Conflict Analysis of Deontic Contracts

Stephen Fenech
Supervisors: Dr Gordon Pace & Dr Gerardo Schneider

Department of Computer Science and Artificial Intelligence
University of Malta

September 2008

Submitted in partial fulfilment of the requirements
for the degree of M.Sc. (Computer Science)

Abstract

Industry is currently pushing towards Service-Oriented Architectures, where code execution is not limited to the organisational borders but may be extended beyond. Typically, these sources are not accessible making it impossible to fully trust their execution. Contracts, expressing obligations, permissions and prohibitions of the different actors, can be used to protect the interests of the organisations engaged in such service exchange. Composition of different services with different contracts, and the combination of service contracts with local contracts can give rise to unexpected conflicts, exposing the need for automatic techniques for contract analysis. In this work we start by applying the contract language $\mathcal{CL}$ to the CoCoME case study in order to compare with other methods of specification. $\mathcal{CL}$ takes a deontic approach to specifying contracts formally. We then present a trace semantics of $\mathcal{CL}$ suitable for conflict analysis, and a decision procedure for detecting conflicts (together with its proof of soundness, completeness and termination). We also discuss its implementation and look into other applications of the contract analysis approach we present. These techniques are applied to a small case study of an airline check-in desk.

Dedicated to the memory of Carmel Fenech

Acknowledgments

There are a number of people who I wish to thank. First of all there are my two supervisors, Gordon Pace and Gerardo Schneider, both of which had a lot of patient with me and have been there all throughout my masters. I would like to thank them for guiding me when I felt stuck. I would like to thank Gordon for arranging opportunities to go working with research groups abroad; it was very fruitful for my masters. I would like to thank Gerardo for making sure everything goes smoothly when I was in Oslo and also arranging a stay in Aalborg.

I would like to thank Joseph C. Okika for coming to pick me up from the Aalborg airport when I arrived for my stay at the Aalborg University. He was very helpful, not only with the technical side of the masters but also all throughout my stay in Aalborg. I would like to also take this opportunity to thank Anders P. Ravn with whom I had the pleasure to work with in Aalborg and whose insights where very valuable in this initial period of my work.

From the University of Oslo I would especially like to also thank Olaf Owe who was very supportive all throughout my stay. I would also like to thank the other members of the PMA group with whom I spent my working hours during my stay in Oslo, especially Peter Ölveczky, Martin Steffen, Marcel Kyas, Joakim Bjørk, Arild Torjusen, Cristian Prisacariu and Daniela Lepri.

I would also like to thank my family for being there all through this year, especially my mother who proof read my draft even though most of it she found not interesting; even though she would not admit it openly. Finally I would like to thank Michéle Lange, for being supportive all through this year and also was part of the reason I chose to pursue this masters.

Contents

Preface

In this preface I would like to explain my personal motivation of why I wanted to continue working in academic research and why I took this particular subject. I thought that it would be fit to at least have a single part of a scientific document which takes a lighter, non-scientific style of writing. This way I may give my reasons for taking this masters. For those who are only interested in the scientific work, I would urge them to skip directly to the introduction since here I shall narrate what led me to work on this masters.

I had just finished my bachelor final year project where I created a framework where one could model check concurrent algorithms written in assembly. My approach was to create a virtual machine written in SMV on which one could execute assembly algorithms. SMV is a language used to model systems in order to model check. Thus this virtual machine was a model of how a processor would behave. Given the assembly algorithm my tool would generate an SMV model which will execute the code and then it is up to the user to decide what properties one would like to verify. The user had a number of facilities to change the behaviour and architecture of the system. For example one could decide if a multi-threaded environment rather than actually a multi-processor environment is required. The behaviour of the assembly code could be changed and even create custom assembly instructions. I also investigated a number of other techniques, particularly using induction in order to verify unbounded algorithms.

After this year working with model checking, I found this technique very intriguing. Even though there are a number of limitations on the size of the problems one could model check, I realised that it is a very viable technique which when used appropriately can help find problems in the system from very early on. It will also help in getting an overall picture, even by simply deciding what properties one would like to verify. Unfortunately, by the end of the project, I was feeling as though even though I did a great deal of good work, it still does not apply to the mainstream industry. There are not many industries who write only in assembly and most would rather program using C or C++ and only use assembly for certain specific parts. Thus my framework, even though useful, does not apply directly.

The industry today is pushing towards the service oriented architecture.

Large companies and organisations are taking part in complex communication with external services thus executing code which is external to the organisation. I have spent the past two years working on an open source enterprise service bus so I have quite some background knowledge on how the industry employs the service oriented architecture. So a question that started to dwell in my head was how could we make use of model checking or some formal tools in this scenario.

An enterprise service bus can be seen as a message hub or router that connects diverse systems in the enterprise or even between enterprises. The service bus will be able to transform messages passing through it in order to be able to branch between different systems that might be using different communication protocols or making use of different message structures. One may notice that the service bus will be spanning over most of the enterprise and it has quite a good view of the overall enterprise. It will not have the detailed information of the particular systems since it may not have direct access to the different parts. However, by observing the messages that are passing through it can get an idea of the overall behaviour of the system.

So the message service bus would be an effective place to observe the entire system. One could create a module inside the enterprise service bus in order to monitor the behaviour, or maybe ensure that the enterprise service bus behaves in a certain desired way. As things turned out, I obtained my answer to the question while talking to my supervisor.

When I went to talk to my supervisor about what master ideas he had, he started to talk about electronic contracts and the work he had been doing with a number of colleagues abroad. So if there is a service agreement between two or more companies that are communicating through the enterprise service bus, it would be an ideal place from where to monitor the agreement. This would entail the generation of the monitor from the contract. Furthermore, it would be a good idea if we could verify that the contract has a number of properties that could be verified and maybe even model check the contract.

So this was what I had in mind before starting to work on my masters. Throughout the year I had a number of problems and hassles, from the social hassles of setting up my overseas stays to more technical hurdles like the changing of the $\mathcal{CL}$ syntax. However, I believe that overall I achieved the goals that I started with.

Chapter 1

Introduction

Contracts are a means to protect the signatories in order to ensure that the needs of all the participants are respected. Such agreements are becoming a more integral part of the daily life of the computer science community. For example, in many software development life cycles, documents produced during the requirements phase are regarded to as contracts. These documents are signed by the clients and developers marking an agreement on what the final product should be and what penalties should apply if the contract is broken. Many companies are also outsourcing a lot of development and due to the push towards the Service Oriented Architecture (SOA) it is more frequent that we make use of code that is located outside of our organisation.

This adds a number of risks since now we are dependent on code that we have not developed or even code that we have no access to. In SOA, the services need not be found in our organisation, the code is executing remotely and we have no access to the internals of the service and thus one would like to protect oneself. Thus some form of contract is agreed upon to protect the interest of all parties. However, the interpretation of contracts are, like any other information represented in natural language, subjective and ambiguous thus possibly leading to disagreements.

If we formally describe contracts we avoid these problems since there is only one possible meaning to the contract. Furthermore, having a formal description enables a number of automatic possibilities since the contract is now machine readable and can be reasoned upon. With this formal contract we can possibly automatically analyse and monitor the contract, model check the system or even have automatic negotiation between services.

The contract language $\mathcal{CL}$, is a language designed to describe contracts where, even though it is not a logic in itself, its semantics are formally defined using an extention of the μ-calculus. This way we can logically reason and analyse the contracts. $\mathcal{CL}$ incorporates features from deontic logic, temporal logic and dynamic logic. However, having a formal representation is not enough. It would be desirable to not only have a sound theoretic background but also numerous tools for automatic analysis. This adds value to the usage of the language.

The aim of the masters is to investigatge how one can automatically analyse contracts, focusing mostly on automatic conflict analysis. It is important to note that often, a service is composed of a number of other services, each with its own contract. Thus when combining all the contracts together we need to ensure that they do not conflict with each other. Furthermore, if we look at a client and service provider scenario, they will have a service agreement dictating their interaction but they each may also have contracts dictating their internal behaviour. In this situation we need to ensure that the contracts are not conflicting. Automating this process will not only speed the negotiation process but also ensures that there are no mistakes thus having both parties protected.

Our approach will make use of techniques currently being applied in temporal logic model checking in order to unwind temporal and dynamic operators to represent what the contract is enforcing at any particular time. This results in an automaton that accepts only traces that do not violate the contract and during construction we shall also be able to find any conflicts present.

1.1 Motivation for Automatic Analysis of Contracts

What benefits would we obtain from automatic analysis of contracts? The answers to this question are the motivating points that drive this work. The first task in order to be able to analyse contracts automatically is to have a formal representation of a contract. There has been quite some research in this regard and I shall continue building on it. There has also been a great deal of work attempting to automate parts of the legal system in order to aid lawyers with their work. There has also been work in order to try and mimic the way lawyers and judges reason about the law. This drive to help lawyers has been present in the computer science community for a couple of years already but it was always a challenge due to the logical tools which were present.

Let us consider that a group of lawyers are drafting a contract. It would be desirable if after the draft is written, they can simply push a button and this will verify that the contract has a number of properties. It would be extremely useful to automatically ensure that the contract is conflict free, ensuring that the signatories will never end up in a position that the contract itself will not be satisfiable. It would also be desirable if we could ensure that the contract has certain properties that will ensure that there are no loopholes. Another attractive feature would be to help the lawyers clean up the contract by removing superfluous or overlapping clauses that might be confusing to the reader.

By automating this analysis it will not only speed up the writing of the contract, but it will also verify formally that the contract has the desired properties. This automation of general contracts is quite a big task. In this masters we shall focus only on a specific type of contracts, that is, contracts concerning software services. In this way, as we shall see later, we are able to focus on a usable and practical solution rather than get lost in deep philosophical issues of how to interpret contracts.

1.1.1 SOA

Today the industry is pushing towards the service oriented architecture (SOA). Service oriented architecture encourages the use of quite a number of best practices like code de-coupling and reuse. It also enables industries to be more flexible and also makes it easier to scale up. However, since the services may be found anywhere, not necessarily inside the same organisation, as we have seen in the introduction, many companies turn towards Service Level Agreements (SLAs) in order to protect the interests of the organisation.

How would automation of contract analysis help in this service oriented architecture? As described above, it would be very useful to be able to ensure that there are no conflicts in the contract and also other properties, which for example ensure that there are no loopholes. Also, if we focus on such service level agreements, it should be easier to reach our goals and then slowly extend these techniques to other areas.

Since SLAs usually describe the behaviour of computer systems, we could do even more with automation. It would be desirable as well to obtain a monitor automatically from the contract which will make sure that the parties involved adhere to the contract. If any violation occurs, it is observed by the monitor and it will either automatically take reparative actions or inform the parties to take legal action for example enforce payment of fines.

Contract automation fits very well in the SOA, especially when it comes to automatically generate monitors from the SLAs. There are, however, other areas where contract automation fits.

1.1.2 Agents

A software agent can be defined as any piece of software that acts on behalf of the user. There is a great deal of work in order to have automatic, intelligent agents that interact with each other and constantly looking for other agents that provide better services or offer better prices.

Such agents will also have some form of agreement made with each other and it would be desirable if it would be in some form of a contract in order to protect the interests of the users they are acting on behalf of. This would require some form of negotiation between the agents and this could only be done if we have a formal representation of these contracts and a way to analyse the contract drafts in order to be able to gauge if the agreement is beneficial or not.

This is a very interesting area since it requires some form of artificial intelligence and it may also be seen as a form of game where each agent will try to negotiate for the best deal possible.

1.2 Contribution of the Work

In this masters we mostly aim at developing a theory on how we can check that contracts do not have conflicts. In order to do this we needed to isolate to a certain extent the deontic notions of obligation, prohibition and permission from the temporal aspect of the $\mathcal{CL}$ formula, which is written using dynamic logic notation. In order to do this we opted for an automata theoretic approach. Once we construct the automata we could do further types of analysis and not just conflict analysis.

Another result of this work is the application of $\mathcal{CL}$ in a number of case studies. The CoCoME case study is a project aimed at comparing different specifications. Thus, we use this case study in order to focus on the specification of systems using $\mathcal{CL}$. We also tackle another case study. This shows how the techniques developed in this work could be applied in practice. Furthermore, due to the practical exposure of $\mathcal{CL}$ while working with these case studies, we found a number of points that could be improved in $\mathcal{CL}$ and hence contributing to the continual development of $\mathcal{CL}$.

From the theory we also developed a tool, which implements these analysis. Although this was not the main focus of the masters, by providing a tool and not just theory, we show the viability of using $\mathcal{CL}$ for contracts. Also, by automating analysis, we are adding value to users who use $\mathcal{CL}$.

1.3 Structure

The structure of this thesis is as follows. We shall start in Chapter 2 by describing a number of logics from which $\mathcal{CL}$is inspired. We start with a brief introduction of classical modal logic (Section 2.1) and then some more specific instances of modal logic, mainly dynamic (Section 2.2) and temporal (Section 2.3) logic since these apply more to the area of computer science. This is then followed by a brief overview of Model Checking techniques from where our solutions are inpired. This leads us to Chapter 3 where we discuss Deontic Logics and $\mathcal{CL}$. At the end of this chapter we also discuss some issues which we believe $\mathcal{CL}$ has and possible solutions. We also discuss a number of improvements we would like to eventually add to $\mathcal{CL}$.

After this introduction to $\mathcal{CL}$ we go through the CoCoME case study in Chapter 4. We show how one could specify the case study using $\mathcal{CL}$ and we compare it to how this could have been done using the temporal logics CTL and LTL.

The main contribution can be found in Chapter 5 where we show what changes need to be done to the semantics in order to be able to analyse for contradictions followed by the algorithm for finding conflicts and their proofs. Once we have the main algorithm in place, we could do a number of other types of analysis and these are tackled in the following chapter, Chapter 6.

