in $SYS-U_1$ and the system-event $SYS-E_1$.

6.2.2 The Repeated Obligations (ReO) Template

There are cases in which an obligation has to be executed more than once in specific periods of time. Such examples could be mortgage repayment contracts that require monthly payments for a given period. Cases like this can be easily solved through the use of various time operators, such as SCH for the given example. However, other cases are more complex as they require that an obligation is executed multiple times in a single time-span. This requires the use of counter implemented via system parameters. A model of implementing this mechanism is provided in the following template:

$$
\begin{array}{ll}
TEMPLATE(`ReO', S_1, S_4, S_5, S_6, E_1, TS_1, SYS-P_1) \quad \{ & \\
S(S_2, `initiated obligation', INTERMEDIARY) & \\
S(S_3, `obligation \quad completed \quad once \quad more', INTERMEDIARY) & \\
SYS-U(SYS-U_1, `countdown', SYS-P_1, SUB, 1) & TA([S_1],[S_2],G_1) \\
SYS-E(SYS-E_1, `order \quad reached', SYS-P_1, 0) & TA([S_2],[S_3],G_3) \\
G(G_1, `separate \quad cycle', True) & TA([S_3],[S_2,S_4],G_2) \\
G(G_3, `do \quad action', True, E_1, TS_1) & TA([S_3],[S_5],G_5) \\
G(G_4, `deadline \quad expired', False, E_1, TS_1) & TA([S_2],[S_6],G_4) \\
G(G_5, `discharge \quad obligation', True, SYS-E_1, TS_1, SYS-U_1) & \quad \} \\
G(G_2, `update \quad obligation \quad status', False, SYS-E_1, TS_1, SYS-U_1) &
\end{array}
$$

By making again use of the SCE software tool, the ReO template can be visualised as a PetriNet:

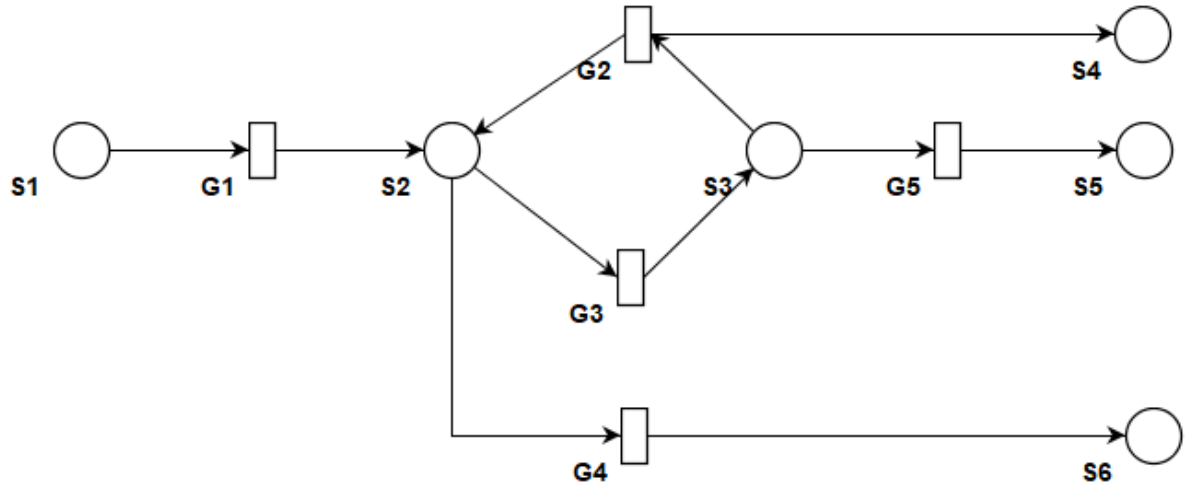


Figure 6.2: The Repeated Obligations (ReO) Template

For the ReO template, the list of parameters is made of one input state S_1, 3 output states S_4, S_5, S_6, the obligatory event E_1, the required time-span TS_1 and number of times the action has to be completed $SYS - P_1$ [6]. S_4 corresponds to the event being completed for the first $SYS - P_1$-1 times and can be used to provide notifications to the party that the event was completed. S_5 corresponds to the obligation being completed for the $SYS - P_1$ time. Finally, S_6 corresponds to the failure the complete the action the required number of times in the required time-span.

G_1 is a permanent true gate which has the simple purpose of separating the cycle in the template from the rest of the contract. This is necessary as obligations might not always be implemented concurrently like prohibitions are. G_3 opens when the action is completed one more time (within the time-spans) and the gates G_2 and G_5 open if the action still has to be completed or if it has been completed the Nth time respectively. G_4 simply checks if the deadline has not yet expired and opens if it is.

With the observations and the analysis made in this chapter and with the visual representations provided by SCE, validations of many of the important concepts and components introduced in the paper have been provided which have also acted as tests for the software created. Appendix A.3 contains two larger tests and analysis by writing two legal agreements in trans-assertion notation as well as presenting the respective Petri-net diagrams. One of the contracts comes with a comparative analysis between the formalism introduced in this paper and Lee's and Pithadia's formalisms. The second contract is impossible to be represented with the old formalism and requires the model introduced in this paper.

[6] i.e. the number of times the event has to be triggered

7 | Summary and Conclusions

This chapter contains a summary of the achievements of the project, a critical assessment of the formalism and suggestions for future research directions on the topics discussed.

7.1 Achievements. Evaluation of the Objectives

The project has started with the aim of investigating a viable semantic system for the construction of high-value, long-duration, highly-regulated, standardised smart contracts that can work to automate various procedures in banking and financial services. This aim has been achieved, together with all the set out objectives: the review and assessment of logic systems for smart contracts was conducted in Chapters 2 to 4; the creation of a system for semantic analysis of smart contracts that includes the temporal, deontic and operational logics and introduces system parameters and templates was done in Chapter 5 and additional information about the syntax, including a BNF, is provided in Appendix A; the testing and validation of the proposed model is present in Chapter 6 and Appendix A.3; a software tool for graphic representations of contracts with the purpose of conducting further tests of the model and helping with its validation has been developed and documented in Appendix B.

Moreover, the project has exceeded the original objectives. New mechanisms have been introduced in order to represent temporal aspects that were impossible to represent before. The concepts of discrete and continuous time-spans have changed completely the way basic deontic components are formalised. These two concepts, together with relative, absolute and recurrent time will form the basis of future research on smart contract formalism.[1] This paper also formalised two fundamental mechanisms (NEP and ReO), contributing to the creation of a formal canonic form for smart contracts. The original purpose of the software has been extended from just drawing Petri-net representations of the contracts to running simulations based on them. Finally, an advanced formalism of the contract components and of the properties of each component has been realised. These properties will form the foundation of future research in the properties of a correctly built smart contract and in achieving a canonic form for them.

[1] Even though some of the concepts have been mentioned in some of the analysed papers, no formal definitions have been given to them until in this project.

7.2 Future Work

Despite important advancements that have been in the project, smart contracting is an on-going area of research. Some of the most immediate items that should be tackled for the improvement of the proposed formalism are presented further in this section.

An analysis of the concurrency mechanism using the proposed formalism should be realised, including the issues that could be encountered. Some smart contract designing practices should be established (e.g. such as always having prohibitions on a concurrent branch) in order to avoid infinite cycles or other concurrency issues typical in software development. There should be an analysis of the basic gate operators provided in Lee's formalism such as conjunction and disjunction, referred to in this paper as branch joining and branch splitting or concurrency respectively as some additional operators might be necessary: for example branch joining is done as a conjunction, but there are cases where it would be beneficial to work as a disjunction. There might be the need for an extension of the system parameters concept and for additional types of system parameters and parameter operations. There is also probably a need for more types of system events, besides the current equality event, such as "greater than" or "smaller than" a given number or equality with another system parameter.

As suggested in the project as well as by Clack [3], future research should look into a pausing mechanism and how it can be implemented in the existing formalism. Moreover, a future paper should follow to introduce an entire set of canonic templates for standard mechanisms, such as the ones introduced here, in order to achieve a formal canonic form for digital contracting. It has also been noted that the temporal system proposed still has a few limitations: e.g. 'to be done within 5 days' is easy to represent using the AFT operator, but there is no way to represent 'to be done within 5 working days'. There is likely a need for more relative time operators.

Furthermore, there is a need for the development of formal semantics for the notation as the current formalism (as well as Lee's[2]) lacks that. Finally, the software provided with the paper should be continued and updated with changes to the formalism. Unfortunately, due to the limited time available for this project, there are a few operators not yet supported by SCE. This tool has proven invaluable in the work to test and validate the formalism and it can help validate whether contracts are built correctly, both by querying the contracts[3] as well as for checking if the contract respects various properties[4].

[2] As indicated by Hvited[12] and as shown in Chapter 2
[3] Option already available in the software
[4] Planned to be implemented in the future

A | Appendix A - Formalism and Syntax

A.1 Description of Syntax. Examples

This section details the syntax for each component of the formalism and provides example of how they can be used. This section will also detail the syntax for all the time and time-span operators and for the date formats together with examples for each one of them. The parameters marked with $*$ are optional ($*$ in the examples of the SCH operator means 'each one'). The formalism components and operators marked with $**$ are not available in the SCE software tool provided with the project.

Component	Formalism	Examples
CONTRACT (C)	Contract (name, start date, end date)	CONTRACT('Acquisition Contract', 1987-04-27, 1987-05-15) C('Non-Disclosure Agreement', 2018, N/A)
PARTY (P)	Party(Id, name)	Party (P1, 'John Smith')
STATE (S)	State (Id, name, type)	State (S1, 'contract beginning', START) S (S3, 'item sent', INTERMEDIARY) S (S7, 'John didn't pay', DEFAULT) STATE (S9, 'contract end', END)
DISCRETE TIME (DT)	DT (Id, name, time reference)	DT (DT1, "year start", 2018-Jan-01) DT (DT2, "contract beginning", 2019-Apr-06 at 09:30:00.00) DT (DT3, "2 days and 12 hours after payment", AFT(E1,00,00,20,12)) DT (DT4, "May every year",SCH(*, [5]))
TIME-SPAN (TS)	Time-Span (Id, name, time operator)	TS (T1,"between dates", RD (DT1, DT2)) TS (T2,"between events", RD (S7, E2))
ACTION (A)	Action (Id, name, *parameter 1,...,*parameter N)	Action (A1, 'Pay', $100, P2)

DISCRETE EVENT (E)	Event (Id, name, actor, action)	E (E1, "paid $100", P1, A1) E (E2, "forbidden function", P2, A2)
GATE (G)	Gate (Id, name, sign, *event, *time-span, *system action)	Gate (G1, "paid $100", True, E1, T2) G (G2, "not paid $100", False, E1, T2) G (G3, "trigger sanction", True) G (G4, "update parameter, True, SYS-U1)
SYSTEM - PARAMETER (SYS-P) **	SYS-P (Id, name, initial value)	SYS-P (SYS-P1, "numeric parameter", 0)
SYSTEM - UPDATE (SYS-U) **	SYS-U (Id, name, parameter, operation)	SYS-U (SYS-U1, "increase with 1", SYS-P1, ADD(1))
SYSTEM - EVENT (SYS-E) **	SYS-E (Id, name, parameter, target value)	SYS-E (SYS-E1, "order reached", SYS-P1, 18)
TRANS-ASSERTION (TA)	TA ([begin states], [end states], gate)	TA ([S1, S2], [S3], G1) TA ([S1, S2, S3], [S4, S5, S6], G2)
TEMPLATE-DEFINITION **	TEMPLATE (name, *template parameter 1, ... *template parameter N) {list of components}	TEMPLATE("PROHIBITION", S1, S2, E1, T1) { G (G1, "prohibition broken", True, E1, T1) TA ([S1], [S2], G1) }
TEMPLATE-INSTANTIATION **	TEMPLATE-NAME (Id, name, *template parameter 1, ... *template parameter N)	TEMPLATE-PROHIBITION(TEMPLATE-PROHIBITION76, "doing prohibited action leads from S5 to S9", S5, S9, E7, TS3)

Table A.1: Syntax of Contract Components

DT & TS Operators	Formalism	Examples
REALISED DURING (RD)	RD(Initial date, final date)	RD(DT1, DT2) RD(E3, S5)
REALISED THROUGHOUT (RT)	RT(Initial date, final date)	RT(DT3, DT7) RT(S2, E8)
AFTER (AFT)	AFT(Reference date, years, *months, *days, *hours, *minutes, *seconds, *milliseconds)	AFT(DT4, 1, 5) AFT(E2, 5, 2, 0, 4, 3)
BEFORE (BFR)	BFR(Reference date, years, *months, *days, *hours, *minutes, *seconds, *milliseconds)	BFR(DT4, 4, 6) BFR(S4, 2)
EARLIEST (EST)	EST(time reference 1, time reference 2)	EST (DT4, DT5)
LATEST (LST)	LST(time reference 1, time reference 2)	LST (DT4, DT7)
SCHEDULE (SCH) **	SCH({YB,YE,YS}, *{MB,ME,MS}, *{DMB,DME,DMS}, *{DWB,DWE,DWS}, *{HB,HE,HS}, *{MINB,MINE,MINS}, *{SB,SE,SS}, *{MLSB,MLSE,MLSS})	SCH(*,*,*, {Mon,Fri,1})
	SCH([Y1, Y2,...,YN], *[M1, M2,...,MN], *[DM1, DM2,...,DMN], *[DW1, DW2,...,DWN], *[H1, H2,...,HN], *[MIN1, MIN2,...,MINN], *[S1, S2,...,SN], *[MLS1, MLS2,...,MLSN])	SCH(*, [Mar])

Table A.2: Syntax of Time and Time-Span Operators

	Formalism	Examples
DISCRETE TIME REFERENCE	yyyy, yyyy-MM, yyyy-MM-dd, yyyy-MM-dd at HH, yyyy-MM-dd at HH:mm, yyyy-MM-dd at HH:mm:ss, yyyy-MM-dd at HH:mm:ss.SSS	2018-Jul, 2019-03-24, 2020-Jul-10 at 03:20:43.752

Table A.3: Syntax of Date Formats

A.2 Backus-Naur Form (BNF)

As the project has to objective of creating a clear and rigorous syntax, it defines the following context free grammar using the Backus-Naur Form (BNF) notation technique. For this, the *backnaur* package[38] for LaTeX has been used. The notation technique is commonly used for defining the syntax of programming languages[1] and it is made of productions. Each one of the productions defines an element of the syntax on a separate line and is composed of two arguments between which stands the 'production' symbol $\models$. The first argument is the name of the syntax element (e.g. "string" , "state") and the second argument is the definition of the syntax element. The symbol epsilon ε means that argument can be empty and | is the *OR* operator.