In Chapter 7, we then look at how $\mathcal{CL}$ could be encoded into other logics thus comparing $\mathcal{CL}$ with a number of different logics. We also show how to use LTL model checking in order to perform certain analysis that could be done using our algorithm as well, mainly satisfiability checking and reachability analysis. These however, unlike our algorithm, are not fully automated.

Finally, in Chapter 8 we take a look at our implementation of the algorithm followed by a case study. In this case study we show how one would go about creating a contract when making use of our tool and techniques. This is then followed by the Conclusion.

Chapter 2

Logics

In philosophy and computer science, the word “logic” is associated to techniques that enable us to reason about truths and usually we associate it with propositional logic, also described as Boolean logic. Boolean logic can be seen as the most basic logic where we have a set of atomic propositions that can take two values, either true or false and have a number of Boolean operators. This logic is useful in order to describe certain state of affairs and reason about these universal truths, however it has limitations. For example, describing the notion that all men are mortal would require that we have a proposition for every possible man. This is clearly not elegant and in cases where the universe of discourse is infinite or unbounded, it would be impossible. This need leads to the introduction of two new operators resulting in the creation of predicate logic.

The two new operators of predicate logic are the existential and universal quantifiers. The existential quantifier describes the situation where there exists some subject that has certain properties whereas the universal quantifier describes the situation where all the subjects have these properties.

As Socrates said in the dialogue “Republic”, “Necessity is the mother of invention” [1] and thus, depending on what is desired from the logic and on what one desires to reason about, different systems have been devised. In this chapter we shall describe a number of these logics that have been developed in order to allow the reasoning and representation of certain properties. We first take a look at Modal logic, which describes different *modes* of truth and we are able to reason about possibility, necessity, obligation, belief or perception among others. This is then followed by Dynamic and Temporal logics, which are also Modal logics but they were aimed at verification of computer programs. We describe these logics since $\mathcal{CL}$ has adopted a number of features from these logics and thus it is useful to see what $\mathcal{CL}$ offers

[1] In the “Republic”, written by Plato, Socrates and Adeimantus where discussing the origin of the state and Socrates said “the true creator is necessity, who is the mother of our invention.”

and what problems it has solved when compared with its ancestors.

2.1 Modal

The term modal logics describes a large set of logics and the literature about modal logic agrees that there is not just *one* definition of what a modal logic is. This can be seen by the definition given by Blackburn et al in their book Modal Logic [49]. They make use of three definitions, what they called Slogans rather than a definition in order to define what modal logic is. In the “Handbook of Modal Logic” [6] two different and seemingly opposing views of what modal logic is are given in order to emphasise this fact.

I would describe Modal logic as any logic that tries to deal with what are known as modalities. A modal is described as qualification of truth of a judgement [23]. Usually it is easier to explain this with an example of what such modes could be. Classically, modal logic had two main modalities, mainly “necessity” and “possibility” and these modes qualify some truth. Some examples of these modal prepositions would be “It is possible that it will rain tomorrow” and “It is necessary that it is raining now or it is not raining now”. These modes when looked at from a linguistic perspective are usually characterised as adverbials [23] and some examples are: *necessarily*, *possibly*, *known* by me to be, *believed* by you to be, *permitted* to be, *eventually* will be, *obliged* to be, *forbidden* to be. This gives room for many different modal logics that define these different modalities and allows us to reason about these different modes of truth. Apart from classical modal logic, which makes use of the possible and necessary modalities, we also have others. Cases in point are temporal logic, which makes use of tense modalities, deontic logic, which makes use of the normative modalities (obligation, permission and prohibition) and epistemic or doxastic modalities, which are aimed at reasoning about knowledge and belief.

Philosophers have developed and discussed these different modes of truth for many centuries [20]. The modalities of necessity and possibility have been discussed by philosophers from the time of the Greek and these two modalities are sometimes referred to as the “alethic” modalities, which comes from the Greek word for “truth”. Aristotle noticed that necessity implies possibility but the converse does not hold and wrote about this in De Interpretatione [73]. He also noticed that these two notions are interdefinable, where the possibility of p is not the necessity to not do p. From Aristotle’s time the discussion about modalities has had many contributions. The Megariens, the Stoics, Ockham and Pseudo-Scotus are but a few from a long list.

This discussion has found itself in a dead end after the work by the Scholastics. It was Leibniz’s suggestion that there are other possible worlds besides the actual world, which gave light to how this logic can keep developing. It was Kripke who formalised the notion of possible worlds and it is one of the most easy to understand views of modal logic giving an intuitive description of modalities. It was however C. I. Lewis, years before Kripke, who first studied modal logics in a formal manner [6] in 1918. He introduced the first modal operators and attempted to solve paradoxes of material implication and to obtain logics of necessity and possibility.

In contemporary modal logic many of the contributions shifted from philosophers to computer scientists since a sound theoretic formalism of these modalities will enable machines to reason about these modalities. Even though this area has its roots in the classical period it is still relatively fresh and there is a great deal of work being done on different aspects of modal logics.

2.1.1 General Interpretation of Modal Logics

A way to look at modal logics is that there exists a number of different worlds and the modal operators enable us to access one world from another. If we compare this to Boolean logic, a truth table is a listing of all possible worlds and each row of a truth table is one such possible world. This representation for modal logics was introduced by Kripke in the 1960's and also independently by Hintikka.

Kripke makes use of a model in order to define these worlds, where a model is the tuple $\langle \mathcal{G}, \mathcal{R}, \models \rangle$ where $\mathcal{G}$ is the set of worlds and $\mathcal{R}$ is the relation $\mathcal{G} \times \mathcal{G}$ where if P and Q are two worlds and $(P, Q) \in \mathcal{R}$ then the world Q is accessible from P. $\models$ is the relation between the possible worlds and the set of atomic propositions; thus, if $(Q, p) \in \models$ then p is true in world Q. We can represent the relations $\mathcal{R}$ and $\models$ as $P \ \mathcal{R} \ Q$ and $Q \models p$ respectively. In literature, one refers to the set of worlds and their accessibility relation as a frame; thus, a frame is the tuple $\langle \mathcal{G}, \mathcal{R} \rangle$.

If we have the classical modalities of possibility and necessity with their classical representation, $\Diamond$ and $\Box$ respectively, we can define truth in a model as follows:

Given a model $\langle \mathcal{G}, \mathcal{R}, \models \rangle$, the relation $\models$ can be extended to arbitrary formulas as follows. For each $P \in \mathcal{G}$

1. $P \models \neg p$ iff $P \not\models p$
2. $P \models (p \wedge q)$ iff $P \models p$ and $P \models q$
3. $P \models (p \vee q)$ iff $P \models p$ or $P \models q$
4. $P \models \Box q$ iff for every world Q such that $P \mathcal{R} Q$ then $Q \models p$
5. $P \models \Diamond q$ iff there exists a world Q such that $P \mathcal{R} Q$ and $Q \models p$

One should note that we do not need to define both $\wedge$ and $\vee$ since they are interdefinable using the negation. $\Diamond$ and $\Box$ are also interdefinable using negation.

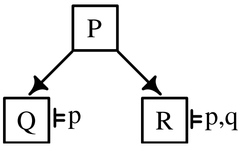


Figure 2.1: An example of a model

Figure 2.1 shows one such model. In this model, since $Q \vDash p$ and $R \vDash q$, then $P \vDash \Box p \vee q$. Furthermore, since $R \vdash p$ then $P \vDash \Box p$. However, $P \nvDash \Box q$ since $Q \nvDash q$ but $P \vDash \Diamond q$ since there exists an accessible world where q is true.

After this brief introduction to modal logics and its classical representation we are going to take a look at two modal logics, Dynamic Logic and Temporal Logic that have been developed with program verification in mind.

2.2 Dynamic

Dynamic logic can be described as a modal logic with the emphasis on actions. It has the classical modal operators $\Box$ and $\Diamond$ but also adds the operators $[]$ and $\langle\rangle$. If a and p are atomic propositions, $[a]p$ would mean that after a is performed p must follow. On the other hand the $\langle a\rangle p$ means that it is possible to perform a and if a is performed then p has to be performed.

Actions found between the square and angle brackets might be composed of a number of atomic actions combined using operators. Such actions are called compound actions. Usually the following operators are used, where $+$ stands for choice, ; or $\cdot$ stands for sequence and * is an iteration (Kleene star). Some other operators that are also used in certain extensions are the test operator ? and concurrency operator &.

This type of modal logic was aimed at verification of programs and was developed in 1974 by Pratt [52].

2.3 Temporal

Temporal Logic, similar to Dynamic Logic, was aimed at program verification [51] but concentrates, as the title suggests, on time. Temporal logic also falls under the umbrella of modal logic where temporal operators or modalities are added in order to reason about how truth values change over time [20]. Temporal logic adds temporal operators to the classical Boolean operators where **X** means in the next instance, **G** means always or globally, **F** means finally or eventually and **U** stands for until.

This type of logic has become very popular since in the 80s an algorithm for verifying temporal logic properties of finite-programs was discovered [57, 13, 37]. This led to a method called "model checking" where the process of verifying these properties on a model was automated. It was called model checking since the program was modelled as a Kripke structure on which the property was verified. We shall go into more detail about model checking in Section 2.6.

Temporal logics can be split into two types, either Linear or Branching. Lamport in [35] describes these two types of temporal logics as being two interpretations to the two different ways of looking at time. One can look at time in a linear fashion, having a string of actions, that is, one event happens after another. The other way of looking at time is as branching, where at each instance there are a number of possibilities that may be done.

There are many papers that discuss the differences between these two types of temporal logics and which of the two logics are best suited for which situation. In [35] linear temporal logic was stated to be better for reasoning about concurrent programs whilst branching for nondeterministic programs. There are also many discussions that compare these two types of logics like [21, 38, 33].

2.3.1 CTL*, CTL and LTL

CTL, LTL and CTL* [19] are all examples of temporal logic. We are going to first take a look at CTL* whose formulas are made up of two parts, a path quantifier and temporal operators. There are two quantifiers available for this type of logic, **A** which means for every path and **E** which means for at least one path. These two operators are similar to the $\forall$ and $\exists$ operators in predicate calculus and thus **A** means $\forall$ reachable paths whilst **E** means $\exists$ a path.

The second part of a CTL* formula is made up of temporal operators. These are

- **X** : meaning that in the next step this property holds
- **F** : meaning that eventually (in the future) this property will hold
- **G** : meaning that this property will always (globally) hold
- **U** : this is a binary operator and means that until the second property holds, the first property will always be true

These operators can be combined together using Boolean operators to build complex temporal formulas for describing desired situations. For example, if we want to say that it will always be the case that in the future we

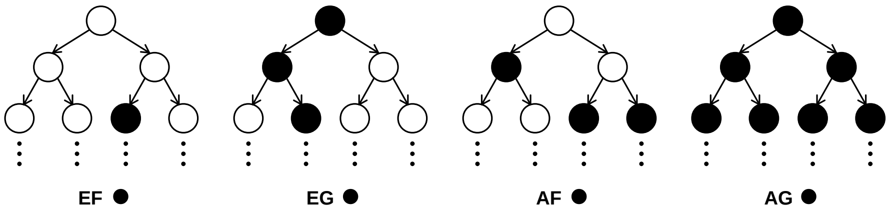


Figure 2.2: State representation of the Basic CTL operators

reach a state where property a is true and property b is false, the equivalent temporal formula would be $\mathbf{AF}(a \wedge \neg b)$. Another example would be if we want to describe that there exists a path that will repeatedly lead to a being true. We would formalise this as $\mathbf{EGF}(a)$.

CTL and LTL are restrictive sublogics of CTL*. CTL* is more powerful than both CTL and LTL. CTL and LTL do intersect each other; thus, being able to represent some identical concepts using all three languages. However CTL can express certain behaviours that LTL cannot and vice-versa. The main difference between CTL and LTL is that CTL is a branching-time logic and LTL is a linear-time logic.

In CTL the temporal operators are used on the possible paths from a given state. Thus every temporal operator must always be preceded by a path quantifier. This clearly makes it less expressive than CTL*. Thus in CTL we cannot define something like $\mathbf{A}(\mathbf{FG}\ \mathbf{p})$ meaning that at some point in the future, p is true and will remain true forever.

In LTL, the temporal operators apply to a single computation path. In other words all temporal formulas specified in LTL will have the **A** path quantifier followed by a path formula. This will result that formulas, like for example $\mathbf{AG}(\mathbf{EF}\ \mathbf{p})$ (meaning that it is always true that a path exists that will lead to p being true in the future), cannot be expressed. Even though all formulas in LTL make use of the **A** path quantifier we still may check that certain properties are true for at least a single path(i.e. **E** quantifier). One can negate the property and the Model Checker should return a trace of one such path, effectively verifying that one such path does exist.

2.4 μ-Calculus

The μ-calculus was developed by Kozen [31]. It makes use of the fixed-point operator making it very powerful. μ-Calculus may also be referred to as a temporal logic. In fact, LTL, CTL and CTL* may be encoded in the μ-calculus. However, μ-calculus is described as being very complex. It is very difficult to make sure that what is being expressed is actually what is desired. Furthermore most desired properties make use of two levels of alternation. So most of the power that μ-calculus has is not typically used.

Because of this, when it comes to verification, most of the time other logics like LTL and CTL are used. For example, the alternation-free μ-calculus model checker evaluator [40] forming part of the CADP toolset [2] allows an easy mechanism of making use of macros in order to encode, for example CTL, so as to be used as the top level language rather than write properties directly in μ-calculus.

2.5 Default Logics

Default logic [58] allows us to reason about incomplete data, something that classical logic is not capable of. As one might deduce from the title of this logic, we are capable of reasoning about incomplete data by making the use of defaults. This type of reasoning is commonly used, for example in the medical profession where instead of performing all possible tests in order to diagnose the patient, from an incomplete set of information the doctor will try to deduce what the patient has by filling up the rest of the information making use of defaults or as colloquially referred to, 'rules of thumb'. So with incomplete information the doctor may deduce a cause and thus, start with the treatment before the results from all the tests have arrived. One should note that as the results start to come in, these may affect the diagnosis and thus, as more facts are added to the picture the final diagnosis may change. This is also captured in Default logic and this is called nonmonotonic reasoning.

Consider we have a set of premises (truths) P and from this set P we may deduce α. If we are using monotonic reasoning, if $P \subseteq P'$ where P' is another set ot truths, then α must also be deducable from P'. If $P = \{a \rightarrow b, a\}$, we can deduce b and if $P' = \{c, a \rightarrow b, a\}$, since $P \subseteq P'$, then it must follow that b is also deducable from P'. This is not the case with nonmonotonic; thus, α is not necessarily deduced from P'. Going back to the diagnosis example, if from the current set of truths we deduce the patient's diagnosis, as we start getting more information about the case, the diagnosis may change.

Default logic was proposed by Raymond Reiter in [58]. This logic has been used very frequently in the area of artificial intelligence and many different extensions have been put forward. This logic has also been used very frequently in legal reasoning since similar to the medical example, law makes use of defaults when there is nothing that specifies the contrary. A good tutorial paper about the subject is [2] by Grigoris Antoniou. We shall now give a brief description of the semantics of default logic.

In default logic, a *default theory* T is a pair (W,D) where W is a set of predicate logic formulas, which are the facts (axioms) and D is a set of

[2] The CADP toolset and information about it and Evaluator model checker may be obtained from http://www.inrialpes.fr/vasy/cadp/

defaults. A default has the following structure:

$$\frac{\sigma : \varphi_1, \varphi_2, \ldots, \varphi_n}{\chi}$$

where σ is refered to as the 'prerequisite', φ the justifications and χ the conclusion. Using this syntax, in order to conclude that χ holds in the current default theory T, σ must be true in T (i.e. an element of W or deduced from W). Furthermore, the justifications should be consistent in T. By consistent we mean that we have no contradicting fact in T. If so, then we may conclude χ. One should note that if we add a new fact to W that contradicts a justification, then we cannot deduce χ anymore. One should also note that the order that we apply defaults to the theory does make a difference and thus certain theories have more than a single interpretation.

2.6 Model Checking

Model checking is the technique to automatically verify finite state systems [19]. It is a very attractive alternative to testing or simulation because the model checker will exhaustively search through all the possible execution paths verifying the desired properties unlike testing or simulation where not all execution paths are necessarily analysed. This type of formalisation is also attractive because once the model is built, everything is done automatically. Furthermore, if there is a situation where the actual specification does not hold, one will typically get a trace of how the system ended up in the bad state thus one is directed to the source of the problem. However model checking does have its draw backs. First there is the state explosion problem which we shall discuss in the following section. Secondly, model checking is restricted to a finite state system. However, this technique is still invaluable in many situations.

In order to model check a system, there are three steps one needs to follow

Modelling The building of the formal model that is accepted by the model checking tool.

Specification Stating properties desired. It is important that the specification is complete since the model checker will only verify properties specified; thus, if you do not have a complete specification there still might be problems in the algorithm.

Verification Automatic verification of the specification on the model.

From these three steps, the first two need to be done by the user whilst the verification step is done automatically.

The modelling usually takes the form of a Kripke structure, which can be seen as a special case of a Finite State Automation. The Kripke structure M is the four-tuple $M = \langle S, S_0, R, L \rangle$ where

- S is a finite set of states
- S$_0$ is the set of initial states ($S_0 \subseteq S$)
- R is the set of possible transitions, which by definition must be total ($R \subseteq S \times S$)
- L is a function that labels every set with the set of propositions true in that state.

The Kripke structure is used to represent all the states of a system and the transitions between these states. The labelling function will label each state with all the true propositions. In order to represent a system with four Boolean state variables the Kripke structure will be made up of 4×4 states. As the number of variables increase, so do the number of states in an exponential manner. This is known as the state explosion problem, due to which we cannot use Model Checking to directly prove a system with a large number of variables.

2.6.1 State Explosion Problem

As the number of state variables in a Kripke structure increases, the number of states increases exponentially, thus making the model impossible to check. A number of techniques have been employed in order to try and reduce as much as possible this state explosion in order to be able to check models with a larger number of states.

Symbolic Model Checking was the first attempt to reduce this problem, where the transition relations of the Kripke structure were represented using Ordered Binary-Decision diagrams [10]. This allows us to model check more than 10^{20} states as seen in the appropriately titled paper "Symbolic Model Checking: 10^{20} States and Beyond" [29].

This does not completely solve the problem since there still is a limit on the size of the model. Researchers turned to abstraction in order to be able to split the model into smaller chunks and thus model checking only parts of the system at a time, or a high level abstracted model with most of the inner state removed.

For a much more detailed description of how the State Explosion Problem has been tackled, please refer to the paper "Progress on the State Explosion Problem in Model Checking" [12].

2.7 Conclusion

In this chapter we have seen an overview of logics which have affected $\mathcal{CL}$ in some way or another. We first tackled a description of modal logics under which both $\mathcal{CL}$ and many other logics that are used in computer science reside. Two special modal logics that we mentioned are dynamic and temporal logics. Dynamic logic is a prominent part of $\mathcal{CL}$ as we shall see in the next chapter. Temporal logic formed part of the initial $\mathcal{CL}$ syntax; however, the authors then decided to make use soley of dynamic logic since temporal aspects may be expressed using dynamic logic. However, we still mentioned temporal logic since the techniques used for model checking of temporal formulas have inspired the conflict analysis approach of this work. We also discussed briefly μ-Calculus. It was used in order to specify the initial semantics of $\mathcal{CL}$ and default logic, which has some attractive features with respects to contracts. Finally we briefly described the model checking problem.

In the next chapter we shall tackle deontic logics and finally $\mathcal{CL}$. We shall have a brief introduction on deontic logics and the issues faced, mainly the paradoxes. Then we have an extensive description of $\mathcal{CL}$, seeing how the syntax and semantics changed and where $\mathcal{CL}$ currently stands.

Chapter 3

Deontic Logic

3.1 Deontic Logic

Deontic logic may be described as the logic of obligation, prohibition and permission; however, an even better description of deontic logic is the logic of ideal behaviour and actual behaviour [44], or equivalently to reason about normative and non-normative behaviour [45]. Another indication to what deontic logic means is by looking at the root of the word deontic, which comes from the greek word $\delta\epsilon o\upsilon\tau\omega\varsigma$ meaning “as it should be”.

As one might expect, Deontic logic has its foundation in philosophy of law and ethics. However, as happens with many subjects in philosophy, it has now also become a subject of interest for computer scientists, who added their own flavour to it moving away from the philosophical into the more formal paradigm. With the view of deontic logic as a method for reasoning about ideal and actual behaviour, we may use it to model and describe the ideal behaviour of the system and the exceptional behaviour that cannot be completly described using other classical logics.

Clasically, in deontic logic, one makes use of three deontic operators representing permission, obligation and prohibition; however, other formalisations are possible for concepts like duty, right, power and liability. Some examples of statements we could express using deontic logic would be “You have the permission and duty to vote unless you are incarcerated, in which case you are forbidden from voting” or “you are obliged to pay, otherwise we shall be obliged to fine you”.

We shall now go through a brief historical overview of deontic logic. The normative notions of deontic logic have been investigated since Aristotle’s time; however, we shall not go back so far. We shall take a look at how deontic logic developed from when a formal semantics or representation was given, or attempted to be given, to this deontic reasoning.

3.1.1 A Brief Historical Overview of Deontic Logic

Ernst Mally was the first person to try to formally reason about deontic logic in 1926 in his paper entitled ‘Elemente der Logik des Willens’. We use the word ‘try’ since Mally’s system was in fact a fragment of alethic propositional logic and thus not really a deontic logic after all.

Standard Deontic Logic or SDL was the first *real* attempt towards deontic reasoning. This was done by von Wright in [69] where he made use of a modal Kripke style description. This system is today called the ‘Old System’ (OS) since another system has been developed on this ‘Old System’.

SDL consists of the following axioms and rules

(K) $O(\varphi \to \psi) \to (O\varphi \to O\psi)$

(D) $\neg O\bot$

(N) $\frac{\varphi}{O\varphi}$

(P) $P\varphi \leftrightarrow \neg O\neg\varphi$

(F) $F\varphi \leftrightarrow O\neg\varphi$

(T) All (or enough) tautologies of propositional logic

(MP) $\frac{\varphi, \varphi\to\psi}{\psi}$

The semantics of the standard system were based on the semantics given to modal logics where we have the notion of possible worlds. The possible worlds act as the set S in a Kripke model. Each world has a set of primitive propositions which have been assigned a truth value using a valuation function π and there is a relation $R \subseteq S \times S$ which relates these worlds together.

Using this Kripke model $M = \langle S, \pi, R\rangle$ we may give the semantics to the deontic operators as we would do to modal operators where

$$(M,s) \models Op \text{ iff } \forall s' \text{ s.t. } (s,s') \in R \text{ then } (M,s') \models p$$
$$(M,s) \models Pp \text{ iff } \exists s' \text{ s.t. } (s,s') \in R \text{ and } (M,s') \models p$$
$$(M,s) \models Fp \text{ iff } \forall s' \text{ s.t. } (s,s') \in R \text{ then } (M,s') \not\models p$$

It is important to note that here we are treating O as the basic modal operator, where we define its semantics as for Op to be true in a world s then p must be true in all the worlds which are related (accessible) from s. From a modal logic perspective, O is similar to the necessity operator. The P and F operators may be defined using the O operator. For Pp to hold in a world s then there must exist an accessible world where p is true whereas for Fp to be true, p cannot hold in all accessible worlds.

With this view of SDL, we allow the obligation of a tautology, or what is called an 'Empty Normative System'. This was not really accepted by von Wright since he deemed it to be a paradox. We shall discuss this in Section 3.1.2 where we deal with paradoxes of deontic logic.

Allowing conditional obligations gave rise to many severe problems. One such example is Ross's paradox. In order to solve this problem von Wright in [70] introduced the 'New System'(NS) (and made some corrections in [71]). The OS syntax is augmented by $O(p/q)$ meaning that p is obliged under some condition q. Unfortunately, other undesirable theorems are introduced, in particular $O(p/q) \to \neg O(\neg p/r)$. This is described by von Wright himself as "manifestly absurd".

Many amendments to this system were attempted; however, some of these undesirable properties remained and some of the proposed solutions are quite complicated. Some people who attempted these amendments are von Wright himself, van Fraassen, van Eck, Hansson, al-Hibri, McCary and Soeteman.

The idea to introduce temporal notions to deontic logic was first proposed by van Eck [63] . Enriching deontic logic with this information about time gives us a more refined way of describing deontic formulas. This permitted them to remove of certain paradoxes like Chisholm's Paradox. Thomason [61] was another author who firmly believed that deontic logic needs this foundation in temporal logic.

In 1992, Fiadeiro and Maibaum [22] proposed to do this reduction from a deontic specification to a temporal one semantically. They interpreted the obligation to do something as a liveness property, so Op would be translated to Fp, in the future p will hold. They also made use of features from dynamic logic.

Meyer proposed another reduction into propositional dynamic logic using the violation atomic preposition V. This approach resulted in a shift of how deontic logic is viewed, where now, instead of taking the "ought to be" approach (in deontic literature called "Seinsollen") we take the "ought to do" approach (in deontic literature called "Seinsollen").

The three deontic operators are defined using dynamic logic as follows

$$\begin{aligned} F\alpha &\equiv [\alpha]V \\ P\alpha &\equiv \neg F\alpha(\langle\alpha\rangle\neg V) \\ O\alpha &\equiv F(\overline{\alpha})([\overline{\alpha}]V) \end{aligned}$$

Being forbidden to perform action α is reduced to the formula that states that if α is observed, this will lead to a state where V holds. The formula α is permitted is reduced to α is not forbidden, or that there exists some way to perform α without ending in a state where V is true. The definition of the obligation makes use of the compliment of action α, where now, if we observe the compliment of α we shall end up in a violating state. Even though the complement of an action might be intuitive, it poses a number of formal problems.

Furthermore, as we have seen in Section 2.2 actions might be compound making use of sequential composition, non deterministic passive or imposed choice and parallel composition.

The beauty of this approach is twofold. First of all, we get rid of many, if not all of the nasty paradoxes that have been a burden on traditional deontic logic. Some publications discussing this are Meyer [43], Anderson [1] and Castaneda [46]. Secondly, we can directly integrate deontic constraints with dynamic integrity constraints as done in [66] and [68].

Using the reduction of the deontic operators in conjuction with the compound action operators we have the following truths

$$\begin{array}{rcl}
F(\alpha;\beta) & \equiv & [\alpha]F(\beta) \\
O(\alpha;\beta) & \equiv & O\alpha \wedge [\alpha]O\beta \\
P(\alpha;\beta) & \equiv & \langle\alpha\rangle P\beta \\
F(\alpha+\beta) & \equiv & F\alpha \wedge F\beta \\
P(\alpha+\beta) & \equiv & P\alpha \vee P\beta \\
O(\alpha\&\beta) & \equiv & O\alpha \wedge O\beta \\
 & & \\
O\alpha \vee O\beta & \rightarrow & O(\alpha+\beta) \\
F\alpha \vee F\beta & \rightarrow & F(\alpha\&\beta) \\
P(\alpha\&\beta) & \rightarrow & P\alpha \wedge P\beta
\end{array}$$

As shown in [43] this approach resolves most paradoxes, including the classical Chisholm's paradox. Furthermore, certain paradoxes are not even expressible in the language any more. Unfortunately, this representation still allows Ross's paradox; however, some doentic logicians do not consider this a genuine paradox. Even though we solve most problems, we end up introducing new paradoxes. Consider the third equality, looking at the implication from left to right we may describe the following: If after shooting the president, it is permitted to remain silent, then it is also permitted to shoot the president and remain silent.