⟨contract_definition⟩ ⊨ ⟨contract_header⟩⟨contract_body⟩

⟨contract_header⟩ ⊨ 'CONTRACT' '(' ⟨name⟩ ',' ⟨contract_start_date⟩
',' ⟨contract_termination_date⟩')'

⟨contract_body⟩ ⊨ ⟨contract_element⟩ | ⟨contract_element⟩⟨contract_body⟩

⟨contract_element⟩ ⊨ ⟨party_definition⟩ | ⟨state_definition⟩ | ⟨discrete-time_definition⟩
| ⟨time-span_definition⟩ | ⟨action_definition⟩ | ⟨event_definition⟩
| ⟨gate_definition⟩ | ⟨system-parameter_definition⟩
| ⟨system-update_definition⟩ | ⟨trans-assertion_definition⟩
| ⟨template_definition⟩ | ⟨template_instantiation⟩

⟨contract_start_date⟩ ⊨ ⟨discrete_time_reference⟩

⟨contract_termination_date⟩ ⊨ ⟨discrete_time_reference⟩ | 'N/A'

⟨party_definition⟩ ⊨ 'P' '(' ⟨party_id⟩ ',' ⟨name⟩')'

⟨state_definition⟩ ⊨ 'S' '(' ⟨state_id⟩ ',' ⟨name⟩ ',' ⟨state_type⟩ ')'

⟨discrete-time_definition⟩ ⊨ 'DT' '(' ⟨discrete-time_id⟩ ',' ⟨name⟩ ',' ⟨time_reference⟩ ')'

⟨time-span_definition⟩ ⊨ 'TS' '(' ⟨time-span_id⟩ ',' ⟨name⟩ ',' ⟨time-span_operator⟩ ')'

⟨action_definition⟩ ⊨ 'A' '(' ⟨action_id⟩ ',' ⟨name⟩ ',' ⟨action_parameter_list⟩')'

[1] A Java BNF example can be found at https://users-cs.au.dk/amoeller/RegAut/JavaBNF.html

⟨system-parameter_definition⟩ ⊨ 'SYS-P' '(' ⟨system-parameter_id⟩ ',' ⟨name⟩ ','
⟨system-parameter_initial_value⟩')'

⟨system-update_definition⟩ ⊨ 'SYS-U' '(' ⟨system-update_id⟩ ',' ⟨name⟩ ','
⟨system-parameter_id⟩ ',' ⟨system-update_operation⟩')'

⟨event_definition⟩ ⊨ ⟨discrete-event_definition⟩ | ⟨system-event_definition⟩

⟨discrete-event_definition⟩ ⊨ 'E' '(' ⟨discrete-event_id⟩ ',' ⟨name⟩ ',' ⟨party_id⟩ ',' ⟨action_id⟩')'

⟨system-event_definition⟩ ⊨ 'SYS-E' '(' ⟨system-event_id⟩ ',' ⟨name⟩ ','
⟨system-parameter_id⟩ ',' ⟨system-parameter_target_value⟩')'

⟨gate_definition⟩ ⊨ 'G' '(' ⟨gate_id⟩ ',' ⟨name⟩ ',' ⟨boolean_sign⟩')'
| 'G' '(' ⟨gate_id⟩ ',' ⟨name⟩ ',' ⟨boolean_sign⟩ ','
⟨event_id⟩ ',' ⟨time-span_id⟩')'
| 'G' '(' ⟨gate_id⟩ ',' ⟨name⟩ ',' ⟨boolean_sign⟩ ',' ⟨system-event_id⟩')'

⟨trans-assertion_definition⟩ ⊨ 'TA' '(' ⟨state_list⟩ ',' ⟨state_list⟩ ',' ⟨gate_id⟩')'

⟨template_definition⟩ ⊨ ⟨template_header⟩ '{' ⟨template_body⟩ '}'

⟨template_header⟩ ⊨ 'TEMPLATE' '(' template_name ',' template_list_of_parameters ')'

⟨template_body⟩ ⊨ ⟨contract_element⟩ | ⟨template_body⟩⟨contract_element⟩

⟨template_instantiation⟩ ⊨ ⟨template_name⟩ '(' ⟨template_instance_id⟩ ','
⟨template_instance_name⟩ ',' ⟨template_list_of_parameters⟩ ')'

⟨template_list_of_parameters⟩ ⊨ ⟨contract_element⟩
| ⟨contract_element⟩ ',' ⟨template_list_of_parameters⟩

⟨template_name⟩ ⊨ 'TEMPLATE' '_' ⟨string_letters_only⟩

⟨template_instance_name⟩ ⊨ ⟨template_name⟩⟨number⟩

⟨system-parameter_initial_value⟩ ⊨ ⟨system-parameter_value⟩

⟨system-parameter_target_value⟩ ⊨ ⟨system-parameter_value⟩

⟨system-parameter_value⟩ ⊨ string | number | time_reference

⟨state_type⟩ ⊨ ‘START’ | ‘INTERMEDIARY’ | ‘END’
| ‘DEFAULT’ | ‘PAUSE’

⟨time_reference⟩ ⊨ ⟨discrete-time_reference⟩ | ⟨time_operator⟩
| ⟨state_id⟩ | ⟨event_id⟩ | ⟨discrete-time_id⟩

⟨time_operator⟩ ⊨ ⟨after_operator⟩ | ⟨before_operator⟩ | ⟨schedule_operator⟩
⟨earliest_operator⟩ | ⟨latest_operator⟩

⟨after_operator⟩ ⊨ ‘AFT’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_period⟩‘)’

⟨before_operator⟩ ⊨ ‘BFR’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_period⟩‘)’

⟨earliest_operator⟩ ⊨ ‘EST’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_reference⟩‘)’

⟨latest_operator⟩ ⊨ ‘LST’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_reference⟩‘)’

⟨schedule_operator⟩ ⊨ ‘SCH’ ‘(’ ⟨schedule_parameter_list⟩‘)’

⟨time-span_operator⟩ ⊨ ⟨realised_during_operator⟩ | ⟨realised-throughout_operator⟩

⟨realised-during_operator⟩ ⊨ ‘RD’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_reference⟩‘)’

⟨realised-throughout_operator⟩ ⊨ ‘RT’ ‘(’ ⟨time_reference⟩ ‘,’ ⟨time_reference⟩‘)’

⟨schedule_parameter_list⟩ ⊨ schedule_years ‘,’ schedule_months ‘,’ schedule_days ‘,’
schedule_hours ‘,’ schedule_minutes ‘,’ schedule_seconds ‘,’
schedule_milliseconds

⟨system-update_operation⟩ ⊨ ⟨addition_operation⟩ | ⟨subtraction_operation⟩
| ⟨multiply_operation⟩ | ⟨divide_operation⟩
| ⟨attribute_operation⟩ | ⟨remove-last-characters_operation⟩

⟨addition_operation⟩ ⊨ ‘ADD’ ‘(’ ⟨number⟩‘)’ | ‘ADD’ ‘(’ ⟨time_period⟩‘)’
| ‘ADD’ ‘(’ ⟨string⟩‘)’

⟨subtraction_operation⟩ ⊨ ‘SUB’ ‘(’ ⟨number⟩‘)’ | ‘SUB’ ‘(’ ⟨time_period⟩‘)’

⟨multiply_operation⟩ ⊨ ‘MUL’ ‘(’ ⟨number⟩‘)’

⟨remove_operation⟩ ⊨ ‘DIV’ ‘(’ ⟨number⟩‘)’

⟨remove-last-characters_operation⟩ ⊨ ‘RMV_LST’ ‘(’ ⟨number_of_characters_to_remove⟩‘)’

⟨number_of_characters_to_remove⟩ ⊨ ⟨number⟩

⟨attribute_operation⟩ ⊨ ‘ATR’ ‘(’ ⟨system-parameter_id⟩‘)’
| ‘ATR’ ‘(’ ⟨system-parameter_value⟩‘)’

⟨state_list⟩ ⊨ ‘[’ ⟨state_parameter_list⟩ ‘]’

⟨state_parameter_list⟩ ⊨ ⟨state_id⟩ | ⟨state_id⟩ ‘,’ ⟨state_parameter_list⟩

⟨action_parameter_list⟩ ⊨ ⟨string⟩ | ⟨string⟩ ‘,’ ⟨action_parameter_list⟩

⟨contract_element_id⟩ ⊨ ⟨party_id⟩ | ⟨state_id⟩ | ⟨discrete-time_id⟩ | ⟨time-span_id⟩ | ⟨action_id⟩ | ⟨event_id⟩ | ⟨system-update_id⟩ | ⟨system-parameter_id⟩ | ⟨gate_id⟩ | ⟨template_instance_id⟩

⟨party_id⟩ ⊨ ‘P’ ⟨number⟩

⟨state_id⟩ ⊨ ‘S’ ⟨number⟩

⟨discrete-time_id⟩ ⊨ ‘DT’ ⟨number⟩

⟨time-span_id⟩ ⊨ ‘TS’ ⟨number⟩

⟨action_id⟩ ⊨ ‘A’ ⟨number⟩

⟨event_id⟩ ⊨ ⟨discrete-event_id⟩ | ⟨system-event_id⟩

⟨discrete-event_id⟩ ⊨ ‘E’ ⟨number⟩

⟨system-event_id⟩ ⊨ ‘SYS-E’ ⟨number⟩

⟨system-update_id⟩ ⊨ ‘SYS-U’ ⟨number⟩

⟨system-parameter_id⟩ ⊨ ‘SYS-P’ ⟨number⟩

⟨gate_id⟩ ⊨ ‘G’ ⟨number⟩

⟨template_instance_id⟩ ⊨ ⟨template_name⟩⟨number⟩

⟨number⟩ ⊨ ⟨digit⟩ | ⟨digit⟩⟨number⟩

⟨digit⟩ ⊨ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨name⟩ ⊨ ‘ " ’ ⟨string⟩ ‘ " ’ | ‘ ‘ ’ ⟨string⟩ ‘ ’ ’

⟨string⟩ ⊨ ⟨character⟩ | ⟨character⟩⟨string⟩

⟨string_letters_only⟩ ⊨ ⟨letter⟩ | ⟨letter⟩⟨string_letters_only⟩

⟨character⟩ ⊨ ⟨letter⟩ | ‘ ’ | ‘-’ | ‘_’ | ‘ ’ ’ | ⟨digit⟩

⟨letter⟩ ⊨ ‘A’ … ‘Z’ | ‘a’ … ‘z’

⟨boolean_sign⟩ ⊨ ‘TRUE’ | ‘FALSE’

⟨discrete_time_reference⟩ ⊨ ⟨year⟩ | ⟨year⟩ ‘-’ ⟨month⟩ | ⟨year⟩ ‘-’ ⟨month⟩ ‘-’ ⟨day_of_month⟩
| ⟨year⟩ ‘-’ ⟨month⟩ ‘-’ ⟨day_of_month⟩ ‘at’ ⟨hour⟩
| ⟨year⟩ ‘-’ ⟨month⟩ ‘-’ ⟨day_of_month⟩ ‘at’ ⟨hour⟩ ‘:’ ⟨minute⟩
| ⟨year⟩ ‘-’ ⟨month⟩ ‘-’ ⟨day_of_month⟩ ‘at’ ⟨hour⟩ ‘:’ ⟨minute⟩ ‘:’ ⟨second⟩
| ⟨year⟩ ‘-’ ⟨month⟩ ‘-’ ⟨day_of_month⟩ ‘at’ ⟨hour⟩ ‘:’ ⟨minute⟩ ‘:’ ⟨second⟩
‘.’ ⟨millisecond⟩

⟨time_period⟩ ⊨ ⟨year⟩ | ⟨year⟩ ‘,’ ⟨month⟩ | ⟨year⟩ ‘,’ ⟨month⟩ ‘,’ ⟨day_of_month⟩
| ⟨year⟩ ‘,’ ⟨month⟩ ‘,’ ⟨day_of_month⟩ ‘,’ ⟨hour⟩
| ⟨year⟩ ‘,’ ⟨month⟩ ‘,’ ⟨day_of_month⟩ ‘,’ ⟨hour⟩ ‘,’ ⟨minute⟩
| ⟨year⟩ ‘,’ ⟨month⟩ ‘,’ ⟨day_of_month⟩ ‘,’ ⟨hour⟩ ‘,’ ⟨minute⟩ ‘,’ ⟨second⟩
| ⟨year⟩ ‘,’ ⟨month⟩ ‘,’ ⟨day_of_month⟩ ‘,’ ⟨hour⟩ ‘,’ ⟨minute⟩ ‘,’ ⟨second⟩
‘,’ ⟨millisecond⟩

⟨year⟩ ⊨ ⟨number⟩

⟨month⟩ ⊨ 1 …12 | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct |
Nov | Dec | January | February | March | April | May | June | July |
August | September | October | November | December

⟨day_of_month⟩ ⊨ 1 …31

⟨hour⟩ ⊨ 0 …23

⟨minute⟩ ⊨ 0 …59

⟨second⟩ ⊨ 1 …59

⟨millisecond⟩ ⊨ 1 …999

⟨schedule_years⟩ ⊨ '*' | '{' schedule_years_start ',' schedule_years_end ',' schedule_years_step | '[' years_list ']' | ε

⟨years_list⟩ ⊨ ⟨year⟩ | ⟨year⟩ ',' ⟨years_list⟩

⟨schedule_years_step⟩ ⊨ ⟨number⟩

⟨schedule_months⟩ ⊨ '*' | '{' schedule_months_start ',' schedule_months_end ',' schedule_months_step | '[' months_list ']' | ε

⟨months_list⟩ ⊨ ⟨month⟩ | ⟨month⟩ ',' ⟨months_list⟩

⟨schedule_months_step⟩ ⊨ ⟨number⟩

⟨schedule_days_of_month⟩ ⊨ '*' | '{' schedule_days_of_month_start ',' schedule_days_of_month_end ',' schedule_days_of_month_step '}' | '[' days_list ']' | ε