The current approach here is that the end justifies the means, that is, we only check if we have a violation of the contract at the end of a sequence ($\langle\alpha\rangle P\beta \equiv \langle\alpha\rangle(\langle\beta\rangle\neg V)$). This was observed by van der Meyden in [62] where he also gave a solution to this. We could also solve this by having a stronger permission operator $P*$ where $P*(\alpha;\beta) \equiv P*\alpha \wedge \langle\alpha\rangle P*\beta$ and $P*\alpha \equiv P\alpha$. So without sequences of actions, $P*$ behaves like P. It is clear that using this stronger version of P the president paradox does not hold. This $P*$

operator is however not strong enough either since we need to ensure that after the first action of the sequence we do not end up in a violating state and thus, we shall define $P**(\alpha;\beta) \equiv \langle\alpha\rangle(\neg V \wedge P**\beta)$. This version of the permission operator does not suffer these problems since we ensure that every step is permitted and that every step does not lead to a violating state.

Another method is the application of defeasible reasoning [1] methods to the notions of obligations. In defeasible reasoning we are faced with the problem of inferencing without having complete knowledge; thus, this generally results in a logically unsound reasoning mechanism where by adding new information might result in a contradiction. This is then followed by some mechanism to reinstate consistency.

This approach is quite different from what has already been done. For example now it is freely accepted to have contradicting obligations thus we accept the situation where we arrive in a state where a violation will surely occur. This will however allow us to reason about which obligation should we violate with some form of preference, for example depending on the liability or reparation that has to be done.

3.1.2 Paradoxes of Deontic Logic

Deontic logic can be seen as a logic plagued with paradoxes and it is because of this that many logicians have given up on the practicality of this logic. However we believe that if we restrict the domain of the logic, for example in our case, the domain of electronic contracts, we may have a logic that is free from these paradoxes but yet expressive enough for our purposes. We shall now go through a list of the most famous deontic paradoxes.

Paradoxes in this field may be described as logical expressions that are valid in the logical system for deontic reasoning but which unfortunately is counterintuitive when the expression is described in an intuitive/informal manner [44]. Most other logical systems have such paradoxes but what is of note in deontic logic is that most of these paradoxes are fairly simple and may be easily described. Also, such paradoxes have been studied for years and yet, a solution for all these paradoxes is yet to be found.

Empty normative system

$$O\varphi \quad \text{(where } \varphi \text{ is a tautology, e.g. } O(\psi \vee \neg\psi))$$

In this situation, we are enforcing or obliging something that the underlying logic already does. Some authors, including von Wright [69] believe that this is an undesirable property of deontic logic. Another example is forbidding a contradiction. The act of forbidding a contradiction is not contributing

[1] Default Logic falls under the category of defeasible reasoning

anything to the meaning since a contradiction is already, to a certain extent, being forbidden by the underlying logical system and thus, is superfluous.

Ross's Paradox

$$O\varphi \rightarrow O(\varphi \vee \psi)$$

The classical interpretation of this paradox is that if we are obliged to mail a letter then this also implies that we are obliged to mail a letter or burn it. Remember that a disjunction can always be introduced since $a \rightarrow a \vee b$. However, in ordinary language, this introduction of a new possibility is not always allowed. As seen in this case this sounds paradoxical.

No free choice permission

$$(P\varphi \vee P\psi) \leftrightarrow P(\varphi \vee \psi)$$

In this paradox, we see that having the permission to do φ or having the permission to do ψ implies having the permission to do φ or ψ. Consider the situation where we are forbidden to do ψ. This will still imply that we are permitted to do either φ or ψ ($P(\varphi \vee \psi)$), however we do not really have this choice since ψ is forbidden. Thus since we expect that when $P(\varphi \vee \psi)$ holds we have a free choice of choosing either φ or ψ it is paradoxical to have this free choice taken from us. So basically, $P(\varphi \vee \psi)$ would have a natural language reading of $P\varphi \wedge P\psi$ rather than $(P\varphi \vee P\psi)$.

Penitent's paradox

$$F\varphi \rightarrow F(\varphi \wedge \psi)$$

This paradox is similar to the No free choice permission where now if we are forbidden to do something, we can also be forbidden from doing anything else. The classical example is that if we are forbidden to do a crime, then we are also forbidden from doing penitence.

Good Samaritan paradox

$$\varphi \rightarrow \psi \vdash O\varphi \rightarrow O\psi$$

This paradox says that every logical consequence of something that is obligated is itself obligated. As the title suggests the classical example is from the Bible where if the Good Samaritan helps the injured traveller implies that he has been injured. Thus if the Good Samaritan is obliged to help the injured traveller then it is also obliged that the traveller is injured. This conclusion is surely not the lesson that is being taught.

Chisholm's paradox

$$(O\varphi \wedge O(\varphi \rightarrow \psi) \wedge (\neg\varphi \rightarrow O\neg\psi) \wedge \neg\varphi) \rightarrow \bot$$

This paradox is described as being awkward and to a certain point even embarrassing by deontic logicians [44]. This paradox may be described using the example where φ stands for "go to the party" and ψ stands for "tell we are coming". So if we are obliged to go to a party then it is obliged that if we go to the party we tell that we are coming, but if we do not go we are obliged not to tell we are coming. In such a setting if we end up not going to the party it would be still ok and thus is intuitively consistent however it is inconsistent in Standard Deontic Logic (SDL).

Forrester's paradox of gentle murder

$$(F\varphi \wedge (\varphi \rightarrow O\psi) \wedge (\psi \rightarrow \varphi) \wedge \varphi) \vdash \bot$$

This time we let φ stand for "murder" and ψ stand for "murder gently". So this paradox would be exemplified with we are forbidden to murder, but if we do murder, we are obliged to murder gently. Furthermore, if we murder gently it implies that we have also murdered, and finally one murders someone. This is again inconsistent in SDL but actually makes common sense.

Conflicting obligations

$$\neg(O\varphi \wedge O\neg\varphi), \text{ or equivalently, } O\varphi \rightarrow P\varphi$$

Here we state that we do not want to have any conflicting duties. This is a desirable property. This cannot really be seen as a paradox but rather that this does not always manifest itself in real life; thus, this situation may seem unrealistic.

Derived Obligation

$$O\varphi \rightarrow O(\psi \rightarrow \varphi)$$
$$F\psi \rightarrow O(\psi \rightarrow \varphi)$$
$$\neg\varphi \rightarrow (\varphi \rightarrow O\psi)$$

These are the deontic equivalent of the paradoxes of material implication in classical logic. The paradox of the material implication of classical logic may be described using two examples. The first example is $((l \rightarrow e) \wedge (p \rightarrow f)) \vdash ((l \rightarrow f) \vee (p \rightarrow e))$ where l stands for London, e England, p Paris and f France. The interpretation is that if we are in London, then we are in England and if we are in Paris, then we are in France. This statement is sound in everyday language; however, we can end up with if we are in London then we are in France or if we are in Paris then we are in England, which is not true but holds in the classical Boolean framework. The second example is $p \wedge q \rightarrow r \vdash ((p \rightarrow r) \vee (q \rightarrow r))$ where p and q stand for switch one and switch two being on respectively and r stands for the light being on. So if both switch one and two are on, then the light is on. From this statement we may deduce logically that either if switch one is on then the light is on or if switch two is on then the light is on, which in reality is not true if the switches are in series. With these two examples we may conclude that using classical logic and taking material implication to mean if-then is an unsafe method of reasoning that may give erroneous results and this also holds for the deontic equivalents.

Thus this boils down in manner that we "read" the → (implication).

Deontic detachment

$$(O\varphi \wedge O(\varphi \rightarrow \psi)) \rightarrow O\psi$$

This states that obligations are closed under implication. This statement raises controversies that are similar to that of the Good Samaritan problem. Using this setting, we may say that the Good Samaritan is obliged to help the traveller but also that the Good Samaritan is obliged that if he helps the traveller then the traveller is injured and thus, there is the obligation for the traveller to be injured.

Kant's ought implies can

$$O\varphi \rightarrow \Diamond\varphi$$

This formula states that we may only be obliged to do things that are possible and thus, this denies the possibility to specify that one is really obligated to do the impossible. This cannot be described as a paradox; however, we should note that in real life we may be obliged to do the impossible.

Epistemic obligation

$$OK\varphi \rightarrow O\varphi$$

This paradox may be exemplified using the following instance. If we are obliged to know that a spouse is committing adultery, then it ought to be the case that the spouse is committing adultery.

Remarks on Paradoxes

After this listing of the most common paradoxes, we should note that some paradoxes, like the derived obligation, disappear when it is analysed exactly as what is stated by the formula rather than applying a looser reading of it. Also, there are certain paradoxes that cannot really be called paradoxes but rather that they exemplify an ideal world, for example Kant's 'ought implies can' restricts us to be obliged only to do possible things. However, some paradoxes are quite serious, the most of note being Chisholm's Paradox and the Good Samaritan paradox.

We shall now take a look at some uses of deontic logic in computer science.

3.1.3 Uses of Deontic Logic in Computer Science

In this section we are going to investigate how deontic logic is being used by computer scientists. This section is based on the paper "Applications of Deontic Logic in Computer Science: A Concise Overview" written by R.J. Wieringa and J.-J.Ch. Meyer [67].

As one might assume, there is a great deal of work concerning the use of deontic logic in order to perform automation of legal processes. It is quite recent that other areas are being explored where deontic logic may be employed. An example is in the description of fault-tolerant systems where we need to differentiate between ideal and actual behaviour.

Structured view of deontic applications in computer science

After a historical overview, the authors continue to give a description of the type of applications that have made use of deontic logic.

Fault-tolerant Systems Deontic logic may be used to describe the ideal and actual behaviour and thus, is used in order to specify exception handling since fault-tolerant systems need to be able to handle non ideal situations.

Normative user behaviour Using deontic logic one may specify user behaviour making a distinction between desirable and non-desirable user interaction. This can also be seen as a form of exception handling where we may easily describe what should be done in case a user does not behave in an ideal fashion while still making a distinction between ideal and exceptional behaviour.

Policy specification In this scenario deontic logic is used to make the policy of an organisation unambiguous and also to reason about different policies and analyse the repercussions. Most commonly it is the security policy that is specified using deontic logic.

Specification of Law This scenario is similar to the policy specification; however, the overall result is different. The focus is on law from the legal perspective whereas the policy specification takes a computer science flavour. This will also encompass the simulation of legal thinking where instead of simply having a set of formulas representing the law, we model the way that lawyers and judges think using deontic logic.

Looking at the way the authors split the applications of deontic logic, we may summarise that there are two main classes of applications of deontic logic. The first one concerns the representation of exception handling and the other dealing directly with the legal realm. However, there have been other uses of deontic logic like in integrity constraints of databases. There have also been more creative uses of deontic logic. It has been used in the solving of scheduling problems where the system can still handle the situations where the allocation of jobs will result in some violations of deadlines. Again, we believe that these applications may be split under these two headers.

3.2 $\mathcal{CL}$

The contract language $\mathcal{CL}$ first appeared in the paper "A Formal Language for Electronic Contracts" [55] written by Prisacariu and Schneider. We shall go through the papers and technical reports concerning $\mathcal{CL}$ in order to attempt to document the development of $\mathcal{CL}$. We deem this important since $\mathcal{CL}$ was changing while we were working on the masters and thus, affected the directions and decisions that were taken. We shall thus start with reporting the contributions of the first paper about $\mathcal{CL}$ [55].

3.2.1 A Survey of $\mathcal{CL}$

$\mathcal{CL}$ is introduced in the paper "A Formal Language for Electronic Contracts" [55]. The authors describe where such a language would fit in today's industry. It is very common today that we find applications spanning across organisational borders mainly due to the internet and the trend to develop applications using the Service Oriented Architecture. In such architectures each organisation participating will not have access to complete information about services residing outside the organisation. Most often, the organisation would not have access to the external code let alone the ability to verify that the implementation is correct. This dependability on external entities drives the need of having an agreement between the parties before collaboration starts in order to safeguard the interests of both parties. Such an agreement is usually called a contract and determines what the duties of every party are and the penalties if these duties are violated.

Not only would we desire that we have a contract between parties but also to have a contract that is machine readable. This opens up a number of opportunities, for example automatic contract negotiation and automatic monitoring. For example, the signatories of the electronic contracts may entrust a third party who will monitor the transactions and ensures that the contract is not violated. These are but a few of the possibilities of having unambiguous machine readable contracts.

There are a number of approaches to define formal contracts; however, the authors of [55] believe that the most promising approach is the one based on logics. They also add that a logic for contracts does not necessarily have to be based on some form of *deontic logic* but would always contain some form of normative notions like obligation, permission and prohibition. The task of formalising these deontic notions is not an easy task and this may be witnessed all through the history of such research starting as back as 1926 [39] (See Section 3.1 for more details and Section 3.1.1 for a more in dept history of deontic logic).

Most often, contracts contain clauses that by definition are violable and in such situation some other clause (that may be for example a penalty/reparation) is enacted [55]. These are usually referred to as *contrary-to-duty* (CTD) and *contrary-to-prohibition* (CTP) clauses. The authors also point out that other deep issues with deontic logic are the relation between the deontic notions (the duality of the operators and the definitions in terms of each other) and the understanding of their truth-value (or even if such notions have a truth value). The main concern in [55] is the formal definition of e-contracts, attempting to avoid the philosophical problems of deontic logic and focus more on the practicality of the logic. The starting point of $\mathcal{CL}$ was [9] where a fixed point characterisation of obligation, permission and prohibition is given using modal μ-calculus. In [55] we again see semantics given in an extension of the μ-calculus; however, the semantics are different from that in [9] as we shall see.

The main contribution of [55] is the definition of a contract language which has the properties that:

1. avoids the most of the classical paradoxes of deontic logic

2. enables the expression of obligations, permissions and prohibitions over concurrent actions while keeping their intuitive meaning

3. obligation of disjunctive and conjunctive actions is defined compositionally

4. the definition and the semantics of obligation do not contain action negation

5. it is possible to express CTDs and CTPs
6. has a formal semantics given in a variant of the propositional μ-calculus

Other contributions are that the paper provides new insights into how one should relate deontic notions to the context of e-contracts, provide a natural and precise interpretation of disjunctions on obligations and also extend the propositional μ-calculus with concurrent and deterministic actions.