⟨days_of_month_list⟩ ⊨ ⟨day_of_month⟩ | ⟨day_of_month⟩ ',' ⟨days_of_month_list⟩

⟨schedule_days_of_month_step⟩ ⊨ ⟨number⟩

⟨schedule_hours⟩ ⊨ '*' | '{' schedule_hours_start ',' schedule_hours_end ',' schedule_hours_step '}' | '[' hours_list ']' | ε

⟨hours_list⟩ ⊨ ⟨hour⟩ | ⟨hour⟩ ',' ⟨hours_list⟩

⟨schedule_hours_step⟩ ⊨ ⟨number⟩

⟨schedule_minutes⟩ ⊨ '*' | '{' schedule_minutes_start ',' schedule_minutes_end ',' schedule_minutes_step '}' | '[' minutes_list ']' | ε

⟨minutes_list⟩ ⊨ ⟨minute⟩ | ⟨minute⟩ ',' ⟨minutes_list⟩

⟨schedule_minutes_step⟩ ⊨ ⟨number⟩

⟨schedule_seconds⟩ ⊨ '*' | '{' schedule_seconds_start ',' schedule_seconds_end ',' schedule_seconds_step '}' | '[' seconds_list ']' | ε

⟨seconds_list⟩ ⊨ ⟨second⟩ | ⟨second⟩ ',' ⟨seconds_list⟩

⟨schedule_seconds_step⟩ ⊨ ⟨number⟩

⟨schedule_milliseconds⟩ ⊨ '*' | '{' schedule_milliseconds_start ',' schedule_milliseconds_end ',' schedule_milliseconds_step '}' | '[' milliseconds_list ']' | ε

⟨milliseconds_list⟩ ⊨ ⟨millisecond⟩ | ⟨millisecond⟩ ',' ⟨milliseconds_list⟩

⟨schedule_milliseconds_step⟩ ⊨ ⟨number⟩

A.3 Additional Testing and Validation of Formalism

This section provides two examples of contracts that use the introduced formalism together with an analysis made in each one of the examples. Two legal agreements will be presented using the trans-assertion notation and Petri-net diagrams. One of the contracts comes with a comparative analysis between the formalism introduced in this paper and Lee's and Pithadia's formalisms. The second contract is impossible to be represented with the old formalism and requires the model introduced in this paper. These two examples contribute to testing whether the formalism brought forward represents all introduced concepts and that it solves most of the issues identified in Chapter 4 about Lee's and Pithadia's model.

Product Purchase Contract

In his paper, Lee[5] provides a number of contracts to demonstrate his formalism by representing them using the trans-assertion notation. The most detailed contract provided in his paper is an agreement between two parties (Jones and Smith) for the purchase of an appliance. Pithadia [27] identifies a few conflicts and issues and re-writes this contract using his model. A third method of writing this contract using the formalism provided in this paper will be presented, as such enabling a comparison between the three models. Let there be the following agreement provided in Lee's paper [5]:

'Jones agrees to pay £500 to Smith by May 3,1987. Following that, Smith agrees to deliver a washing machine to Jones by 15 May 1987.'

Trans-Assertion (TA) Notation

Lee[5] provides the following trans-assertion notation for this agreement:

$$trans([s(start, 1-Jan-0)], [s(1, \_93)], True)$$

$trans([s(1,\_83)],[s(2,\_89)],Jones : rb(3-may-1987) : Pay(Smith,£500))$

$trans([s(1,\_83)],[s(default(Jones),\_90)],Jones : rb(3-may-1987) : Pay(Smith,£500))$

$trans([s(2,\_83)],[s(3,\_89)],true)$

$trans([s(3,\_83)],[s(4,\_89)],true)$

$trans([s(4,\_83)],[s(5,\_89)],Smith : rw(10days) : Deliver(Jones,Washer))$

$trans([s(4,\_83)],[s(default(Smith),\_90)],Smith : rw(10days) : Deliver(Jones,Washer))$

$trans([s(5,\_83)],[s(finish,\_89)],true)$

Following the corrections made by himself, Pithadia[27] provides the following notation:

$trans(s(start,1-Jan-0),s(1,83),True),$

$trans(s(1,83),s(2,84),Jones : rb((3-May-1987)) : Pay(Smith,£500)),$

$trans(s(1,83),s(default(Jones),90),\ Jones : rb((3-May-1987)) : Pay(Smith,£500)),$

$trans(s(2,84),s(3,85),Smith : rb((15-May-1987)) : Deliver(Jones,Washer)),$

$trans(s(2,84),s(default(Smith),90),\ Smith : rb((15-May-1987)) : Deliver(Jones,Washer)),$

$trans(s(3,85),s(finish,91),True)$

The first thing to notice here is that Pithadia abandons the bracket notation for sets of states. He also removes redundant trans-assertions and fixes some trans-assertion that seem to be wrongly connected. Otherwise, besides introducing around strings, he provides little changes to the formalism.

Using the formalism introduced here, the same legal agreement would be translated in trans-assertion notation in the following way:

$CONTRACT(\text{"}Washer\quad Acquisition\quad Contract\text{"},1987-04-27,1987-05-15)$

$PARTY(P1,\text{"}Jones\text{"})$

$PARTY(P2,\text{"}Smith\text{"})$

$STATE(S1,'START',START)$

$STATE(S2,'Initiate\quad obligation',INTERMEDIARY)$

$STATE(S3,'Jones\quad paid\quad 500',INTERMEDIARY)$

$STATE(S4,'DEFAULT: \quad Jones \quad did \quad not \quad pay \quad £500',DEFAULT)$

$STATE(S5,'Smith \quad delivered \quad washer',INTERMEDIARY)$

$STATE(S6,'DEFAULT: \quad Smith \quad did \quad not \quad deliver \quad washer',DEFAULT)$

$STATE(S7,'END',END)$

$DISCRETE-TIME(DT1,"Payment \quad due \quad date",1987-05-03)$

$DISCRETE-TIME(DT2,"Delivery \quad due \quad date",1987-05-15)$

$ACTION(A1,"PAY",P2,£500)$

$ACTION(A2,"DELIVER",P1,Washer)$

$EVENT(E1,"Pay \quad for \quad Washer",P1,A1)$

$EVENT(E2,"Deliver \quad Washer",P2,A2)$

$TIME-SPAN(TS1,"Payment \quad Deadline",RD(S1,DT1))$

$TIME-SPAN(TS2,"Delivery \quad Deadline",RD(E1,DT2))$

$GATE(G1,"Initiate \quad obligation",TRUE)$

$GATE(G2,"Washer \quad not \quad paid \quad in \quad time",TRUE,E1,TS1)$

$GATE(G3,"Pay \quad for \quad Washer",FALSE,E1,TS1)$

$GATE(G4,"Deliver \quad Washer",TRUE,E2,TS2)$

$GATE(G5,"Washer \quad not \quad delivered \quad in \quad time",FALSE,E2,TS2)$

$GATE(G6,"End \quad contract",TRUE)$

$TA([S1],[S2],G1)$

$TA([S2],[S3],G2)$

$TA([S2],[S4],G3)$

$TA([S3],[S5],G4)$

$TA([S3],[S6],G5)$

$TA([S5],[S7],G6)$

The first thing one probably notices is that the formalism introduced makes the contract much larger. Besides the fact that the introduced model allows for a large number of new mechanism and solves many issues, it is scalable, easier to understand and, likely, the difference in size might decrease when considering very large contracts. The following thing to note about this trans-assertion notation is that

it resembles much more the format of a legal text: it starts with the name and the start and end dates of the contracts, it continues with the name of the parties, it describes the various states of the contracts, then the timing of various actions and then it contains the clauses of the contract.

Petri-net Diagram Representation via SCE

Being able to visualise the contract in the form of Petri-net diagrams provides a sizeable contribution to validating the formalism or to finding out whether the contract is correctly built (e.g. if a state is not connected to any other state, that indicates an error). When inputing the trans-assertion in the Smart Contract Editor (SCE) software tool provided, the following Petri-net diagram is provided:

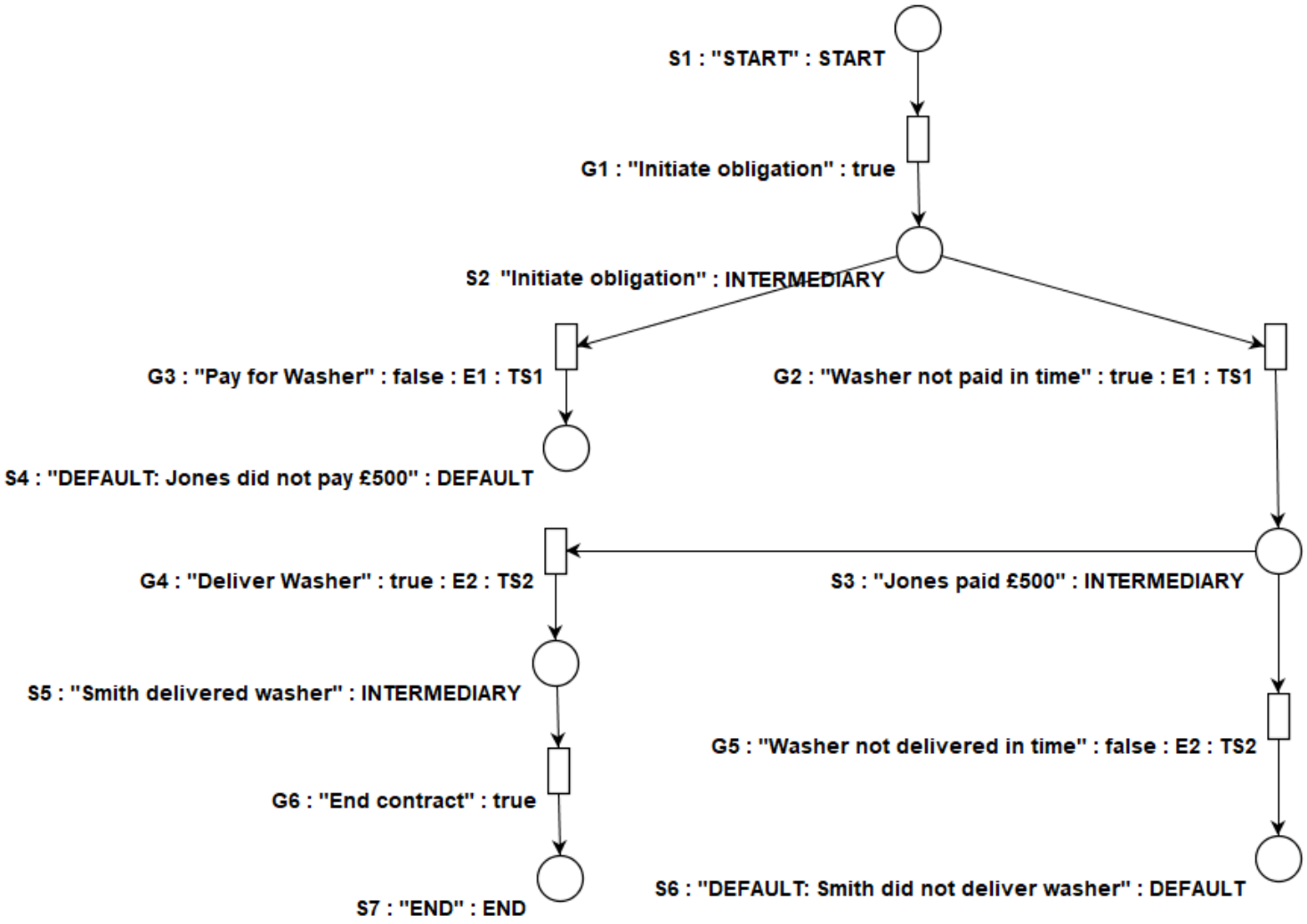


Figure A.1: Product Purchase Contract: Petri-net Diagram

SCE Simulations

SCE is not limited to showing Petri-net diagrams, but it can also be used to run simulations on the contract.

The first scenario to be run is the following: If the washer was paid on 02/05/1987 and delivered on 10/05/1987 and if the year is now 2018, this information is introduced into the software in the following format: $(E1, 1987-05-02), (E2, 1987-05-10), (NOW, 2018)$ where the events E1 and E2 correspond to paying and delivering the washer. Once this is introduced in the console of SCE, the software computes the result in Figure A.2. The states now coloured blue are states that have been reached. Also, all the introduced formal components are shown in the left side of the contract. In the right side, above the console, the simulator confirms that that contract has ended after reached the End state.

Another scenario to be run is the payment being done in time but the delivery being too late, on the 18/05/1987. By introducing the following information in the console $(E1, 1987-05-02), (E2, 1987-05-18), (NOW, 2018)$, SCE computes the result in Figure A.3. The screen above the terminal now shows: *'Contract default at the following state: S6: "DEFAULT: Smith did not deliver washer'*. Also, as a default state was reached, that is now coloured red.

The last simulation to be run is the payment one in time, the delivery not yet been completed and the current date being before the delivery deadline: $(E1, 1987-05-02), (NOW, 1987-05-12)$. SCE provides the result in Figure A.4. This time the contract has not ended and there is no default. The contract is pending for the following action at state S3. SCE indicats the following in the results screen: *'Action: "DELIVER" pending to be completed by "Smith" by 1987-05-15 at 00:00:00.000 '*.

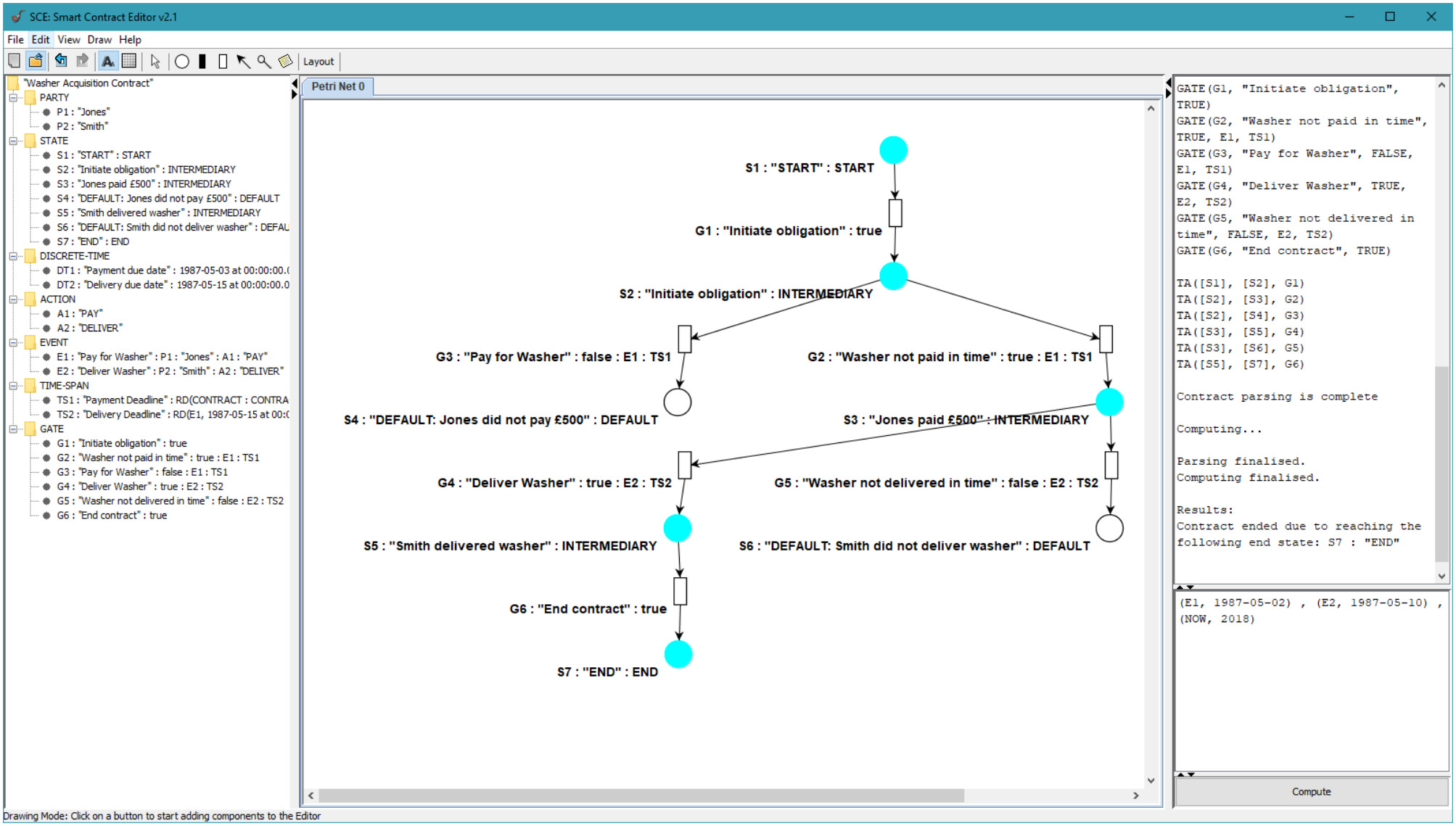


Figure A.2: Product Purchase Contract: Simulation 1 - of Successful Payment and Delivery

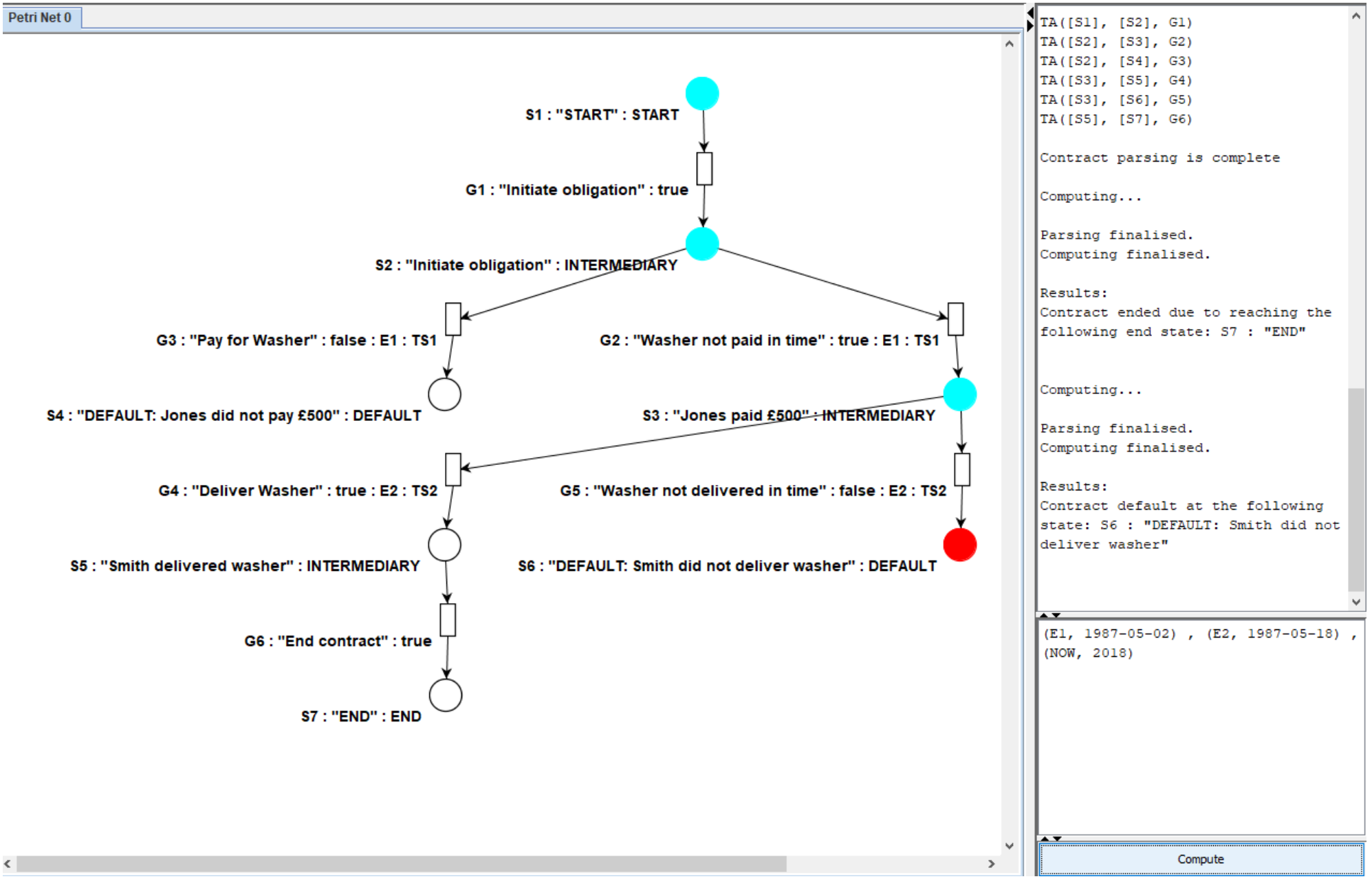


Figure A.3: Product Purchase Contract: Simulation 2 - Successful Payment, Failed Delivery

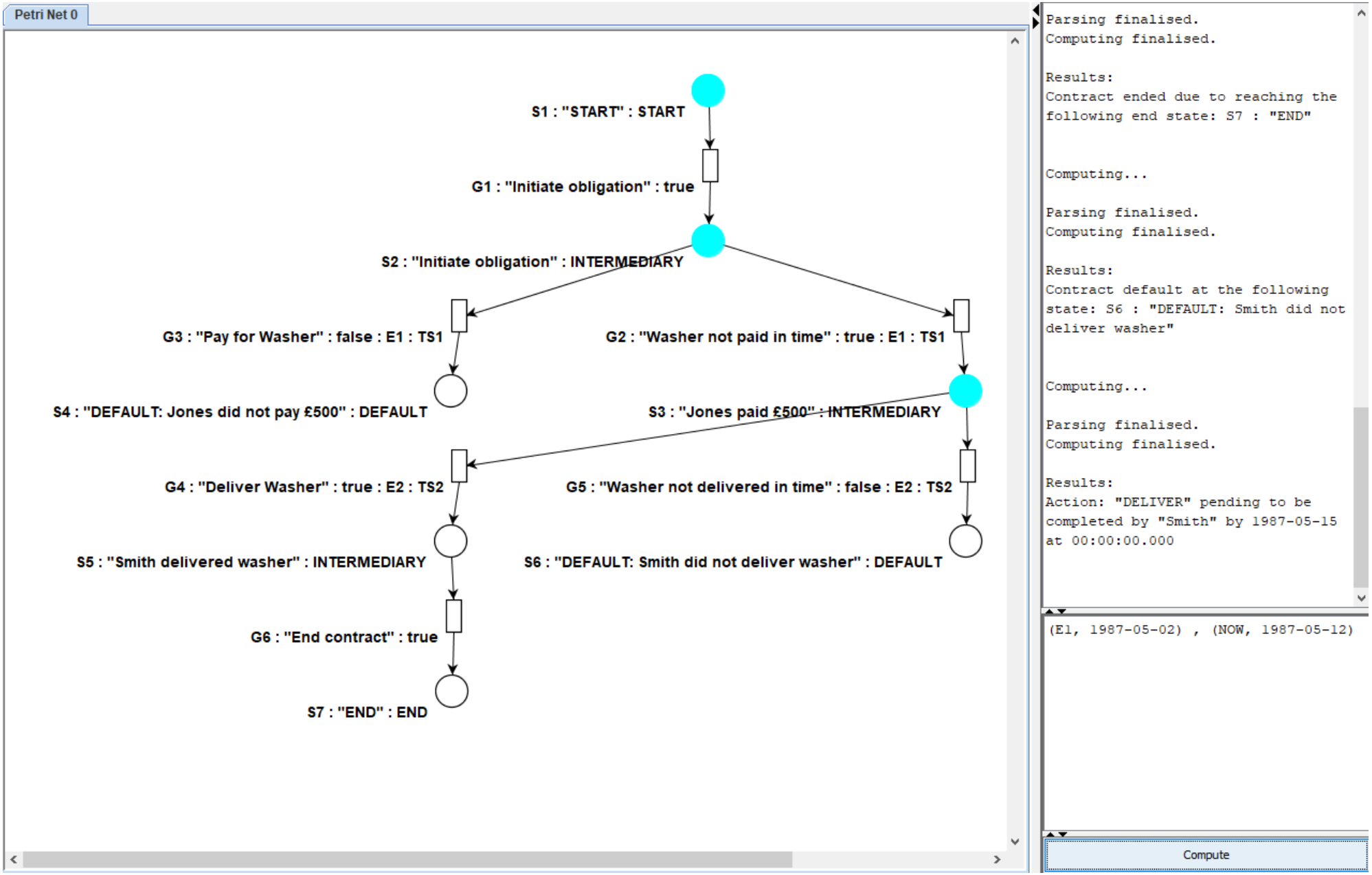


Figure A.4: Product Purchase Contract: Simulation 3 - Successful Payment, Awaiting Delivery

The Loan Lending Agreement

This subsection will provide an example of a more complex legal agreement between two corporations for the lending of a large loan over a period of years in which only interest in payable in instalments and the sum of money is repaid in its entirety at the end of the period. There are legal sanctioning mechanism for early payments and for missed payments. The full legal text was created by using the legal agreement templates provided for free by Law Depot UK[39] and is provided on the following pages:

LOAN AGREEMENT

THIS LOAN AGREEMENT (this "Agreement") dated this 1th day of April, 2018

BETWEEN:

London Bank PLC of 1 Triton Square, London
(the "Lender")

OF THE FIRST PART

AND

British Semiconductors PLC of 2 Triton Square, London
(the "Borrower")

OF THE SECOND PART

IN CONSIDERATION OF the Lender loaning certain monies (the "Loan") to the Borrower, and the Borrower repaying the Loan to the Lender, both parties agree to keep, perform and fulfil the promises and conditions set out in this Agreement:

Loan Amount & Interest

1. The Lender promises to loan £10,000,000.00 GBP to the Borrower and the Borrower promises to repay this principal amount to the Lender, with interest payable on the unpaid principal at the rate of 3.00 percent per annum, calculated yearly not in advance, beginning on 6 April 2018.

Payment

2. This Loan will be repaid in consecutive yearly instalments of interest only commencing on 6 April 2018 and continuing on the 6th of April of each following year until 5 April 2023 with the balance then owing under this Agreement being paid at that time.

Default

3. Notwithstanding anything to the contrary in this Agreement, if the Borrower defaults in the performance of any obligation under this Agreement, then the Lender may declare the principal amount owing and interest due under this Agreement at that time to be immediately due and payable.

4. Further, if the Lender declares the principal amount owing under this Agreement to be immediately due and payable, and the Borrower fails to provide full payment, interest at the rate of 6.00 percent per annum, calculated yearly not in advance, will be charged on the outstanding amount, commencing the day the principal amount is declared due and payable, until full payment is received by the Lender.

Sanctions

5. If the Borrower repays the loan early, they will be charged a fee equivalent to the remaining interest to be paid until the agreed date of the repayment.

6. In the case in which the Borrower is late with an interest repayment, the interest at the rate of 6.00 percent per annum, calculated yearly not in advance, will be charged on the outstanding amount, commencing with the day until which that said payment should have been made.

7. In the case in which the Borrower is late with the repayment of the loan, they will be charged a fee of £50,000.00 GBP.

Governing Law

8. This Agreement will be construed in accordance with and governed by the laws of the Country of England.

Costs

9. All costs, expenses and expenditures including, without limitation, the complete legal costs incurred by enforcing this Agreement as a result of any default by the Borrower, will be added to the principal then outstanding and will immediately be paid by the Borrower.

Binding Effect

10. This Agreement will pass to the benefit of and be binding upon the respective heirs, executors, administrators, successors and permitted assigns of the Borrower and Lender. The Borrower waives presentment for payment, notice of non-payment, protest, and notice of protest.

Amendments

11. This Agreement may only be amended or modified by a written instrument executed by both the Borrower and the Lender.

Severability

12. The clauses and paragraphs contained in this Agreement are intended to be read and construed independently of each other. If any term, covenant, condition or provision of this Agreement is held by a court of competent jurisdiction to be invalid, void or unenforceable, it is the parties' intent that such provision be reduced in scope by the court only to the extent deemed necessary by that court to render the provision reasonable and enforceable and the remainder of the provisions of this Agreement will in no way be affected, impaired or invalidated as a result.