We shall now see what properties a contract should have as described in [55].

Desirable Properties of a Language for Electronic Contracts

When it comes to desirable properties of a deontic logic, the first and foremost would be to try and avoid as much of the deontic paradoxes as possible. For a discussion on deontic paradoxes please refer to Section 3.1.2. In [55], the authors particularly refer to Ross's paradox (defined as $O(a) \rightarrow O(a+b)$) and the free choice paradox (defined as $P(a) \rightarrow P(a+b)$). They also want to disallow the disjunction between deontic modalities where they actually limit the use since they allow it in certain specific cases[2]. Conjunction of obligation should imply that we execute the obligated actions concurrently in order not to violate any of the obligations. Also, an obligation of a sequence of actions should imply the obligation of all the subsequent actions. The language should also permit the specification of reparations and also allow the definition of conditional obligations (i.e. $\varphi \rightarrow O(a)$). Obligation should imply permission but these two deontic notions should not be interdefinable. Many authors commented on this issue such as Hintikka in [26] and von Wright in [72]. However, they believe that defining permission in terms of prohibition is natural and desirable. This view is not shared by authors like Hintikka [26] and Broersen [8].

These properties need to match properties of actual electronic contracts. This is an important argument since the authors believe that the philosophical problems with deontic logics may be avoided if we restrict ourselves to only electronic contracts. We find the justification that the negation of deontic notions is not necessary in [55]. It is not natural to say that we are not obliged to do something, so $\neg O(a)$ should not occur. Furthermore, instead of saying that we are not permitted to perform action a we can say that we are forbidden to do a and vice versa so $\neg P(a) \equiv F(a)$ and $\neg F(a) \equiv P(a)$. Having the choice operator inside the forbidden operator is counter intuitive where $F(a+b)$ would usually stand for you are forbidden from doing both a and b, and thus $F(a+b)$ should be represented as $F(a) \wedge F(b)$ since as $F(a+b)$ might have the semantics that you are forbidden to have the choice

[2] The syntax of $\mathcal{CL}$ allows disjunction only in the following case: $F(a) \vee [a]F(b)$. This was originally allowed in order to define CTPs; however, in later semantics the CTPs were defined as part of the language

of performing action a or b which is not that intuitive but still possible. This ambiguity of choice and prohibition is also extended over the exclusive choice.

They also make some changes to the classical dynamic algebra [53] by removing the Kleene star (iteration) and the inclusion of concurrent actions. They believed that it is unnatural to have the Kleene star under deontic operators. Consider $O(a^*)$. This cannot be violated since any amount of action a could happen; thus, no particular information is given but is rather confusing and can be described as $O(true)$. $F(a*)$ is also counter intuitive since if we perform any amount of a we shall violate this formula and thus, this formula cannot be satisfied and may be represented as $F(true)$. They also define the negation of an action as any trace of actions that takes us outside of the negated action trace [9].

The Initial Syntax of $\mathcal{CL}$

Definition 3.2.1. *The contract language syntax is defined by:*

$$\begin{array}{rcl} Contract & := & D;C \\ C & := & \phi|C_O|C_P|C_F|C \wedge C|[\alpha]C|\langle\alpha\rangle C|CUC|\bigcirc C \\ C_O & := & O(\alpha)|C_O \oplus C_O \\ C_P & := & P(\alpha)|C_P \oplus C_P \\ C_F & := & F(\delta)|C_F \vee [\delta]C_F \end{array}$$

A contract is made up of two parts: the definitions and clauses. The definitions are left unspecified; thus, leaving the freedom to be able to define anything. These definitions are the atomic assertions and actions like for example 'the budget is more than x'. C stands for a general clause whereas C_O, C_P and C_F stand for the obligation, permission and prohibition clauses respectively. The $\wedge$,$\vee$ and $\oplus$ have the classical meaning of conjunction, disjunction and exclusive disjunction. α stands for a compound action (syntax to be given later) and δ denotes a compound action not containing any occurrence of $+$. Please note that the $\vee$ operator can only appear in certain circumstances with the prohibition deontic operator and the $\oplus$ operator can never appear between prohibitions.

The $[.]$ and $\langle.\rangle$ operators are taken from Propositional Dynamic Logic (PDL) where $[\alpha]C$ means that after performing α, C must hold whereas $\langle\alpha\langle C$ means that there must be the possibility of executing α and if it is executed C must hold afterwards.

There are also temporal logic (TL) operators [50] where C_1UC_2 means that C_1 must hold until C_2 holds and $\bigcirc C$ means that C should hold in the next moment. Using these operators we may define $\Box C$ and $\Diamond C$ to mean always and eventually.

From our experience using $\mathcal{CL}$ the Until modality gives a choice when some clause starts to apply and thus, it has the problems of free choice (internal vs. external choice) which lead to many different interpretations possible. Furthermore, since there is no notion of true and false defined in the syntax, it is not clear how we may define these temporal operators since they typically concern themselves with truth. For example consider $O(a)UF(b)$. We somehow need to know that suddenly $F(b)$ has started to be enacted and thus we are not still obliged to perform a. The Until modality has been removed from the $\mathcal{CL}$ syntax; thus, solving this issue but still allowing the same expressivity if desired by the introduction of the Kleene star to actions that are not inside deontic operators.

The following are a number of rules that show the decomposition of compound actions.

$$\begin{array}{rcl} O(\alpha + \beta) & \equiv & O(\alpha) \oplus O(\beta) \\ O(\alpha \& \beta) & \equiv & O(\alpha) \wedge O(\beta) \\ O(\alpha ; \beta) & \equiv & O(\alpha) \wedge [\alpha] O(\beta) \\ P(\alpha + \beta) & \equiv & P(\alpha) \oplus P(\beta) \\ P(\alpha ; \beta) & \equiv & P(\alpha) \wedge \langle \alpha \rangle P(\beta) \\ F(\alpha ; \beta) & \equiv & F(\alpha) \vee [\alpha] F(B) \end{array}$$

For a more in depth discussion on these composition rules we are referred to their technical report [56]. The normal form of obligations may be achieved by applying a number of rewrite rules. Using this syntax CTDs are defined as $O_C(\alpha) \equiv O(\alpha) \wedge [\overline{\alpha}]C$ whereas CTPs are defined as $F_C(\alpha) \equiv F(\alpha) \vee [\alpha]C$; however, CTPs and CTDs are incorporated as part of the language in [34].

The semantics of $\mathcal{CL}$ is given using an extension to μ-calculus called Cμ where concurrent actions together with special prepositional constants are added to the standard μ-calculus. A translation function from $\mathcal{CL}$ into Cμ is given in [55]. From these semantics, the following properties of $\mathcal{CL}$ hold:

$$\begin{array}{rcl} P(\alpha) & \equiv & \neg F(\alpha) \\ F(\alpha) & \equiv & \neg P(\alpha) \\ O(\alpha) & \rightarrow & P(\alpha) \\ P(a) & \nrightarrow & P(a \& b) \\ F(a) & \nrightarrow & F(a \& b) \\ F(a \& b) & \nrightarrow & F(a) \\ P(a \& b) & \nrightarrow & P(a) \end{array}$$

$\mathcal{CL}$ was translated into a variant of the μ-calculus because the μ-calculus is decidable [32], has a complete axiomatic system [65] and a complete Gentzen-style proof system [64].

Model Checking Contracts

$\mathcal{CL}$ could also be used in order to model check contracts [48]. The authors of "Model Checking Contracts - a case study" argue that an *e-contract* may be viewed in two different ways: "(1) The executable version of a conventional contract by translating the paper version into the electronic one; (2) As contracts by themselves obtained directly from certain software applications like web services". In [48] we see how model checking techniques may be applied in the context of electronic contracts. The method presented in [48] has seven steps:

1. translate the conventional contract written in English into $\mathcal{CL}$
2. translate syntactically $\mathcal{CL}$ into Cμ
3. obtain a Kripke-like model (labelled transition system, LTS) from Cμ formulas
4. translate LTS into the input language of NuSMV
5. perform model checking using NuSMV
6. in case of counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repair contract until the property desired is satisfied.

The syntax given in [48] is identical to [55] except that the $\Box$ is added to the syntax rather than defined by using the other operators. The method applied in [48] of how the LTS was constructed was ad hoc and furthermore, the properties verified were LTL properties rather than $\mathcal{CL}$ properties. If we could also specify $\mathcal{CL}$ clauses we would be able to verify that some contract implies another contract and more.

Run-time Monitoring of Electronic Contracts

Apart from being able to model check $\mathcal{CL}$ clauses by translating them into an LTS, one can perform run-time monitoring of electronic contracts specified in $\mathcal{CL}$ [34]. The authors of [34] focus on a subset of the $\mathcal{CL}$ semantics that is required in order to perform monitoring. For monitoring, the $\mathcal{CL}$ syntax changes slightly. The temporal operators are entirely removed ($\mathcal{U}$,$\bigcirc$ and the later $\Box$). The new syntax given in [34] is as follows:

$$\begin{array}{rcl}
C & := & C_O|C_P|C_F|C \wedge C|[\beta]C|\top|\bot \\
C_O & := & O_C(\alpha)|C_O \oplus C_O \\
C_P & := & P(\alpha)|C_P \oplus C_P \\
C_F & := & F_C(\alpha)|C_F \vee [\alpha]C_F \\
\alpha & := & 0|1|a|\alpha\&\alpha|\alpha;\alpha|\alpha+\alpha \\
\beta & := & 0|1|a|\beta\&\beta|\beta;\beta|\beta+\beta|\beta*|C?
\end{array}$$

Another difference that one should note in this new syntax is that the compound action is defined as part of the $\mathcal{CL}$ syntax. This is unlike in the previous syntax where the syntax of actions was not specified. One should also note that in this syntax the Kleene star has been added to compound actions; however, compound actions that contain the Kleene star may only be used in $[]$ and the $\langle\rangle$ operators. The authors of [34] still believe that the Kleene star is counter intuitive inside obligations, permissions or prohibitions. The Kleene star was added so that now we may encode the temporal operators using only dynamic logic constructs. Furthermore, the assertion was removed and the reparations are defined as basic operations; thus, CTDs and CTPs are modelled directly. A deep discussion about the action algebra and action negation may be found in the technical report [54].

A trace is defined as being a sequence of sets of atomic actions [34]. The canonical form may be obtained for any α that was defined with the operators +,; or &. So for any α it can be put in the form of $+_{i\in I}\alpha^i_\&;\alpha^i$ where $\alpha^i_\& \in \mathcal{A}^\&_\mathcal{B}$ and α^i is a compound action in canonical form. We should note that $\mathcal{A}_\mathcal{B}$ is the set of the basic (atomic) actions whereas $\mathcal{A}^\&_\mathcal{B}$ is the set of all possible concurrent sets built from the set of atomic actions.

The negation of an action is based on the canonical form where $\overline{\alpha} = \overline{+_{i\in I}\alpha^i_\&;\alpha^i}$ that in turn is equal to $+_{\gamma\in\overline{R}}\gamma + (+_{i\in I}\alpha^i_\&;\overline{\alpha^i})$ where set $\overline{R} = \gamma|\gamma \in \mathcal{A}^\&_B$, and $\forall i \in I, \alpha^i_\& \nsubseteq \gamma$, meaning that the set $\overline{R}$ contains all the concurrent actions γ that do not include any action $\alpha^i_\&$ for all $i \in I$.

The satisfaction relation $\vDash$ of a trace is defined as 'a trace σ is said to satisfy (not violate) a contract C if $\sigma \vDash C$' whereas a trace σ that violates the contract C would be described as $\sigma \nvDash C$. The recursive definition of the trace semantics of $\mathcal{CL}$ may be found in the following section.

3.2.2 The Syntax and Semantics of $\mathcal{CL}$

As just witnessed, the syntax and semantics have both changed a number of times. The syntax has moved away from that shown in the original paper where now the temporal operators have been substituted with the dynamic logic operators and the Kleene star in order to be able to encode the original temporal operators. When the syntax changed there was no formal definition of the full semantics of $\mathcal{CL}$ with the new syntax. However, we shall be needing the trace semantics of $\mathcal{CL}$ as we shall see in Section 5.1.1 of which we do have the semantics using the latest syntax. The following is the syntax and trace semantics that have been used throughout this work.