General Provisions

13. Headings are inserted for the convenience of the parties only and are not to be considered when interpreting this Agreement. Words in the singular mean and include the plural and vice versa. Words in the masculine mean and include the feminine and vice versa.

Entire Agreement

14. This Agreement constitutes the entire agreement between the parties and there are no further items or provisions, either oral or otherwise.

IN WITNESS WHEREOF, the parties have duly affixed their signatures on this 6th day of April, 2018.

SIGNED, SEALED AND DELIVERED
before me, this 1th day of April, 2018

London Bank PLC

per: _____(SEAL)

SIGNED, SEALED AND DELIVERED
before me, this 1th day of April, 2018

British Semiconductors PLC

per: _____(SEAL)

Trans-Assertion (TA) Notation

As Lee's and Pithadia's formalisms do no offer support for action tracking, there is no way to represent the legal text in trans-assertion notation without changing its meaning. As such, this contract can only be expressed in TA notation using the formalism brought forward by this dissertation:

$$CONTRACT(\text{"}Loan\quad Agreement\quad Contract\text{"},\quad 2018-04-01,\quad 2023-04-05)$$

$$PARTY(P1,\text{"}London\quad Bank\quad PLC\text{"})$$
$$PARTY(P2,\text{"}British\quad Semiconductors\quad PLC\text{"})$$

$$STATE(S1,'START',START)$$
$$STATE(S2,'Sum\quad Lent',INTERMEDIARY)$$
$$STATE(S3,'Termination\quad due\quad to\quad lending\quad failure',END)$$
$$STATE(S4,'Installement\quad Not\quad on\quad Time',INTERMEDIARY)$$
$$STATE(S5,'Notification\quad of\quad Late\quad Installement',INTERMEDIARY)$$
$$STATE(S6,'Default\quad on\quad Installement',DEFAULT)$$
$$STATE(S7,'Interest\quad Rate\quad Raised',INTERMEDIARY)$$
$$STATE(S8,'Debt\quad paid',INTERMEDIARY)$$
$$STATE(S9,'Installement\quad Paid',INTERMEDIARY)$$
$$STATE(S10,'All\quad Installements\quad Paid',INTERMEDIARY)$$
$$STATE(S11,'Succesful\quad Termination',END)$$
$$STATE(S12,'Debt\quad Not\quad on\quad Time',INTERMEDIARY)$$
$$STATE(S13,'Default\quad on\quad Debt\quad Repayment',DEFAULT)$$
$$STATE(S14,`Start\quad debt\quad repayment\quad branch',INTERMEDIARY)$$
$$STATE(S15,`Start\quad installments\quad payment\quad branch',INTERMEDIARY)$$

$$DISCRETE-TIME(DT1,\text{"}Sum\quad lending\quad due\quad date\text{"},2018-04-06)$$
$$DISCRETE-TIME(DT2,\text{"}Installement\quad start\quad period\text{"},SCH(2018,2022,1,[APR],[6]))$$

$DISCRETE-TIME(DT3,"Installement\quad end\quad period",SCH(2019,2023,1,[APR],[5]))$

$DISCRETE-TIME(DT4,"Installement\quad Not\quad Paid\quad Notification\quad Deadline\quad 5\quad days",$
$AFT(S5,0000,00,05))$

$DISCRETE-TIME(DT5,"Installement\quad Late\quad Payment\quad Deadline\quad 10\quad days",AFT(S5,0000,00,10))$

$DISCRETE-TIME(DT6,"Debt\quad Late\quad Payment\quad Deadline\quad 15\quad days",AFT(S12,0000,00,15))$

$SYS-P(SYS-P1,"Sum\quad to\quad lend",10000000)$

$SYS-P(SYS-P2,"Sum\quad to\quad repay",SYS-P1)$

$SYS-P(SYS-P3,"Yearly\quad Installement",SYS-P2)$

$SYS-P(SYS-P4,"Installments\quad due",5)$

$SYS-P(SYS-P5,"Interest\quad rate",0.03)$

$SYS-P(SYS-P6,"Penalty\quad Interest\quad rate",0.06)$

$SYS-P(SYS-P7,"Fine",50000)$

$SYS-E(SYS-E1,"all\quad installements\quad paid",SYS-P4,0)$

$SYS-U(SYS-U1,"Set\quad yearly\quad Installement\quad value",SYS-P3,MULT,SYS-P5)$

$SYS-U(SYS-U2,"Decrease\quad number\quad of\quad installements\quad due",SYS-P4,SUB,1)$

$SYS-U(SYS-U3,"Raise\quad interest\quad rate",SYS-P5,ATR,SYS-P6)$

$SYS-U(SYS-U4,"Reset\quad yearly\quad Installement\quad value",SYS-P3,ATR,SYS-P2)$

$SYS-U(SYS-U5,"Apply\quad fine",SYS-2,ADD,SYS-P7)$

$ACTION(A1,"Lend\quad sum",P2,SYS-P1,GBP)$

$ACTION(A2,"Pay\quad Yearly\quad Installement",P1,SYS-P3,GBP)$

$ACTION(A3,"Notify\quad of\quad failureto\quad pay\quad installement\quad on\quad time",P2)$

$ACTION(A4,"Pay\quad sum\quad due",P1,SYS-P2,GBP)$

$EVENT(E1,"Sum\quad lent",P1,A1)$

$EVENT(E2,"Yearly\ \ Installement\ \ Paid",P2,A2)$

$EVENT(E3,"Notification\ \ of\ \ failure\ \ to\ \ pay\ \ installement\ \ on\ \ time\ \ dispatched",P1,A3)$

$EVENT(E4,"Sum\ \ due\ \ fully\ \ paid",P2,A4)$

$TIME-SPAN(TS1,"Sum\ \ Lending\ \ Timespan",RD(CONTRACT_START,DT1))$

$TIME-SPAN(TS2,"Installement\ \ Payment\ \ Timespan",RD(DT2,DT3))$

$TIME-SPAN(TS3,"Installement\ \ NotPaid\ \ Notification\ \ Timespan",RD(S4,DT4))$

$TIME-SPAN(TS4,"Installement\ \ Late\ \ Payment\ \ Timespan",RD(S5,DT5))$

$TIME-SPAN(TS5,"Debt\ \ Late\ \ Payment\ \ Timespan",RD(S12,DT6))$

$TIME-SPAN(TS6,"Contract\ \ duration",RD(CONTRACT\_START,\ \ CONTRACT\_TERMINATION))$

$GATE(G1,"Sum\ \ lent",TRUE,E1,TS1,SYS-U1)$

$GATE(G2,"Sum\ \ not\ \ lent",FALSE,E1,TS1)$

$GATE(G3,"Installement\ \ Paid",TRUE,E2,TS2,SYS-U2)$

$GATE(G4,"Installement\ \ Not\ \ Yet\ \ Paid",FALSE,E2,TS2)$

$GATE(G5,"Notify\ \ about\ \ installement\ \ not\ \ paid",TRUE,E3,TS3,SYS-U1)$

$GATE(G6,"Recalculate\ \ yearly\ \ installements",TRUE,SYS-U4)$

$GATE(G7,"Late\ \ Installement\ \ Not\ \ Yet\ \ Paid",FALSE,E2,TS4)$

$GATE(G8,"Late\ \ Installement\ \ Paid",TRUE,E2,TS4,SYS-U2)$

$GATE(G9,"Debt\ \ Fully\ \ Paid",TRUE,E4,TS6)$

$GATE(G10,"All\ \ Installements\ \ Paid",TRUE,SYS-E1)$

$GATE(G11,"Installements\ \ Stil\ \ toPay",FALSE,SYS-E1)$

$GATE(G12,"Debt\ \ Not\ \ Yet\ \ Paid",FALSE,E4,TS6,SYS-U5)$

$GATE(G13,"Debt\ \ Fully\ \ Paid\ \ Late",TRUE,E4,TS5)$

$GATE(G14,"Debt\ \ Payment\ \ Failure",FALSE,E4,TS5)$

$GATE(G15,"Aggregate\ \ Debt\ \ Paid\ \ and\ \ Installements\ \ Paid",TRUE)$

$GATE(G16,"Initiateconcurrency",TRUE)$

$TA([S1],[S2],G1)$	$TA([S15],[S4],G4)$	$TA([S5],[S9],G8)$
$TA([S1],[S3],G2)$	$TA([S4],[S7],G6)$	$TA([S14],[S8],G9)$
$TA([S2],[S14,S15],G16)$	$TA([S8,S10],[S11],G15)$	
$TA([S15],[S9],G3)$	$TA([S4],[S7],G6)$	$TA([S14],[S12],G12)$
$TA([S9],[S15],G11)$	$TA([S7],[S5],G5)$	$TA([S12],[S8],G13)$
$TA([S9],[S10],G10)$	$TA([S5],[S6],G7)$	$TA([S12],[S13],G14)$

Petri-net Diagram Representation via SCE

By introducing the presented legal agreement in SCE, using the trans-assertion notation, the Petri-net diagram at Figure A.5 is obtained. Despite this contract being much more complex than previous examples, this proves how scalable the formalism is and how it is able to easily deal with contracts of any sizes.

It should be noted that this contract is the first example which contains two different end states representing two various ways of ending the contract: the first, S_3, stands for the loan being refused, whilst the other, S_{11}, stands for the interest instalments being fully paid, represented by S_{10} as well as the initial debt being fully paid, represented by S_8. This is a very good example of how concurrency can be used, by splitting the contract at S_2 in two branches: one that takes care of the debt repayment and another one which deals with the interest (instalments) payments. The two branches, provided there was no default along the way, are then aggregated through the immediate true gate G_{15}. As such, the example makes use of gate conjunction, gate disjunction and state mutually exclusive disjunction, as they were described in Chapter 3 and in Section 4.4.

This contract is also the first one in this paper that gives example of sanctions in the case of late interest payment which is essentialy a contrary-to-duty obligation, as dicussed in Subsection 2.2.3.4,and of providing additional time for an obligation to be completed (in the case of late debt repayment). The late instalments sanction requires some additional attention as it actually changes the terms of the repayment by increasing the interest rate. Due to using the system parameter mechanisms, it is without difficulty to implement the specific clauses for the sanction.

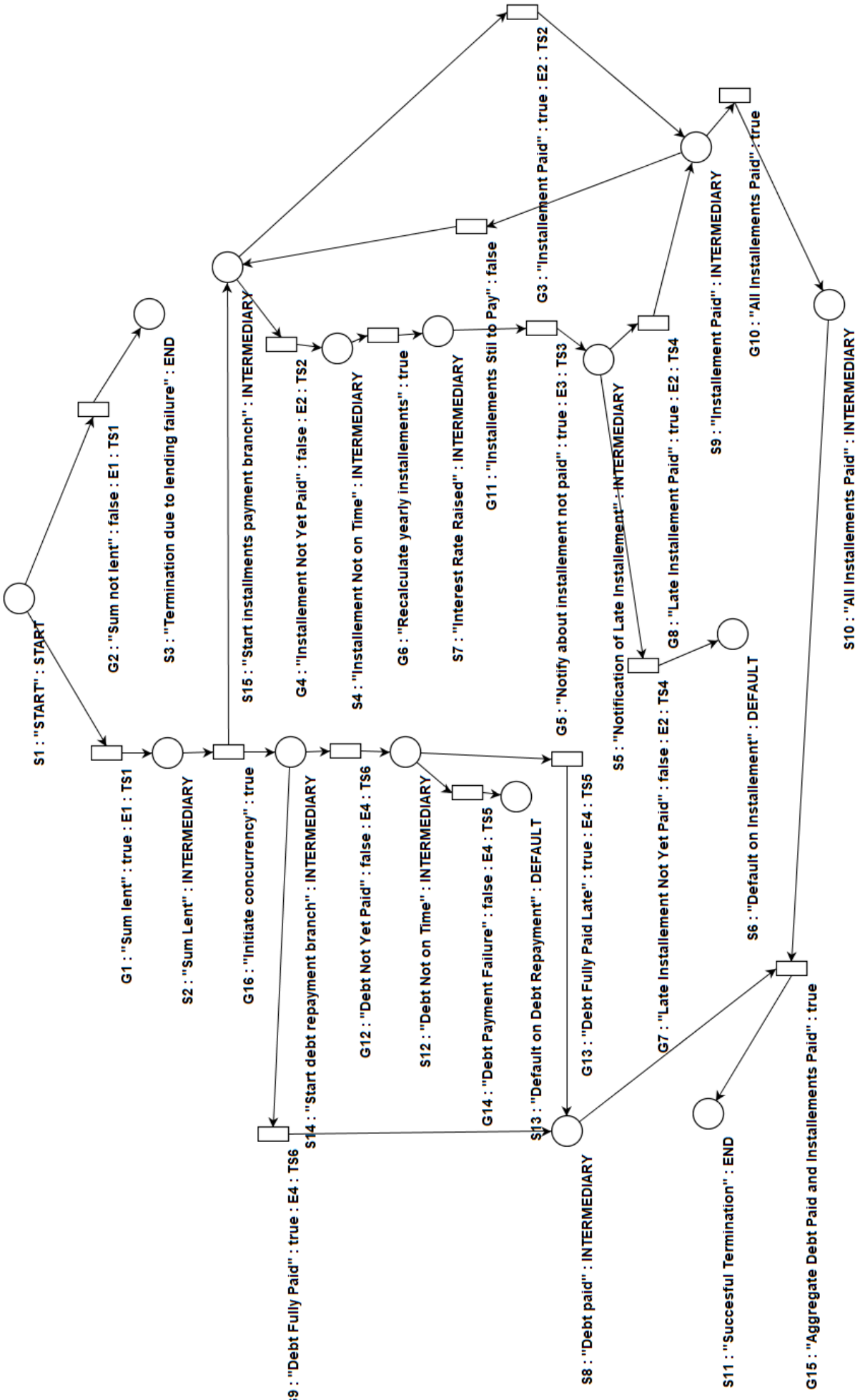


Figure A.5: Petri Net Diagram of the Loan Agreement Contract

B | Appendix B - Smart Contract Editor

Smart Contract Editor (SCE) is the software tool provided with this project with the purpose of validating and testing the syntax and the formalism. It has three main functions:

1. Constructing an internal model of the contract by parsing the agreement introduced in the form of Petri-net notation; this model is viewable in the left panel of the software.
2. Drawing a Petri-Net diagram that contains the states, gates and arcs in the TA notation.
3. Running simulations on the contracts introduced.

The SCE software was developed through a re-purposing of the Platform Independent Petri Net Editor (PIPE) software (Source available at [40]). PIPE is an open source Java tool used for the manual creation of Petri nets and developed between 2002 and 2007 as part of four MSc dissertaions [1] and continued between 2007 and 2016 as part of multiple research projects as well as the subject of two scientific publications: [41] and [42]. As PIPE is built exclusively with the purpose of research, it was considered to be the best choice for re-purposing it as a Petri-net builder.

The SCE source code and the compiled JAR file is provided with this project and they are also available on a public GitHub repository at https://github.com/gabrielv33/Smart-Contract-Editor.

B.1 Features and Limitations

The PIPE development was abandoned in a beta stage with many of the advanced features of the software such as Petri-net analysis not working. As these were not the purpose of the software tool anyway, they were deactivated. The only components that were kept from PIPE are the drawing component and the user interface which has, however, been heavily modified. The user interface now includes three additional components: a panel containing components list on the left hand side of the window, a text panel that provides various information on the right side and a console underneath which is to be used for contract simulations. The diagram panel is placed in the center of the frame.

Besides the redesigned user interface, the following main components have been added: the Parser, the Tree Viewer, the Constructor and the Simulator.

[1] Available at http://pipe2.sourceforge.net/docs.html

The Parser component loads and reads the input file, it converts it to a standard format, it uses Regex pattern matching to extract information from each line of input and then it passes the parameters of each component to the relevant constructor method, creating and initialising the objects; each component is represented by a different object of a relevant class (e.g. State class, Gate class etc). Once these objects have been created, they are loaded in the internal ArrayList of a Contract class instance. Finally, each component gets connected to the relevant other components (e.g. a gate with its time-span or event object) and it construct a virtual model of the contract stored in a graph structure where each component is a node.

The Tree Viewer component displays all the components of a contract in a tree like structure in the panel that is located in the left hand side of the screen.

The Constructor component builds the Petri-net diagram from the given virtual model. This component uses parts of the PIPE software in order to draw the Petri-net software on the screen.

The Simulator component is the component responsible for running simulations (or queries) on the contract. It loads a list of tuples that represent events completed and the time they were completed as well as one parameter that represents the current time. It then runs a breath first search (BFS) algorithm on the internal graph and it opens gates if the relevant event has happened or not before the given deadline (i.e. in the specific time-span). Once that is completed, it displays the results in two ways: via the console - presents whether the BFS reached a default, end or pause state or, in the case of intermediary states, the real world action that needs to be completed for the contract to advance to the following state - and graphically, it marks the states that were reached on the Petri-net by filling the figure with various colours (some types of states, such as default, are coloured in a different colour to mark an irregular behaviour or an immediate action to be taken).

Due to the limited amount of time available for development of this tool, SCE has a few limitations in the way it represents the formalism presented in the paper. These limitations are the following:

- The schedule time operator (SCH) is not supported.
- The use of system parameters is not supported.
- Templates are not supported.
- The parsing of the input is top-down which means that each component used must have been defined previously. (Similar to C/C++ top-down parsing.)

As the Petri-net diagrams only contain the states, gates and trans-assertions, SCE can draw a contract that uses SCH or system-parameters, but it will not be able to run simulations on it.

It should also be noted that, even though the software allows the graphic editing of the Petri net diagram, the changes made on the diagram are temporary and they will not change the internal structure of the contract model. The feature of allowing the editing of contracts via the diagram was not planned for this project, but it could be developed in the future.

B.2 User Manual

SCE is an open-source multi-platform software and it will run on any system that supports Java. The development and testing was mainly done on a system running the Windows 10 operating system. It is highly recommended to have the latest version of the Java Runtime Environment installed. There is no installation required as SCE is a portable software tool. To run the software, one only needs to run the JAR file named "PetriNet Editor.jar" provided with this project. The file uses approximately 40 MBs of disk space.

As SCE is heavily computing based, the amount of computation power and of RAM memory necessary will depend on the size of the contracts and on the number of tabs open. With the examples provided in this paper and with 10 contracts open, the RAM usage will typically vary between 150MB and 250MB.

As the main purpose of the software tool is parsing, computing and drawing, the user interaction is purposely made very simple. First step to get the program operating is to load the file that contains the contract in the form of trans-assertion notation. This can be done by using the 'Open' button on the toolbar or the one in the File tab or by pressing 'CTRL+O' on Windows. There are two example contracts provided in separate files with this paper: the product purchase contract and the loan lending agreement. The example files are saved in '.txt' format, but they can be saved in any text format.

When writing files, a note should be made of the following properties:

1. The types of the components are not case sensitive and they can be used by their designated abbreviation (e.g. S for state, TA for trans-assertion). The names and the parameters of the components are case sensitive.
2. Any of the following symbols can be used around the name of components: ' ′ ".
3. The parsing is done top-down so a component has to be defined before it is used.

4. Empty lines will be ignored. All lines that do not contain a pair of parentheses will be ignored.
5. Only one component should be defined on a line. Everything defined after the first component on a line will be ignored
6. Comments can be made by using ‘ // ’ before one line comments or ‘ /* ’ and ‘ */ ’ around multi-line comments.

Once the contract is loaded, the parsing will commence automatically. Updates will be provided in the right panel. If the panel does not display the full contract followed by ‘Contract parsing is complete’, that means an error has occurred. SCE will provide scarce information about errors. Once the contract is loaded and the diagram drawn, graphical components can be moved, deleted, edited, but these will have no effect on the internal model of the contract.

Multiple contracts can be loaded at once. A new tab will open for each contract. The panels can be minimised. SCE is scaling to the resolution of the display and has been tested on multiple monitors with various sizes and resolutions. The buttons on the tool bar above the panels are almost identically the same buttons as the ones in the various button tabs (e.g. File, Edit). For most buttons there is a keyboard short-cut available (described in the button tabs).

The available buttons are the following: New (creates an empty tab), Open, Undo, Redo, Cycle text size (will cycle the font size between 8, 10, 12, 14, 16 and 20 with the default being 12; the diagram will re-arrange itself when the font size is modified), Cycle grid size (the grid on which the diagram is drawn on), Select (the selection mode must be on when moving graphical components). These are followed by a set of buttons for creating various Petri-net components: states, immediate transitions, timed transitions, arc, inhibitor arc and note. The last button, Layout, rearranges the Petri-net diagram in top-down format by taking into consideration the size of the graphical components and the size of the text.

To run simulations, a list of tuples containing the events that occurred and the current time must be provided in the console located in the bottom right corner of the screen. E.g. $(E1, 2002), (E2, 2003-06-20 \quad at \quad 08:30), (E2, 2005-08-24 \quad at \quad 09:30:24.773), (NOW, \quad 2018-09-01)$. The list of events can be empty, but the current time must always be provided or the following error will be displayed in the right panel above the console: ‘Error: Current time now found. Computation cancelled.’. Once the computation is finalised, the results will be drawn on the diagram by marking states that were reached (default states will be marked with a different colour) as well a shortly present

in the right panel above the console.

B.3 Software Design. System Manual

The SCE software tool contains 3,337 pages of computer code (including unit testing files), from which over 130 pages have been written for this dissertation alone. As mentioned previously, PIPE, the software on which SCE is built, was the main topic of two research papers and four MSc dissertations. Therefore, even though many features from the old software have been deactivated, it is well beyond the purpose of this paper to provide a detailed explanation of the way SCE is operating. However, this section will provide a short explanation of the UCL (sub)package which contains most of the important classes written for this project (93 pages).

Finally, this section will provide a UML diagram that shows the way the classes for the contract components (e.g. states, gates etc) are constructed. This is highly relevant to the project as the inheritance and the interfaces used strongly reflect many of the formalism definitions, rules and properties introduced in this dissertation.

The InputLineParser class is responsible for the loading of the input file which is stored as InputLine objects that are passed to the LineParser which is the class responsible for normalising the lines (normalising quotes, removing white spaces etc.), removing the comments and of actually parsing the lines. The Regex code for patern matching these is very advanced.

In order to remove all white spaces except the ones inside quotes and not without recognising escaped quotes, SCE use the following[43]:

```
inputString = inputString.replaceAll("\\s+(?=((\\\\[\\\\\"]|[^\\\\\"
])*\"(\\\\[\\\\\"]|[^\\\\\"])*\")*(\\\\[\\\\\"]|[^\\\\\"])*$)", "");
```

Similarly, in order to remove all comments, both single line and multiple line, SCE uses the following[44]:

```
inputString = inputString.replaceAll("(/\\*([^*]|[\\r\\n]
|(\\*+([^*/]|[\\r\\n])))*\\*+/)|(//.*)", "");
```

In order to normalise the quotes, SCE uses the following:

```
inputString = inputString.replaceAll("'", "\"");
```

```
inputString = inputString.replaceAll("‘", "\"");
```

Finally, in order to be able to split each line of the input file into parameters, SCE uses Regex to split the string by commas into an array of strings. The issue is that the splitting must be ignore text and other operators and operations. As such, the splitting by commas must ignore the content between quotes, parentheses, brackets and braces. For this, SCE uses the following patern matching (based on [45] and extended to ignore parentheses, brackets and braces as well):

```
String Split_otherThanQuote = " [^\"] ";
String Split_quotedString = format(" \" %s* \" ", Split_otherThanQuote);
String Split_regex = format("(?x) " + // enable comments, ignore white spaces
",                  " + // match a comma
"(?=                " + // start positive look ahead
"  (?:              " + // start non-capturing group 1
"    %s*            " + // match 'Split_otherThanQuote' zero or more times
"    %s             " + // match 'Split_quotedString'
"  )*               " + // end group 1 and repeat it zero or more times
"  %s*              " + // match 'Split_otherThanQuote'
"  $                " + // match the end of the string
")                  " + // stop positive look ahead
"(?![^\\(]*\\))"  + // ignore commas in parentheses
"(?![^\\[]*\\])"  + // ignore commas in brackets
"(?![^\\{]*\\})", // ignore commas in braces
Split_otherThanQuote, Split_quotedString, Split_otherThanQuote);

String[] tokens = stringToSplit.split(Split_regex, -1);
```

The parsed lines are converted into lists of parameters which are then sent to the internal constructor of the relevant contract component class (e.g. StateElement, GateElement etc). The ConsoleManager is the class that parses the input from the console and that runs the actual simulations through a Breath First Search (BFS) algorithm.

StateType is an enumeration of the types of states: start, end, intermediary, pause and default. GraphicalRepresentation is an interface implemented by components that have attached graphical components, specifically by GateElemenent and StateElement. GetDiscreteTime is another interface which is implemented by the components and operators that can be time references, specifically: a time operator (AfterTime, BeforeTime, EarliestTime, LatestTime, ScheduleTime), a DiscreteTimeElement, a StateElement or an EventElement (see the time encapsulation property of states and events).

TimeOperator is an abstract class inherited by all the time operator classes and TimeSpanOperator is the same for all time-span operator classes (RDuring and RThroughout). ContractElement is also an abstract class inherited by all classes that represent a component of the formalism. Each component is then stored in an ArrayList located in an instance of the Contract class. It should be noted that TransAssertion is not inheriting ContractElement as trans-assertions do not need to be connected to other elements(they themselves describe the connections) and can be safely discarded after the arcs are extracted from them and stored in the StateElement and GateElement class instances.

Finally, the ContractTreeManager is responsible for managing the left panel where all the contract components are stored in the form of a tree-like structure and the ConsoleFrameManager is responsible for managing the right panel where the simulation results are displayed.

The SCE software was designed in such a way as to follow the formalism, rules and properties of the Vanca model introduced in this paper. The abstract classes, the inheritance, enumerations, interfaces prove that. This becomes obvious when looking at the UML diagram from Figure B.2 provided on the following page.