$$\begin{aligned}
C &:= C_O|C_P|C_F|C \wedge C|[\beta]C|\top|\bot \\
C_O &:= O_C(\alpha)|C_O \oplus C_O \\
C_P &:= P(\alpha)|C_P \oplus C_P \\
C_F &:= F_C(\alpha)|C_F \vee [\alpha]C_F \\
\alpha &:= 0|1|a|\alpha\&\alpha|\alpha;\alpha|\alpha+\alpha \\
\beta &:= 0|1|a|\beta\&\beta|\beta;\beta|\beta+\beta|\beta^*|C?
\end{aligned}$$

$$\begin{aligned}
\sigma &\vDash C_1 \wedge C_2 \text{ if } \sigma \vDash C_1 \text{ and } \sigma \vDash C_2 \\
\sigma &\vDash C_1 \vee C_2 \text{ if } \sigma \vDash C_1 \text{ or } \sigma \vDash C_2 \\
\sigma &\vDash C_1 \oplus C_2 \text{ if } (\sigma \vDash C_1 \text{ and } \sigma \nvDash C_2) \text{ or } (\sigma \nvDash C_1 \text{ and } \sigma \vDash C_2) \\
\sigma &\vDash [\alpha_\&]C \text{ if } \alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..) \vDash C, \text{ or } \alpha_\& \nsubseteq \sigma(0) \\
\sigma &\vDash [\beta;\beta']C \text{ if } \sigma \vDash [\beta][\beta']C \\
\sigma &\vDash [\beta+\beta']C \text{ if } \sigma \vDash [\beta]C \wedge \sigma \vDash [\beta']C \\
\sigma &\vDash [\beta^*]C \text{ if } \sigma \vDash C \text{ and } \sigma \vDash [\beta][\beta^*]C \\
\sigma &\vDash [C_1?]C_2 \text{ if } \sigma \nvDash C_1, \text{ or } \text{ if } \sigma \vDash C_1 \text{ and } \sigma \vDash C_2 \\
\sigma &\vDash O_C(\alpha_\&) \text{ if } \alpha_\& \subseteq \sigma(0), \text{ or } \text{ if } \sigma(1..) \vDash C \\
\sigma &\vDash O_C(\alpha;\alpha') \text{ if } \sigma \vDash O_C(\alpha) \text{ and } \sigma \vDash [\alpha]O_C(\alpha') \\
\sigma &\vDash O_C(\alpha+\alpha') \text{ if } \sigma \vDash O_\bot \text{ or } \sigma \vDash O_\bot(\alpha') \text{ or } \sigma \vDash [\overline{\alpha+\alpha'}]C \\
\sigma &\vDash F_C(\alpha_\&) \text{ if } \alpha_\& \nsubseteq \sigma(0), \text{ or } \text{ if } \alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..) \vDash C \\
\sigma &\vDash F_C(\alpha;\alpha') \text{ if } \sigma \vDash F_\bot(\alpha) \text{ or } \sigma \vDash [\alpha]F_C(\alpha') \\
\sigma &\vDash F_C(\alpha+\alpha') \text{ if } \sigma \vDash F_C(\alpha) \text{ and } \sigma \vDash F_C(\alpha') \\
\sigma &\vDash [\overline{\alpha_\&}]C \text{ if } \alpha_\& \nsubseteq \sigma(0) \text{ and } \sigma(1..) \vDash C \text{ or } \text{ if } \alpha_\& \subseteq \sigma(0) \\
\sigma &\vDash [\overline{\alpha;\alpha'}]C \text{ if } \sigma \vDash [\overline{\alpha}] \text{ and } \sigma \vDash [\alpha][\overline{\alpha'}]C] \\
\sigma &\vDash [\overline{\alpha+\alpha'}]C \text{ if } \sigma \vDash [\overline{\alpha}]C \text{ or } \sigma \vDash [\overline{\alpha'}]C
\end{aligned}$$

3.3 The Exclusive-Or and $\mathcal{CL}$

The semantics of the exclusive-or given in [34] introduces a number of issues into $\mathcal{CL}$. In [55] it is argued that one should not allow negation in front of obligations, permissions and prohibitions. We agree with this since it does not make sense to say that ‘we are not obliged to perform an action’ since it means that ‘we are free to do whatever we want’. ‘We are not permitted to do an action’ may be easily translated to ‘we are forbidden to perform an action’ and vice versa.

Recall that the exclusive-or may only appear between obligation clauses and between permission clauses and not between prohibition clauses. This was done in order to avoid paradoxes. Consider the following $O(\alpha) \oplus O(\alpha')$. This could be translated into $O(\alpha + \alpha') \wedge F(\alpha \& \alpha')$ while keeping the same meaning.

Proof. Proving that the exclusive disjunction of two obligations may be represented without using an exclusive disjunction. This is done since the exclusive disjunction of two obligations may be represented as the obligation to perform either of the actions and being forbidden to perform both:

$$\sigma \vDash O(\alpha) \oplus O(\alpha') \Leftrightarrow \sigma \vDash O(\alpha + \alpha') \wedge F(\alpha \& \alpha')$$

$$\begin{array}{ll}
 & \sigma \vDash O(\alpha) \oplus O(\alpha') \\
\Leftrightarrow & \{\text{Definition of } \oplus\} \\
 & (\sigma \vDash O(\alpha) \text{ and } \sigma \nvDash O(\alpha')) \text{ or } (\sigma \nvDash O(\alpha) \text{ and } \sigma \vDash O(\alpha')) \\
\Leftrightarrow & \{\text{Definition of } \sigma \vDash O \text{ and its negation for } \sigma \nvDash O\} \\
 & (\alpha \subseteq \sigma(0) \text{ and } \alpha' \nsubseteq \sigma(0)) \text{ or } (\alpha \nsubseteq \sigma(0) \text{ and } \alpha' \subseteq \sigma(0)) \\
\Leftrightarrow & \{(p \wedge \neg q) \vee (\neg p \wedge q) \Leftrightarrow (p \vee q) \wedge \neg(p \wedge q)\} \\
 & (\alpha \subseteq \sigma(0) \text{ or } \alpha' \subseteq \sigma(0)) \wedge (\alpha \cup \alpha' \nsubseteq \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of Obligation and Prohibition}\} \\
 & O(\alpha + \alpha') \wedge F(\alpha \& \alpha')
\end{array}$$

□

This might be a clean way to remove the exclusive disjunction; however, when one adds CTDs the problem complicates itself — one may even end up with an exclusive-or between any of the deontic notions which is not allowed. Consider $O_{F(a)}(0) \oplus O_{P(b)}(0)$. Since 0 is the impossible action, both obligations will fail. Since the obligations are violated, the reparations come into effect; however, they are still under the exclusive disjunction effectively having the exclusive disjunction between the reparations. Thus, this clause could have been represented as $[1](F(a) \oplus P(b))$, which is undesirable and also not part of the $\mathcal{CL}$ syntax.

Furthermore, one could also define a formula equivalent to a form of negation on obligations. Using the exclusive-or we could define a clause which will effectively represent the clause that we need to violate an obligation in order to satisfy the contract. This has the same philosophical problems as the negation of deontic operators. Consider $O(1) \oplus O(b)$. Since the first obligation will always be satisfied the clause could have been represented as $F(b)$ since if b is observed then both obligations would have been satisfied thus violating the exclusive disjunction. Please note that this is not equivalent to “We are not obliged to perform b” but rather “We are obliged to violate the obligation to perform b” or “We cannot satisfy the obligation to perform b”.

The exclusive-or as defined in these papers gives space to express a number of clauses that do not have a clear meaning and can also be described as paradoxical. One should be very careful when using the exclusive operator and it appears that this operator philosophically solves certain classical problems but introduces new ones as well.

3.4 Prohibition of Choice

The trace semantics of $\mathcal{CL}$ for the prohibition of choice is defined as follows:

$$\sigma \vDash F_C(\alpha + \alpha') \text{ if } F_C(\alpha) \text{ and } F_C(\alpha')$$

This semantics does capture the required meaning. For example, if we are forbidden from performing actions a or b, then we are forbidden to perform both. The question we would like to raise is about the behaviour of the reparation. Consider $F_C(a+b)$. If we perform a we want the reparation C to hold, if we perform b then we want the reparation C to hold and if we perform both a and b we also want the reparation C to hold. This is the behaviour one would expect from $F_C(a+b)$ and also the behaviour obtained by using the semantics $F_C(a)$ and $F_C(b)$. However now consider $F_C(a + (b \cdot c))$. This formula, using the semantics, would be equivalent to $F_C(a) \wedge [b]F_C(c)$. So if we perform a then the reparation C must hold due to the breaking of the first prohibition. However, if we performed $a\&b$ followed by c then the reparation would have to hold twice, first due to the breaking of the prohibition of performing a and then in the following step due to the prohibition of performing c.

As just seen, when violating the clause $F_C(a + (b \cdot c))$ with the trace $(a\&b) \cdot c$ we are ending up having to satisfy the reparation twice after each other. This however is counter intuitive. From the structure of the formula, since the reparation is tied directly to the violation of a single clause, one would expect for it to occur only once, even though there are more than a single way to violate the clause.

The semantics of this representation would be slightly more complicated:

$$\sigma \vDash F_C(\alpha + \alpha') \text{ if } (F_\perp(\alpha) \text{ and } F_\perp(\alpha')) \text{ or } [\alpha + \alpha']C$$

However, one should note that its semantics is similar to that of the obligation of choices. Also, we still cannot perform neither α nor α' in order not to violate the contract; however, we have grouped the reparation such that if either is violated the reparation is performed only once rather than possibly multiple times.

3.5 Related Work

This section follows the structure of the related work found in the initial paper about $\mathcal{CL}$ [55]. We did not find any new work in the field significant enough to show the diverse views that one may take in order to tackle the problem of reasoning about contracts.

Most related work in this field takes a different approach at attempting to arrive at a definition of a contract in a formal manner. Unfortunately none have really reached a state so as to be a candidate for the solution of the problem of formally defining a contract. Certain solutions have attractive semantics with a proof system or model theory but then no machanism for monitoring while other solutions provide a good method for monitoring but then do not have a formal semantics or a reasoning system. Solution based on deontic logic typically focuses too much on the logical properties forgetting the practical side, which is the opposite approach taken for the creation of $\mathcal{CL}$.

The similar formalisms are based on either of the following:

- Models of computation:
    - Finite State Machines (FSMs). In [11] the authors describe how one may generate an FSM directly from a business contract thus removing any form of ambiguity and also permit the monitoring of the contract. They also suggest of employing such a monitoring system inside the B2BObjects, which is a distributed object middleware in order to monitor the interactions between parties. However they assume that the contracts are already in place and do not consider the notion of negotiation or the analysis of properties but rather the translation. If we look at $\mathcal{CL}$ once we have translated the contract into $\mathcal{CL}$ we may generate automatically the FSM [3] while we are also still capable of analysing the contract and reason about the contract rather than just the FSM.
    - Petri Nets. As seen in [16], in spirit it is similar to FSMs but a different modelling method is used. An attractive thing about using Petri Nets is that there are many tools that help us create Petri Nets and also tools to perform verification. In fact the author explicitly says that a main reason of the choice of using Petri Nets was due to the fact of the existence and availability of these tools. Unfortunately there is still the task of converting a contract from textual form into these models that are still performed in an ad-hoc manner even if guidelines are provided.
- Logic-based:

[3] The automatic generation of the automaton from $\mathcal{CL}$ is one of the results of this work

- Classical Logic. The approach taken in [18] is quite interesting. Their domain is that of Semantic Web Services and focus on the specification of both process modelling and that of contracting services. They make use of classical first-order logic together with temporal logic [20] and concurrent transaction logic [7]. They also include concepts taken from game theory [47] since a contract usually involves different parties that might have different and possibly conflicting goals or priorities. They provide both a model theory and also a proof theory which make it quite a nice alternative to the negotiation part of contracting. They also provide outlines of algorithms required in order to solve the constraints of the temporal workflow.
- Modal Logic [17], where their main concern is the performance or execution of contracts in the Business-to-Business setting. The authors use both modal action logic and also deontic action logic; thus, this paper bridges between modal and deontic logic. However the model developed is typical for Modal Logic rather than deontic. In fact they make use of an adapted notation which is a modal language, which is typically used for transition systems for representation. Looking at the results of this paper, their focus is on the creation of the transition system that represents the contract and thus, not as high level as a contractual representation.
- Deontic logic. In [25] the authors take a theoretical deontic approach to what a CTD is. They describe a CTD as an exceptional represenation; thus, they do not look at a CTD as conflicting obligation that overrides the primary obligation but rather as a logic exception of the primary obligation. So now an obligation is looked at as a sequence of formulae where if the first one is not satisfied, then the second one should be satisfied, and if the second one is not satisfied either, then the third and so on. Using their approach they show their solution to a number of paradoxes, most importantly Chisholm's Paradox and the Gentle Murder.
- Defeasible logic. A very important feature in defeasible logic is its nonmonotonicity. Having conflicts or contradictions in the formulas is not a problem in defeasible logic and such situations are resolved by some form of priority or "scepticism". In [24] they make use of defeasible logic in order to represent contracts and also perform not only monitoring but also some form of analysis. They make use of an extension of RuleML to translate the contracts into a machine readable form and their representation also has flavours from deontic logic. They believe that the use of deontic logic is important since it allows "ease of expression and comprehension" and also "clear and intuitive semantics". RuleML was

chosen because it allows the execution and exchange of rules between systems and is also the expected method to define rules on the web and distributed systems. Once the contract is described in their suggested extension of RuleML we could automatically monitor and to a certain extent perform certain conflict analysis. In [60] they then show how they can nest rules in order to have a more hierarchical structure making it easy to handle larger contracts.

I believe that defeasible logic has a number of desirable features for contract representation, especially default logic since one could look at a contract as a set of default behaviour and a set of exceptions that "break" this default behaviour; thus, pairing up default logic together with deontic logic would probably have interesting results.

- Based on contract taxonomies. This approach is quite different than all the above since its focus is not automation but rather how to structure contracts and contractual facts. An example of such usage is in [5] where they focus on how one can incorporate contractual information into components. Their approach shows how one can incorporate contractual information inside the code of the components (in the form for example of interfaces) and also how to manage the information.

$\mathcal{CL}$ is close to the notion of using a propositional constant in an action-based logic for giving the semantics to the deontic operators. This was first presented in [43], where a special constant V was used to denote a state where a violation has occured. In the case of $\mathcal{CL}$ V will correspond to F_a in the original semantics; however, in this work we shall make use of V for the violation and F_a will label a state where we are forbidden to perform a. In $\mathcal{CL}$ we also have the constant O_a. This allows us to define obligation not in terms of action negation; thus, deviating $\mathcal{CL}$ from other approaches like in [43]

$\mathcal{CL}$ is mostly related to the approach taken by Broersen et al [9]. Broersen makes use of μ^a-calculus in order to define the deontic notions over regular expressions on actions. Thus they take the ought-to-do approach on regular actions similar to $\mathcal{CL}$; however, they do not have a notion of contract languages but rather the characterisation of obligations, permissions and prohibitions in the μ^a-calculus. Furthermore, they define only one deontic primitive, the permission over atomic actions and derive the rest from this definition. This, resulted in the obligation to be defined using an infinite conjunction of all violating action. This infinite conjunction is avoided by defining obligation and prohibition as the deontic primitives and define permission as the negation of prohibition. Broersen et al also make use of the Kleene star that we believe is unnatural to be used inside the deontic operators. However it is permitted to make use of the Kleene star when describing actions outside of the deontic notions. The obligation on the choice of actions is not compositional in the logic presented in [9] but in the case of $\mathcal{CL}$ it is. Also, $\mathcal{CL}$ has concurrent actions whereas Broersen's approach does not. Another differentiating feature is that Broersen makes use of disjunction over deontic operators, whereas $\mathcal{CL}$ makes use of the exclusive disjunction. The negation is defined in a similar manner (the negation of an action means not performing the action but rather performing something else). However, at the $\mathcal{CL}$ language level the number of actions is finite whereas in Broersen's approach it is infinite. In order to express CTDs in Broersen's approach we shall need to extend the μ^a-calculus and possibly change the definitions slightly since as we explained previously, Broersen's basic deontic operator is the permission. In the case of $\mathcal{CL}$ both CTDs and CTPs may be easily defined. The semantics of the deontic notions in the case of Broersen is given over traces whereas in the original $\mathcal{CL}$ this is given over a Kripke style structure. The semantics of $\mathcal{CL}$ are also later given over a trace semantics but it is, however, not the complete semantics of $\mathcal{CL}$.