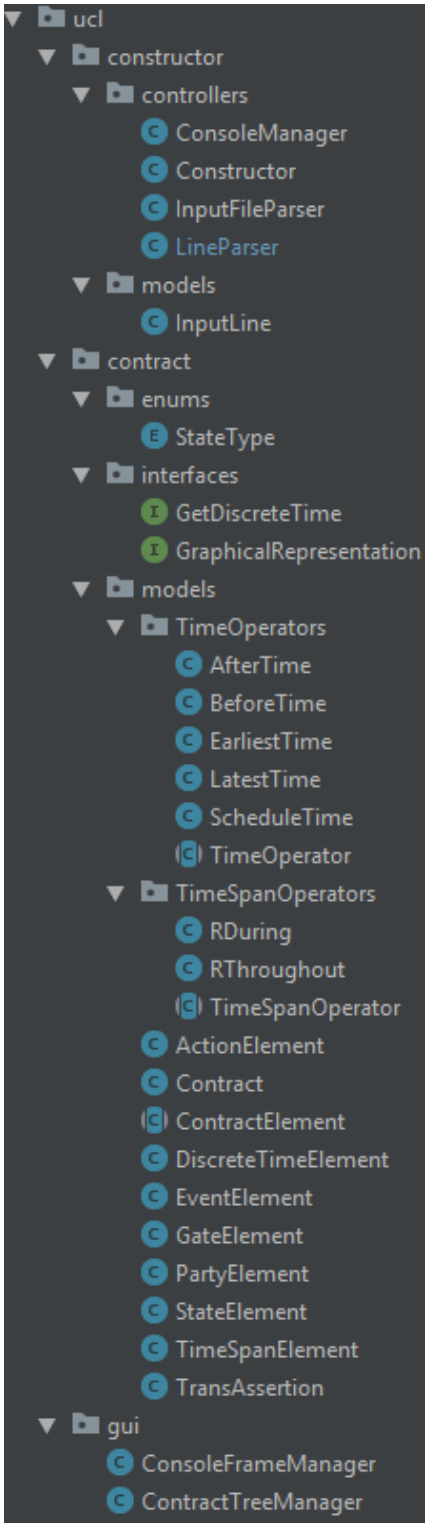


Figure B.1: SCE: The UCL Package Structure

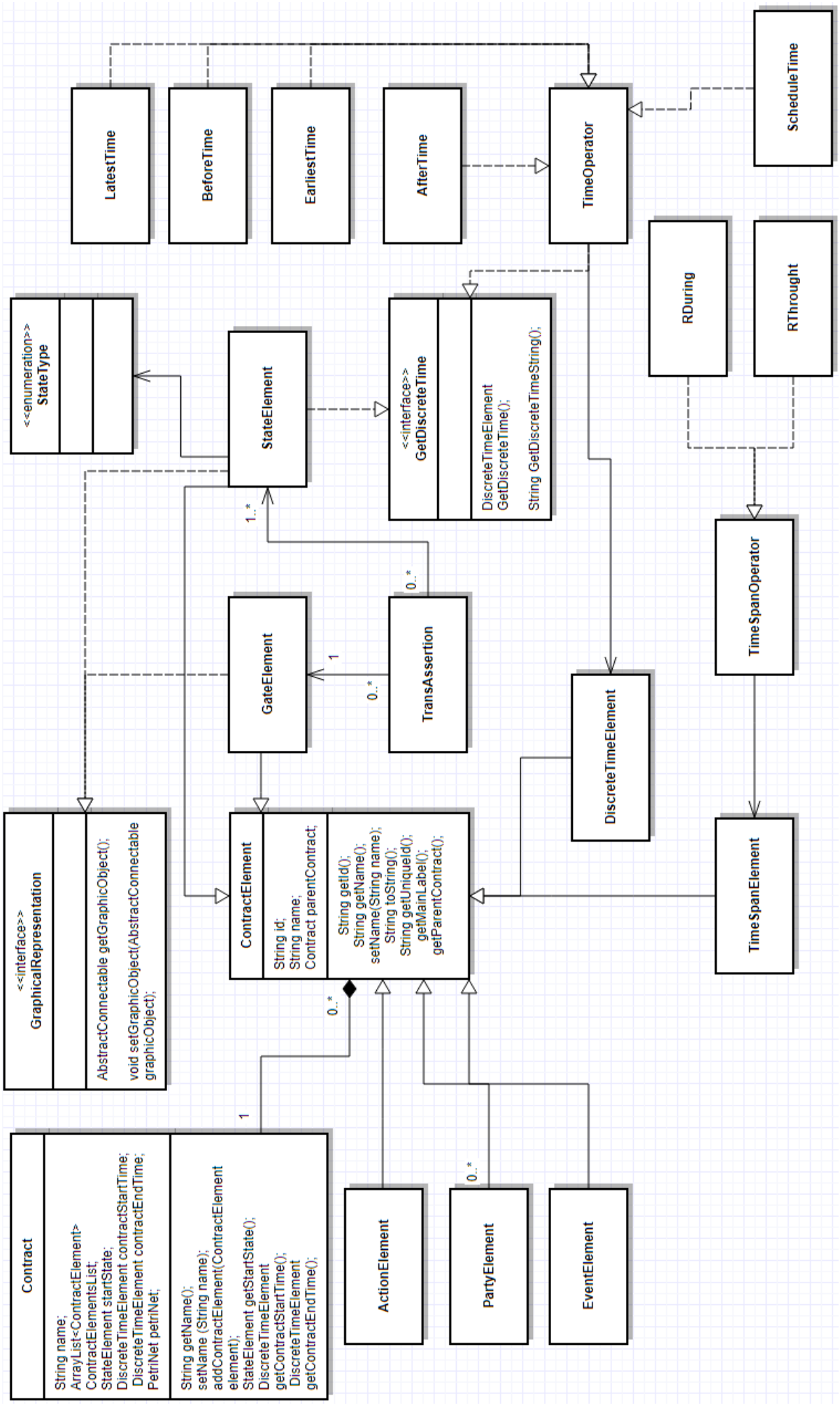


Figure B.2: SCE: UML Diagram

C | Appendix C - Project Planing

This section contains information about the project planning as per dissertation requirements. It includes the Project Plan of November 10, 2017 and the Interim Report of January 28, 2018.

The Semantics of Smart Contracts Used in Banking and Financial Services

Project Plan

Gabriel Vanca

Supervisor: Dr Chris Clack

Department of Computer Science

University College London

November 10, 2017

Contents

Chapter 1

Aims and Objectives

1.1 Aims

The main aim of the project is to investigate a viable semantic system for the construction of high-value, standardised smart contracts that can work to automate various procedures in banking and financial services, making them faster, error-proof, tamper-proof, cheaper to implement, easier to use and, in some cases, more accessible to customers.

1.2 Objectives

In the current project, the following objectives are planned to be accomplished:

1. Review the existing techniques and logic system(s) for representing smart contracts and the issues related to them.

2. The creation of a system for semantic analysis of smart contracts that can primarily be used across the financial and banking sectors.

3. A system that includes at least temporal, deontic and operational logics that take into consideration time related issues, rights, obligations and prohibitions of each involved party as well as actions and inactions of the contractual parties.

4. Design templates for more advanced procedures in the design (e.g. sanctions, permitting an action a limited and fixed number of times etc.)

If time permits, the following objectives are to be accomplished as well:

1. Develop an upgraded version of the trans-assertion notation that reflects the newly designed or extended semantic system in a format that is easily computable by software.

2. Define a concrete system of rules for correctly built smart contracts.

3. Develop software tools that can help visualise and evaluate the viability and performance of the proposed semantic system.

Chapter 2

Deliverables

The current project aims to deliver the following items:

1. A literature survey that summarises previous work in the area of smart-contracting, specifically on existing semantic systems for electronic contracting, as well as literature on correlation and combination of various logical systems that are representative for the conducted work.

2. An analytical discussion on the performance, drawbacks and any issues related to the existing semantic systems for smart-contracting.

3. Two fully documented and functional pieces of software.

    - An extension and repurposing of an existing program for the analysis of Petri Nets with added functionality that allows the creation of Petri Nets according to the developed logical model from a given contract under the form of a list of trans-assertions.
    - An evaluation program that analyses contracts under the form of a given list of trans-assertions and that assesses whether the contract is correctly built and, if not, what issues does the contract have (e.g. ambiguity, errors etc.)

4. A strategy for testing and evaluating the software deliverables

5. A critical conclusion of the strengths and weaknesses of the proposed approach.

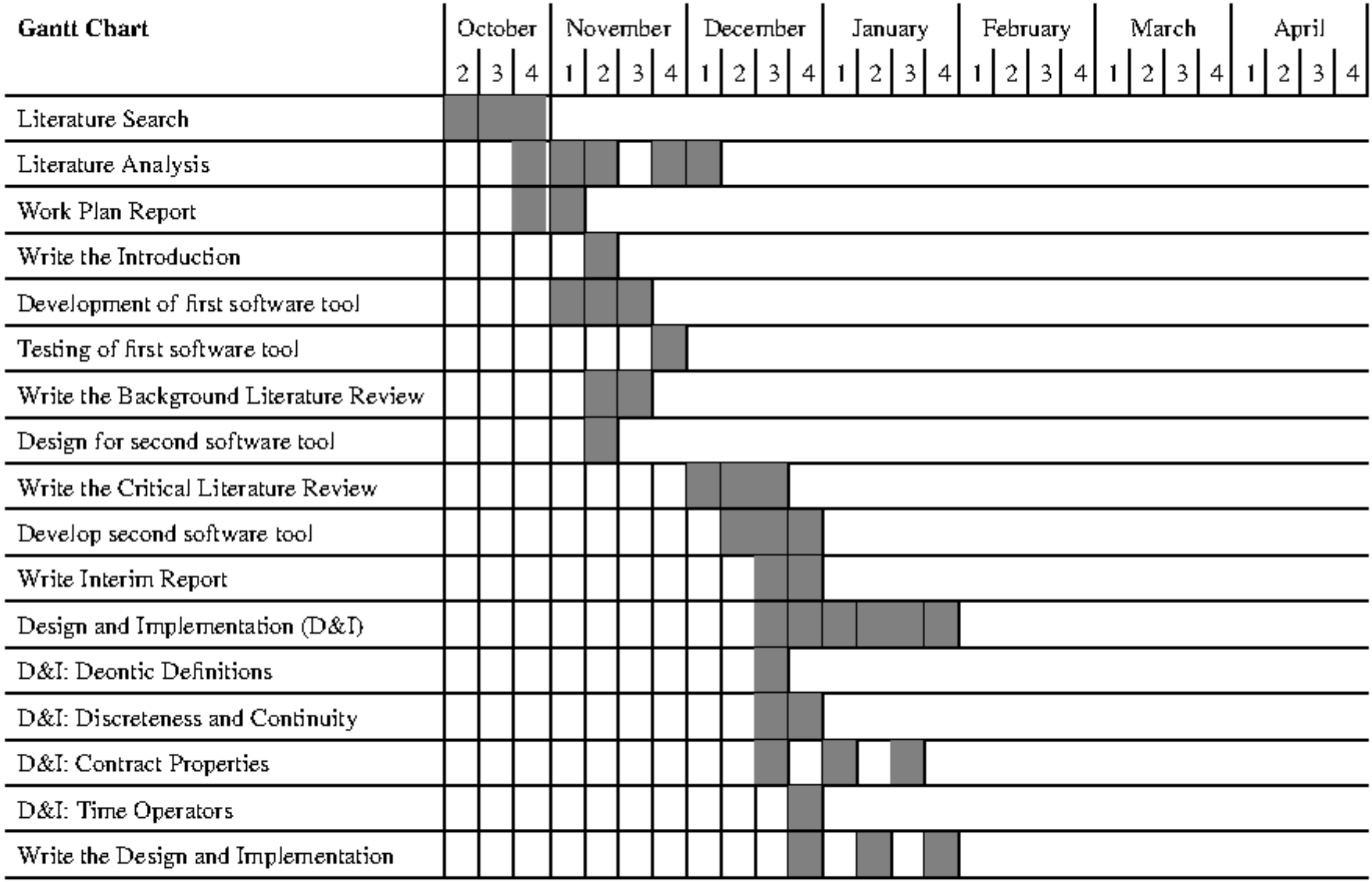
Chapter 3

Work Plan

The work plan is represented via a Gantt Chart. The chart covers the period between the 6th of October 2017, the deadline for the submission of the initial project idea, and the 30th of April 2018, the deadline for the final project report.

For the purposes of constructing the Gantt Chart, each month was divided in 4 quarters which would represent a period of 7-8 days each. Short tasks that take less than 1 week are not represented in the plan.

The chart is meant as a general indication of how the progress with the project is expected to occur, but checks, updates and adjustments will be made during the course of the project on a weekly basis.


Gantt Chart
October
November
December
January
February
March
April
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Literature Search
Literature Analysis
Work Plan Report
Write the Introduction
Development of first software tool
Testing of first software tool
Write the Background Literature Review
Design for second software tool
Write the Critical Literature Review
Develop second software tool
Write Interim Report
Design and Implementation (D&I)
D&I: Deontic Definitions
D&I: Discreteness and Continuity
D&I: Contract Properties
D&I: Time Operators
Write the Design and Implementation

8

	October			November				December				January				February				March				April			
	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
D&I: Conjunctions and Disjunctions																											
D&I: Concurrency																											
D&I: Trans-assertion Templates																											
Test second software tool																											
Testing and Evaluation																											
Write the Testing and Evaluation																											
Finish Report																											

Interim Report

Student Name: Gabriel Vanca

Project Title: The Semantics of Smart Contracts Used in Banking and Financial Services

(Note: The project title has not changed since the November Project Plan)

Internal Supervisor: Dr Christopher Clack

Current Status

At the current time, I am confident I have made sufficient progress with my project report, finishing the following report sections:

1. **Introductory Material** – This section is comprised of a basic description of the project and its motivations and objectives

2. **Background Research. Foundation Papers** – an in-depth literature review of the ongoing research in smart contracting for the past three decades with various explanations about semantic models and an explanation of the approach the paper is taking into the research.
    a. An introduction about smart contracts is made based on UCL's Dr Christopher Clack's papers
    b. The review of semantic modelling is looking at concepts such as deontic logic, temporal logic, obligations, permissions and prohibitions and the differences between them in semantic representation, various types of time references (e.g. continuous time, discrete time, relative time, absolute time, recurrent time).
    c. The paper also looks at the requirements of a good semantic model for smart contracting and why Lee's model was chosen.
    d. Finally, the paper looks at the challenges in constructing a good semantic model.

3. **Lee's Formalism. Syntax** – This is a short section that explains the notions used in Lee's formalism and all the notation used in his papers dating from 1989 all the way to 2012
    a. We look at Petri nets, graphical representations, internal representations (T-calculus), and Lee's trans-assertion notation

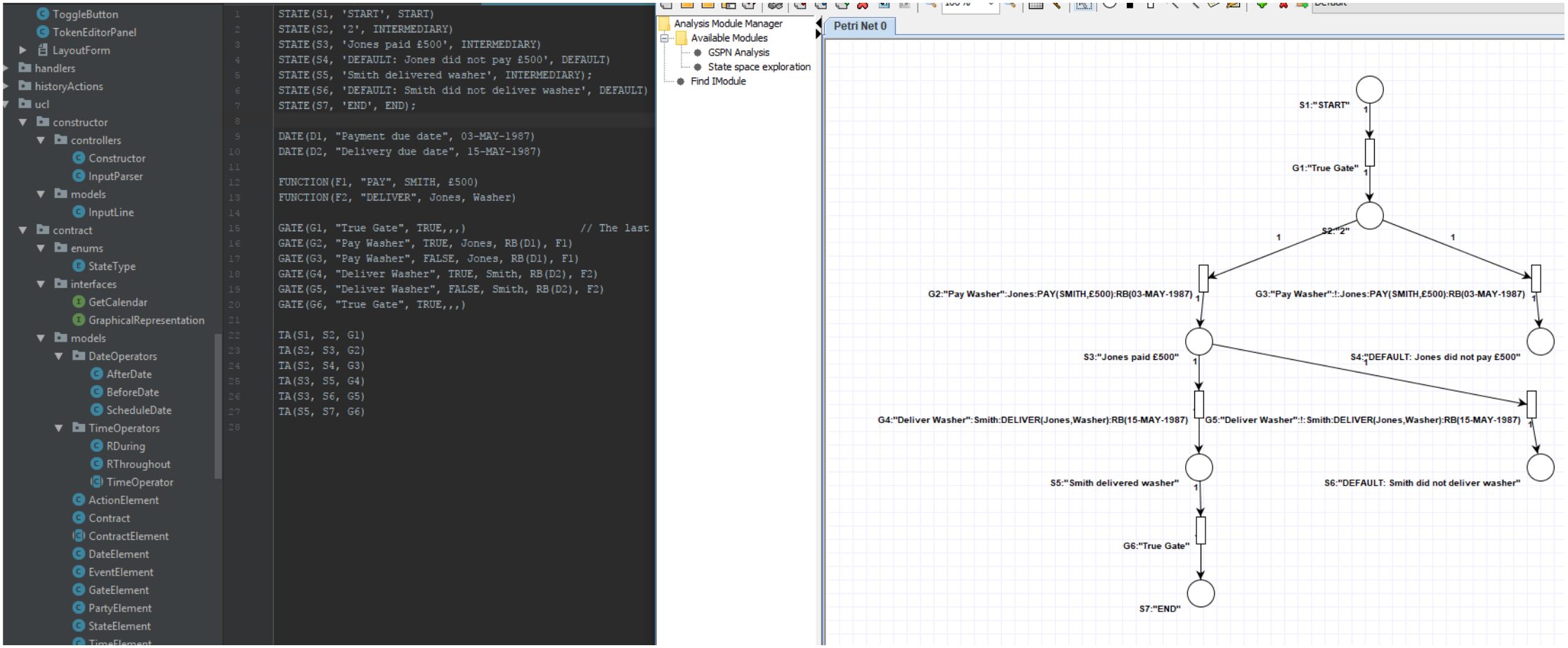
4. **Critical Analysis of Lee's Semantic Model** – This section is an in-depth critical analysis of Lee's original semantic model and of Hirsch's 2015 improvements. The analysis identifies certain important flows in the model, some of them by going back to the literature review from Section 2.
    a. There are two main subsections: Temporal Aspects and Deontic Aspects. We look at various elements and flaws, starting from issues with the notation and the basic operators, all the way to more profound issues such as various types of time

references (as described in Section 2), isomorphism problems, lack of canonical form etc.

I have also started working on the 5th Section – **Design and Implementation of a Revised Semantic Model** which contains the improvements I am proposing for Lee's model, basing myself on Hirsch's work as well as of all the authors in Section 2. Explanations in this section are therefore more straightforward as they are based on references I often make to sections 2 and 4. For now, I have proposed a new formal notation, with quite a few new contractual elements in the semantic formalism, as well as new time operators and two new templates for reaching a canonical.

Regarding the software components, I have conducted important work on the Smart Contract Editor which has, for now, has a basic string parser and the graphics component connected. I am now working on a class structure for each contractual element (e.g. state, gate/transition, parties involved, time, date, events, actions, trans-assertions) combined with a good use of Java interfaces, abstract classes and inheritance so that proposals made in Section 5 are obvious in the program code.

On the following page, I am showing a print screen of the IDE (on the left) showing the classes that reflect the contractual elements, the contract file (in the centre) using a new trans-assertion notation (the final notation includes three times as many elements as shown in the example), and the program running (on the right side) showing the Petri net representation of the contract.


ToggleButton
TokenEditorPanel
LayoutForm
handlers
historyActions
ucl
constructor
controllers
Constructor
InputParser
models
InputLine
contract
enums
StateType
interfaces
GetCalendar
GraphicalRepresentation
models
DateOperators
AfterDate
BeforeDate
ScheduleDate
TimeOperators
RDuring
RThroughout
TimeOperator
ActionElement
Contract
ContractElement
DateElement
EventElement
GateElement
PartyElement
StateElement
STATE(S1, 'START', START)
STATE(S2, '2', INTERMEDIARY)
STATE(S3, 'Jones paid £500', INTERMEDIARY)
STATE(S4, 'DEFAULT: Jones did not pay £500', DEFAULT)
STATE(S5, 'Smith delivered washer', INTERMEDIARY);
STATE(S6, 'DEFAULT: Smith did not deliver washer', DEFAULT)
STATE(S7, 'END', END);
DATE(D1, "Payment due date", 03-MAY-1987)
DATE(D2, "Delivery due date", 15-MAY-1987)
FUNCTION(F1, "PAY", SMITH, £500)
FUNCTION(F2, "DELIVER", Jones, Washer)
GATE(G1, "True Gate", TRUE,,,) // The last
GATE(G2, "Pay Washer", TRUE, Jones, RB(D1), F1)
GATE(G3, "Pay Washer", FALSE, Jones, RB(D1), F1)
GATE(G4, "Deliver Washer", TRUE, Smith, RB(D2), F2)
GATE(G5, "Deliver Washer", FALSE, Smith, RB(D2), F2)
GATE(G6, "True Gate", TRUE,,,)
TA(S1, S2, G1)
TA(S2, S3, G2)
TA(S2, S4, G3)
TA(S3, S5, G4)
TA(S3, S6, G5)
TA(S5, S7, G6)
Analysis Module Manager
Available Modules
GSPN Analysis
State space exploration
Find IModule
Petri Net 0
S1:"START"
G1:"True Gate"
S2:"2"
G2:"Pay Washer":Jones:PAY(SMITH,£500):RB(03-MAY-1987)
G3:"Pay Washer":!:Jones:PAY(SMITH,£500):RB(03-MAY-1987)
S3:"Jones paid £500"
S4:"DEFAULT: Jones did not pay £500"
G4:"Deliver Washer":Smith:DELIVER(Jones,Washer):RB(15-MAY-1987)
G5:"Deliver Washer":!:Smith:DELIVER(Jones,Washer):RB(15-MAY-1987)
S5:"Smith delivered washer"
S6:"DEFAULT: Smith did not deliver washer"
G6:"True Gate"
S7:"END"

Remaining Work

The two main objectives I need to accomplish is finalising the software (which should be done and tested by the end of next week) and finalise the 5th section by the end of February.

The only thing left after that regarding my project report is writing the Testing and Validation of the proposed model by the final deadline which will still involve some serious work.

I also have to reach a decision on how to proceed with my second software tool. I have not yet decided how the tool will work or how I will implement it and I am considering whether I should build it into the first tool. This would mean that I would be able to use the existing class structure and the existing parser which would save precious time for this project, without renouncing any of the functionality.

Bibliography

[1] A. R. Bart Cant, Amol Khadikar and J. B. Bronebakk, "Smart contracts in financial services: Getting from hype to reality," tech. rep., Capgemini Consulting, Aug. 2017.

[2] J. Stark, "Making Sense of Blockchain Smart Contracts," June 2016. https://www.coindesk.com/making-sense-smart-contracts/.

[3] C. D. Clack, "Smart contract templates: legal semantics and code validation," *Journal of Digital Banking*, vol. 2,4, pp. 1–15, 2018.

[4] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *Barclays Bank PLC 2016-2017*, Mar. 2017.

[5] R. M. Lee, "A logic model for electronic contracting," *Decision support systems*, vol. 4, pp. 27–44, Mar. 1988.

[6] R. M. Lee, "International contracting - a formal language approach," in *System Sciences, 1988. Vol. IV. Applications Track., Proceedings of the Twenty-First Annual Hawaii International Conference on*, vol. 4, pp. 69–78, IEEE, Jan. 1988.

[7] R. M. Lee, "Towards open electronic contracting," *Electronic Markets*, vol. 8, no. 3, pp. 3–8, 1998.

[8] R. M. Lee and V. H. Nguyen, "Formal aspects of denontic process modeling," *Decision support systems*, Feb. 2012.

[9] N. Szabo, "Smart contracts: Building blocks for digital markets," *Extropy*, vol. 16, 1996.

[10] Wikipedia, "Smart contract," Nov. 2017. https://en.wikipedia.org/wiki/Smart_contract.

[11] C. Clack, "Smart contract templates: The semantics of smart legal agreements," Nov. 2017.

[12] T. Hvitved, *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, The Faculty of Science, University of Copenhagen, 2011.

[13] G. H. Von Wright, "An essay in deontic logic and the general theory of action," *Acta philosophica Fennica*, vol. 21, 1968.

[14] L. Giordano, A. Martelli, and D. T. Dupré, "Temporal deontic action logic for the verification of compliance to norms in asp," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, pp. 53–62, ACM, 2013.

[15] A. Goodchild, C. Herring, and Z. Milosevic, "Business contracts for b2b.," *ISDO*, vol. 30, 2000.

[16] A. Boulmakoul and M. Sallé, "Integrated contract management," in *Proceedings of the 9th Workshop of the HP OpenView University Association*, 2002.

[17] S. P. Jones and J.-M. Eber, "How to write a financial contract," *Citeseer*, 2003.

[18] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne, "Run-time monitoring and enforcement of electronic contracts," *Electronic Commerce Research and Applications*, vol. 3, no. 2, pp. 108–125, 2004.

[19] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal, "A unified behavioural model and a contract language for extended enterprise," *Data & Knowledge Engineering*, vol. 51, no. 1, pp. 5–29, 2004.

[20] Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni, "On design and implementation of a contract monitoring facility," in *Electronic Contracting, 2004. Proceedings. First IEEE International Workshop on*, pp. 62–70, IEEE, 2004.

[21] O. Marjanovic and Z. Milosevic, "Towards formal modeling of e-contracts," in *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pp. 59–68, IEEE, 2001.

[22] F. Henglein, C. O. E. Stefansen, J. G. Simonsen, J. Andersen, and E. Elsborg, "Compositional specification of commercial contracts (technical. report)," Tech. Rep. 6, Department of Computer Science, University of Copenhagen (DIKU)Copenhagen Denmark and Institute of Theoretical Computer Science, IT University of Copenhagen (ITU)Copenhagen S, Denmark, 2004.

[23] C. Prisacariu and G. Schneider, "A formal language for electronic contracts," in *FMOODS*, vol. 7, pp. 174–189, Springer, 2007.

[24] C. Prisacariu and G. Schneider, "A dynamic deontic logic for complex contracts," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 4, pp. 458–490, 2012.

[25] Wikipedia, "Petri net," Nov 2017. https://en.wikipedia.org/wiki/Petri_net.

[26] G. v. Wright, "And next," *Acta Philosophica Fennica*, vol. 18, pp. 293–304, 1965.

[27] H. J. Pithadia, "Capturing Language Semantics of Smart Contracts," Master's thesis, Departament of Computer Science, UCL, Sep 2016.

[28] N. Rescher and A. Urquhart, *Temporal logic*, vol. 3. Springer Science & Business Media, 2012.

[29] UK Payments Administration Limited, "UK Faster Payments Service (FPS)." http://www.fasterpayments.org.uk/.

[30] A. R. Anderson, "The formal analysis of normative systems," tech. rep., Yale Univ New Haven CT Interaction Lab, 1956.

[31] Citizens Advice, "If your employer says you can't work for a competitor." https://www.citizensadvice.org.uk/work/leaving-a-job/resigning/if-your-employer-says-you-cant-work-for-a-competitor/.

[32] Jobsite Worklife, "What do restrictive covenants mean in employment?," May 2017. https://www.jobsite.co.uk/worklife/beware-restrictive-covenants-contract-employment-restrict-options-move-jobs-10827/.

[33] M. Mansour, M. A. Wahab, and W. M. Soliman, "Petri nets for fault diagnosis of large power generation station," *Ain Shams Engineering Journal, Elsevier*, vol. 4, no. 4, pp. 831–842, 2013.

[34] Admin's Choice Magazine, "Crontab quick reference." www.adminschoice.com/crontab-quick-reference.

[35] WDT.io Inc., "Crontab." https://crontab.guru/.

[36] United Nations - Mechanism for International Criminal Tribunals, "Continuous obligation - ICTR/ICTY/MICT Case Law Database (SAINOVIC et al. (IT-05-87-A))."

[37] Law Insider, "Continuous obligation clause uses in agreement," 2018. https://www.lawinsider.com/usage/services-other-obligations/continuous-obligation-clause-uses-in-agreement.

[38] CTAN Atom, "Backnaur typeset - Backus Naur Form definitions." https://ctan.org/pkg/backnaur.

[39] Law Depot UK, "Loan agreement template - loan agreement form," 2018. https://www.lawdepot.co.uk/contracts/loan-agreement/.

[40] S. Tattersall, "Platform Independent Petri Net Editor version 5 (PIPE 5)," 2018. http://sarahtattersall.github.io/PIPE/.

[41] N. J. Dingle, W. J. Knottenbelt, and T. Suto, "PIPE2: A tool for the performance evaluation of generalised stochastic Petri Nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 34–39, 2009.

[42] P. Bonet, C. M. Lladó, R. Puijaner, and W. J. Knottenbelt, "PIPE v2. 5: A Petri net tool for performance modelling," in *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, 2007.

[43] B. Kiers, "Regular expression to select all whitespace that isn't in quotes." https://stackoverflow.com/questions/9577930/regular-expression-to-select-all-whitespace-that-isnt-in-quotes.

[44] S. Ostermiller, "Finding comments in source code using regular expressions," Mar 2015. https://blog.ostermiller.org/find-comment.

[45] B. Kiers, "Regular expression for splitting a comma-separated string but ignoring commas in quotes." https://stackoverflow.com/questions/1757065/java-splitting-a-comma-separated-string-but-ignoring-commas-in-quotes.

[46] G. H. Von Wright, "A correction to a new system of deontic logic," *Danish Yearbook of Philosophy*, vol. 2, pp. 103–107, 1965.

[47] F. Dignum and R. Kuiper, "Specifying deadlines with continuous time using deontic and temporal logic," *International Journal of Electronic Commerce*, vol. 3, no. 2, pp. 67–85, 1998.

[48] F. Dignum and R. Kuiper, "Combining dynamic deontic logic and temporal logic for the specification of deadlines," in *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on*, vol. 5, pp. 336–346, IEEE, 1997.

[49] J. Broersen, F. Dignum, V. Dignum, and J.-J. C. Meyer, "Designing a deontic logic of deadlines," in *International Workshop on Deontic Logic in Computer Science*, pp. 43–56, Springer, 2004.

[50] F. Dignum, H. Weigand, and E. Verharen, "Meeting the deadline: on the formal specification of temporal deontic constraints," in *International symposium on methodologies for intelligent systems*, pp. 243–252, Springer, 1996.

[51] J. Brunel, J.-P. Bodeveix, and M. Filali, "A state/event temporal deontic logic," in *International Workshop on Deontic Logic and Artificial Normative Systems*, pp. 85–100, Springer, 2006.

[52] G. Governatori, J. Hulstijn, R. Riveret, and A. Rotolo, "Characterising deadlines in temporal modal defeasible logic," in *Australasian Joint Conference on Artificial Intelligence*, pp. 486–496, Springer, 2007.