3.6 Conclusion

In this chapter we have started with a brief history of deontic logic that depicts the complex problems faced when one attempts to formalise deontic notions. We then went through a number of typical paradoxes of deontic logic and a brief overview of uses of deontic logic. We then presented $\mathcal{CL}$ showing how $\mathcal{CL}$ was changed until the current syntax and semantics which have been used in this work. This was then followed by discussing issues present in the current version of the language and some related approaches similar to $\mathcal{CL}$.

In the next chapter we shall compare $\mathcal{CL}$ to other specification methods making use of the CoCoME case study.

Chapter 4

Comparing $\mathcal{CL}$ to Other Specification Approaches

In this chapter we shall tackle a case study that is part of the CoCoME project. We shall specify just a fragment of the whole case study in order to be able to compare different ways of specifying the case study. We shall be comparing between temporal logic specification and deontic specification. Part of the work in this chapter is to be published with an addition of operational semantics together with Anders P. Ravn and Joseph Okika from the university of Aalborg, Gerardo Schneider from the University of Oslo and Gordon J. Pace from the university of Malta.

4.1 CoCoME Overview

Our example is taken from a larger case study — CoCoME (*The Common Component Modelling Example*) experiment [59]. The aim of the CoCoME project was to compare and contrast a number of different component models and description techniques using the same application as an example.

The chosen example was that of a Point-of-Sale (POS) system. The main purpose of the POS is to record the sales and handle the payments. The case study tackles a distributed environment having external services like the Banks and Suppliers. There are also a number of Quality of Service (QoS) requirements.

This case study is made up of a number of use cases but we shall be mostly focusing on Use Case one and two since these are the ones that have the most actors taking part. The other Use Cases are generally inhouse and just involve a few actors where the focus is mostly on performance behaviour. For our intents and purposes we shall focus just on the behavioural aspects of the case study since we want to show that $\mathcal{CL}$ is very good with regards to the specification of exceptional behaviour.

We shall focus on the following specification that is taken from the verbal description of the use cases.

F1 If the customer chooses to pay by card he is obliged to swipe the card followed by entering the correct pin number. If the pin number is incorrect the customer has two more attempts at entering the correct pin after which the client is obliged to pay with cash. If the client refrains to pay with cash the client has to give up the goods. See transition diagram in Figure 4.1.

F2 While in normal mode, the cashier may choose to switch in express mode if in the last hour 50% of the sales had less than eight items (conditionMet). Once in express mode the cashier is obliged to eventually go back to normal mode. If conditionMet holds infinitely often, then the cashier should change to express mode infinitely often. See transition diagram in Figure 4.2.

F3 In express mode, once a sale has commenced, the cashier is obliged to service customers with less than eight items. To service a customer, the items need to be entered in the system and then finish the sale. If a customer has more than eight items then it is up to the cashier's discretion whether to service the client or send him or her back to the end of another line. See transition diagram in Figure 4.3.

Clause F1 describes what is to happen when the card payment is chosen. In such a situation there are two possible choices, either enter the correct pin that would end the sale or else enter the wrong pin, after which the same two possibilities occur. The client has the possibility to enter three wrong pins after which he has the choice of either paying by cash or returning the goods.

Clause F2 describes under which conditions the cashier can change from normal mode to express mode. It includes interesting aspects as permissions, obligations and fairness constraints. In Figure 4.2 the left most state is decorated with a black circle to indicate that the state should be visited infinitely often. This models the part of the clause which states that the cashier is obliged to always eventually go back to normal mode. From the normal state we may only exit when the express condition is met, after which the cashier has the choice of going back to normal mode or express mode. The dashed transition signifies that if this transition is taken infinitely often then the dotted transition needs to be also taken infinitely often, modelling the part of the clause stating that if the condition is met infinitely often then the cash desk needs to infinitely often go into express mode.

Clause F3 describes how the cashier should act when in express mode, where once in express mode, the cashier is obliged to service a client if he has less than eight items since he is following the express desk requirement.

However, if the client has more than eight items then the cashier has the choice of either servicing the client or sending the client back to another queue to a desk that is not in express mode. This choice is up to the cashier's judgment since there are situations where it would be wise to still service customers with more than eight items, for example when some of the customers were already waiting in the queue with more than eight items at the moment of changing the desk to express mode. This 'permission' to the cashier to 'violate' the rule can be seen as an allowed explicit exception.

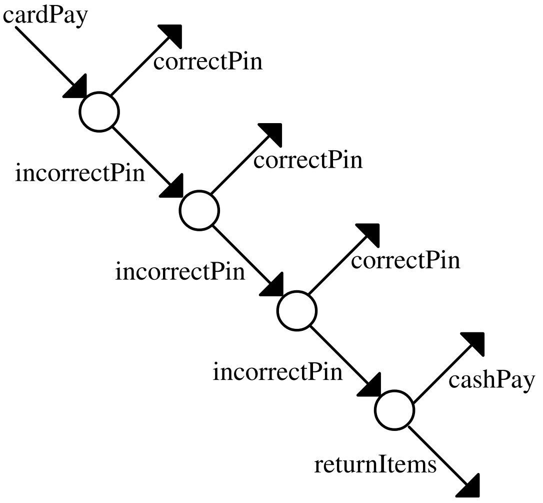


Figure 4.1: Full transition diagram for cardPay (F1)

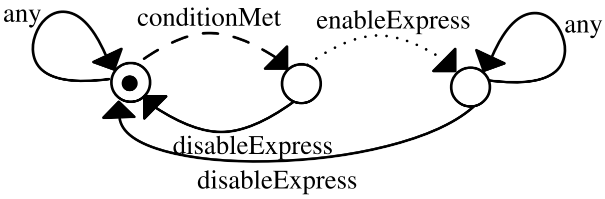


Figure 4.2: Transition diagram for Express mode (F2)

4.2 Specification using Temporal Logics

CTL and LTL are widely used in order to specify properties of systems and both were designed with computer systems in mind as described in Section 2.3 about temporal logics. We shall now attempt to specify all the three properties using both these logics.

Specification of F1 The first clause may be seen as a list of conditional

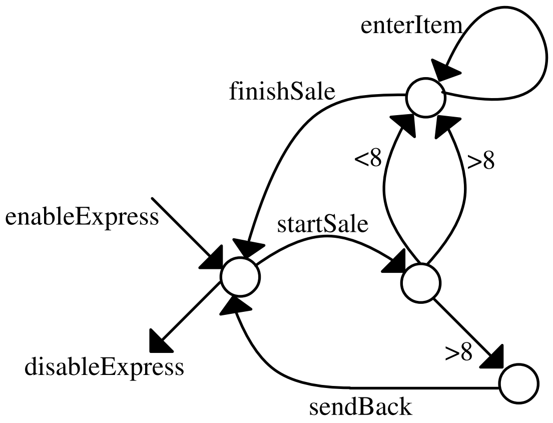


Figure 4.3: Transition diagram for sales process (F3)

statements where it is always the case that after the card is swiped then there is a choice of either entering the correct pin, in which case it would satisfy the formula or else it could be satisfied in the next step. In the next step we repeat the possibility of satisfying the formula by entering the correct pin and if not we again check the next step. This formula may be described in both CTL and LTL:

$$
\begin{aligned}
\mathbf{AG}(\text{cardPay} \rightarrow \mathbf{AX}\,(&\text{correctPin} \vee \\
&\mathbf{AX}(\text{correctPin} \vee \\
&\;\mathbf{AX}(\text{correctPin} \vee \\
&\;\;\mathbf{AX}(\text{cashPay} \vee \text{returnItems})))))
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{G}(\text{cardPay} \rightarrow \mathbf{X}(&\text{correctPin} \vee \\
&\mathbf{X}(\text{correctPin} \vee \\
&\;\mathbf{X}(\text{correctPin} \vee \\
&\;\;\mathbf{X}(\text{cashPay} \vee \text{returnItems})))))
\end{aligned}
$$

Specification of F2 The second clause cannot be described using CTL due to the fairness, unless the logic is extended with fairness constraints. We should note that we may model check CTL formulas with fairness constraints; however, the fairness constraints cannot themselves be expressed in CTL. Moreover, it is not clear how the permissions and obligations of the clause could faithfully be represented in CTL. Fairness is expressible using LTL; however, the clause also requires the existence of the transition leading to express mode which cannot be represented using LTL. Briefly, the combination of fairness, with permissions and obligations, makes this clause difficult to be handled in temporal logics and in this case cannot be represented in neither

LTL nor CTL.

Specification of F3 For the third clause it is always the case that once we go to express mode then we need to satisfy the express mode behaviour until we go back to normal mode. Once a sale is started the client needs to be serviced until the sale is finished or the client is sent to another line. If the client has less than eight items then that implies that he should be serviced, otherwise the cashier has to choose between either servicing the customer or sending the customer back. We are also ensuring that there exists the possibility of both servicing the customer and sending the customer back since this is required by the clause. It is because of this requirement that the behaviour cannot be expressed using LTL. However, in CTL it is:

$$\begin{array}{l}
\mathbf{AG}\,\text{enableExpress} \rightarrow \\
\quad \mathbf{AX}(\text{startSale} \rightarrow \\
\quad\quad \mathbf{AX}(\\
\quad\quad\quad (< 8 \rightarrow \mathbf{AX}(\text{enterItem}\mathbf{AU}\text{finishSale})) \wedge \\
\quad\quad\quad (> 8 \rightarrow \\
\quad\quad\quad\quad \mathbf{AX}(\text{enterItem} \vee \text{sendBack}) \wedge \\
\quad\quad\quad\quad \mathbf{EX}(\text{enterItem}) \wedge \\
\quad\quad\quad\quad \mathbf{EX}(\text{sendBack})) \\
\quad\quad\quad\quad \mathbf{AX}(\text{enterItem} \rightarrow \text{enterItem}\mathbf{AU}\text{finishSale})) \\
\quad\quad) \\
\quad\quad \mathbf{AU}\,\text{disableExpress})
\end{array}$$

4.3 Specification of Properties using $\mathcal{CL}$

We shall make use of $\mathcal{CL}$ as deontic specification. For a detailed description of $\mathcal{CL}$ please refer to Chapter 3.

Specification of F1 For this case we make extensive use of the CTD construct where we have a number of options of how the client may satisfy the payment by card. Once a card is swiped then the client is obliged to enter the correct pin (primary obligation). However, if the entered pin is incorrect then the client may still try again two times (secondary obligation) and in case of failure the exceptional cases of paying by cash or returning the items must be enforced. If none is satisfied, the contract is violated.

$$\Box[\text{cardPay}]\ O(correctPin)_{O(correctPin)_{O(correctPin)_{O(cashPay+returnItems)}}}$$

One should note that using CTDs orders the desirability of the path taken. Looking at Figure 4.1, if the client pays by cash after entering the wrong pin three times is just as viable as entering of the correct pin in the first try and there is no priority as to which path is most desirable. However, when we read the clause it is implicit that it is first and foremost desired to enter the correct pin and the other options are there as reparations.

Specification of F2 Clause F2 starts by stating that the cashier is infinitely often obliged to go to the normal mode: it can never stay in express mode forever. Then we state that it is always the case that after conditionMet is observed (possibility to enable the express mode) then the cashier is obliged to either choose to stay in normal mode or express.

$$\Box\Diamond O(\text{disableExpress})\wedge$$
$$\Box([\text{conditionMet}](O(disableExpress + enableExpress)\wedge$$
$$P(enableExpress))) \wedge \Box\Diamond[\text{conditionMet}] \Box \Diamond O(enableExpress)$$

We also enforce that, once the condition is met, the cashier has the possibility to go to express mode to avoid a model that only returns to normal mode. We do not need to explicitly ensure that there is a possibility to choose to stay in normal mode in a similar way to what we have done with the express mode. We also do not need to ensure that after being in express mode we have the possibility to go back to normal mode. We do not need to make these checks because of the first conjunct which states that we have to go infinitely often to normal mode. The fairness requirement is specified in the final part of the clause where we say that if we infinitely often observe conditionMet, then we shall infinitely often be obliged to go back into express mode.

Specification of F3 It is always the case that once we go to the express mode a certain behaviour needs to be followed until we go back to normal mode. In the case that the client has less than eight items, then the cashier is obliged to service the customer. However, if the client has more than eight items the cashier is obliged to choose to either service the customer or send back the customer to another cash desk and both possibilities should exist. The last property is specified in $\mathcal{CL}$ as follows:

4.4 Comparing between specifications

In Table 4.1 we present a summary of which formulae may be expressed by the Temporal and Deontic Specifications we just used.

$$\begin{array}{l}
\Box(\ [\mathit{enableExpress}](\\
\quad [\mathit{startSale}](\\
\quad\quad [<8]O(\mathit{enterItem})\mathbf{U}\text{finishSale} \wedge\\
\quad\quad [>8](\\
\quad\quad\quad O(\mathit{enterItem} + \mathit{sendBack}) \wedge\\
\quad\quad\quad P(\mathit{sendBack}) \wedge\\
\quad\quad\quad P(\mathit{enterItem}) \wedge\\
\quad\quad\quad [\mathit{enterItem}]O(\mathit{enterItem})\mathbf{U}\text{finishSale}\ )\\
\quad)\ \mathbf{U}\ \text{disableExpress}\ )
\end{array}$$

Table 4.1: Comparisons between specifications

	LTL	CTL	$\mathcal{CL}$
F1	✓	✓	✓
F2	–	–	✓
F3	–	✓	✓

The specification of the example using the different notations shows that CTL allows the specification of the branching behaviour aspect of a contract, mainly permission, which cannot be specified in LTL. However, CTL cannot express global properties such as fairness or liveness which, on the other hand, could be done making use of LTL.

As it is well known, LTL is linear time, while CTL is branching time. $\mathcal{CL}$ combines both linear and branching aspects, with the addition of deontic notions. It has not only information of what actions are to be done to satisfy the $\mathcal{CL}$ clause but also prescriptive information about the action. Namely whenever the action is observable, it is possible to distinguish whether it was required to perform it (as a primary obligation), whether it was a reparation to an obligation, or simply a permitted action.

Moreover, the expression of CTDs and CTPs in terms of basic $\mathcal{CL}$ goes beyond syntactic rewriting, since it still enables a contractual view of when obligations, permissions and prohibitions are active, have been satisfied, or violated. The main advantage of viewing the properties as a deontic contract is that this knowledge is preserved and can be reasoned about. As we noted earlier, by just looking at the Figure 4.1 , there is no information about which would be the most desirable path to take. This also holds when we represent the clause using both CTL and LTL. All the deontic information is lost and thus, we do not know if when performing an action it is actually to satisfy the initial obligation or to satisfy the reparation. This will prevent us from reasoning about these deontic notions.

F2 seems to be relatively complex property and is difficult to be captured in specifications using temporal logics. Deontic specifications seem to be appropriate whenever a right combination of deontic operators with temporal ones are required.

Both LTL and CTL can be used to specify formulas to be verified using model checking [14, 27]. Unfortunately $\mathcal{CL}$ cannot be yet model checked. In this work we develop a method in order to check for deontic inconsistencies. In this way, given a $\mathcal{CL}$ contract, we are able to detect whether the contract contains contradictory obligations, or an obligation and a prohibition to do an action at the same time, and other kinds of contradictions A general model checker for $\mathcal{CL}$ is currently under development, though by using a semantic encoding into an extended μ-calculus [55] it is possible to model check contracts written in $\mathcal{CL}$ as presented in [48].

Let us take an additional example to the 3 clauses seen so far. Consider the contract $[a]O(c) \wedge [b]F(c)$ that is satisfiable except when the concurrent action $a\&b$ is observed: we end in a state where the contract cannot be satisfied since c is both forbidden and required to take place. Using LTL we may find contracts that are not satisfiable. However, in order to identify satisfiable contracts that have conflicts, we would require the modelling of the deontic notions. This is obviously overkill when one can directly use $\mathcal{CL}$. We could encode the $\mathcal{CL}$ trace semantics into LTL; however, the correct encoding of the deontic notions so as to be able to model check contract inconsistencies would be extremely difficult. Moreover, in order to handle the above small example, CTL and LTL should be extended with concurrent actions and a priority order among actions (this is already built-in $\mathcal{CL}$–see [55]).

4.5 Modelling CoCoME

Apart from looking at $\mathcal{CL}$ as a way to specify properties, we may look at $\mathcal{CL}$ as a way to define a model. We may look at a contract as though it is a model of a system that does not violate the contract itself. If we can generate a model from a $\mathcal{CL}$ contract we may then use it to model check the contract itself. This has already been investigated in [48]. We again made use of this case study in order to compare the modelling possibilities. Apart from $\mathcal{CL}$ we also made use of SMV and UPPAAL, both of which are very common model checkers. We shall first introduce both modelling methods, followed by a comparison.

4.5.1 Introduction to SMV

SMV or Symbolic Model Validation was developed by Dr. K.L.McMillan as part of his doctoral thesis [41]. With this tool one may describe (model) a system, specify a number of desired properties and assert these properties. Two other flavours of SMV have also been fashioned. NuSMV is a reimplementation of the original SMV as in [41] with some extensions. The other flavour is Cadence SMV that is targeted more towards the industry and has an expanded language. The main difference is that the Cadence SMV makes use of LTL whilst the original SMV makes use of CTL and NuSMV supports both LTL and CTL. Another attractive feature in NuSMV is that the model could be generated as an automaton directly rather then encoding it in the language.

The SMV language

Similar to c, the SMV language has a main module that the tool will execute first. Another thing to note is that all lines are executed simultaneously since the model is translated into a truth statement by the SMV tool and evaluated. Because of this, a variable can only have a single assignment at any one step, that is, we cannot set x to have a value of 1 and in the same block set it to 2.

In order to be able to affect the next iteration, we have to make use of the next operator which can be seen as a delay, that is, next(x)=1 means that in the next iteration x will have a value of 1. A number of conditional constructs are available to help the programmer structure the model like the classic "if then else" and "switch" construct. The "default" construct is another conditional construct that is made up of two code blocks and is used to set the default value of any unassigned variables. This construct is also useful to introduce priorities to code, since the code in the second block of the default has a higher priority than the default in the first block.

Some other useful constructs are loops that are syntactic sugar to help define models more concisely. These loops are expanded before SMV starts the model checking.

Another very useful construct is the set comprehension. This is used extensively in the models we have developed since it makes it easy to make non-deterministic assignments from a particular set and also used to check for a particular value across an array. $f(i) : i = x..y$ where $f(i)$ will be some expression and $x..y$ is an integer range, that will be expanded as $f(x), .., f(y)$. For example $i * 2 : i = 1..4$ will generate $2, 4, 6, 8$, the set of even numbers from 2 till 8 inclusive. Another extension to this is the conditional construct, $f(i) : i = x..y, c(i)$. For $f(i)$ to be added to the set, $c(i)$ must be true.

Finally there are the specification constructs with which we can assert specifications using the "assert" keyword. The code $foo : assertp$ will make sure that the property p is true for every iteration of the model. The property p can be any LTL formula using the **X G F U** operators and standard Boolean operators. A more in depth description of LTL may be found in Section 2.3.

This was just a brief introduction of the SMV language so as to be able to understand the code that follows. For a more detailed description please check the documentation supplied with the SMV tool [42].

4.5.2 Introduction to UPPAAL

UPPAAL[1] has been a joint project between the Swedish Uppsala University and the Danish Aalborg University. UPPAAL is described on the website as "an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)."

UPPAAL is mainly directed towards real-time controllers and communication protocols since we may model a number of non-deterministic processes with a finite control structure and real-valued clocks that communicate through channels or shared variables. Furthermore, this approach is ideal to model concurrent applications using message passing (not only shared variables). Thus, it might be easier to use in certain situations instead of SMV since the communication issues between the different processes might be taken care of inherintly by UPPAAL unlike in SMV where these had to modelled implicitly.

The tools supplied with the model checker are quite extensive and the graphical user interface make UPPAAL quite easy to use. The GUI tool is not only very useful during the modelling of the system but also makes it very easy to go through the traces returned by the model checker.

Another interesting feature is that we may simulate the model. This is mostly helpful in the initial stages when the model is being developed.

In order to get a good foundation to understand the inner workings of UPPAAL and in which situations it would be very useful, one should refer to [4]. For a quick and to the point introduction to UPPAAL one should refer to [36]. Even though this paper is not the latest and there have been a number of improvements, it still is a good introduction. For an even recent tutorial paper one should refer to [3]. The rest of this introduction is a brief summary of this paper.

UPPAAL is based on the theory of timed automata. For a description of Timed Automata please refer to [3]. UPPAAL has a number of extensions to TA like guards and channels in order to make modelling easier.

The Query language used by UPPAAL to specify properties is a subset of CTL. The difference from CTL is that UPPAAL does not allow nesting of path formulae. UPPAAL supports path formulae that may be classified into reachability, safety and liveness. As part of state formulae, UPPAAL has a special state formula consisting of the keyword deadlock that is satisfied in all deadlock states. Another interesting point is how the liveness properties are described using the notation $\varphi \leadsto \psi$ or in UPPAAL syntax $\varphi - - > \psi$. This is equivalent to the CTL formula $A\,\Box\,(\varphi \Rightarrow \Diamond\psi)$. So UPPAAL does, to a certain extent, support nested path quantifiers.

[1] UPPAAL can be obtained from http://www.UPPAAL.com. The website also has a number of very useful resources

UPPAAL also has three types of locations: normal, urgent and committed locations. When in a committed location, the next action will be to move out of the committed state. In an urgent state, time will not pass while in that state but interleaving with normal states may occur. An urgent state may be seen as a state with the invariant $y <= 0$.

[3] portrays a number of modelling patterns that are found very useful when modelling with UPPAAL.

Variable Reduction One should reset variables when they are not needed in order to reduce the state space.

Synchronous Value Passing A model to send values from one process to another in a synchronous way.

Atomicity One should reduce any interleavings that will not affect the correctness of the model in order to reduce the state space.

Urgent Edges This model shows how to make edges urgent even though they are not directly supported in UPPAAL.

Timers Model to emulate a timer.

Bounded Liveness Checking It is similar to Liveness checking but now we have an upperbound of when a certain state needs to be reached. This makes the verification problem easier but we shall have to make some changes to the model.

Abstraction and Simulation Abstraction will permit us to verify even larger models. There is a light description of how abstraction may be applied but one should look in the Ph.D. thesis of Jensen [28] for the full details.

The material available and also the graphical user interface makes UPPAAL a very easy tool to get to grips with and immediately start modelling with. Its advanced features are quite complex and flexible; however, I believe that it has a less steep learning curve than SMV. Also, a very evident difference is that UPPAAL is centred around modelling using automata whereas SMV takes a programming language approach.

4.5.3 Comparing Modelling Methods

After modelling the same part of the case study using both SMV and UPPAAL we may compare it with modelling the case study as a contract. From our experience, using UPPAAL to model this case study was much more efficient than using SMV since UPPAAL has been designed with this kind of interaction in mind, where one has a number of modules, communicating with each other, and each module's behaviour is determined with an automaton. With SMV we had to mimic this behaviour and thus not directly out of the box.

Using the model we developed in SMV, we could verify that it contains both the LTL and CTL properties using Cadence SMV and NuSMV respectively. Using the UPPAAL model we verified that it has both the CTL properties and that it is deadlock free. Thus by modelling with these methods we automatically get the ability to model check LTL and CTL properties.

Modelling the case study using contracts made the models quite elegant since they were just formulae. Unfortunately at that point in time we had no way to analyse the contract automatically. Instead we translated the contract into an SMV model that was then used to model check the properties. This technique was used in [48] in order to model check the contract. Unfortunately, since this translation is not automatic, there is ample room for mistakes, especially since the approach taken in [48] is ad hoc. However, if we can automatically generate an automaton from the $\mathcal{CL}$ contract and then translate the automaton into either an SMV representation or an UPPAAL representation, we could leverage the model checking possibilities of these languages together with the elegance of representing the system as a contract.

4.6 Conclusion

We have just seen how specifying properties using $\mathcal{CL}$ compares with CTL and LTL. This was done by applying $\mathcal{CL}$ to the CoCoME case study and then compared how these two other specifications would be used. This was then followed by looking at $\mathcal{CL}$ as a modelling language rather than just specification of properties.

In the following chapter we shall tackle how to find conflicts in contracts expressed in $\mathcal{CL}$ automatically. We first extend the trace semantics of $\mathcal{CL}$ in order to be able to define conflicts. This is then followed by an automatic decision procedure that will analyse if a contract is conflict free or not.

Chapter 5

Conflicts in Contracts

A contract has a conflict if it may request a signatory to perform conflicting actions. An example of a conflicting contract would be one obliging us to open the door while at the same time forbidding us to do so. We may informally define a conflict as follows:

1. being obliged and forbidden of doing the same action
   (eg. $O(\mathit{opendoor}) \wedge F(\mathit{opendoor})$)

2. being permitted and forbidden of doing the same action
   (eg. $P(\mathit{opendoor}) \wedge F(\mathit{opendoor})$)

3. being obliged to do two conflicting actions
   (eg. $O(\mathit{opendoor}) \wedge O(\mathit{closedoor})$)

4. being obliged and permitted to do two conflicting actions
   (eg. $O(\mathit{opendoor}) \wedge P(\mathit{closedoor})$)

One should note that once the contract is requesting these conflicting conditions there is no way in which the contract may be satisfied. It is the contract itself that is putting us in a position where we cannot satisfy it and thus such a contract is not desired.

However, there is another type of conflict that is not as clear as the previous case if it should be allowed in a contract or not. Consider the following $\mathcal{CL}$ formula $O(a + b) \wedge F(a)$. Looking at it we realise that here there is a conflict but it can still be resolved since if we perform action b we satisfy both the obligation and the prohibition. But we should also note that this depends on the interpretation of the choice, that is, if it is internal or external. If the choice is up to the user to choose how to satisfy the obligation (either a or b) then the user does not really have any choice here. However, if the choice is imposed by the system, there will be instances where the contract will not be able to be satisfied. However, in certain situations it would be ideal to leave such "conflicts" since it would simplify the representation. Furthermore, it is typical that in contracts first the base case is defined and then further clauses are given which override the base case with the exceptional cases. This view would thus suggest that this should not be considered as a conflict. We shall refer to the first kind of “conflict” discussed in this section as strong conflict and the last type of conflict we discussed as a weak conflict in order to easily distinguish between them.

One might say that this is more of a philosophical discussion and thus if we want to be practical about it, we should find a compromise rather than say which is best. We believe that a good solution would be to identify both and then leave it up to the user to decide if he should make changes to the contract or not.

After this informal introduction to conflicts, we shall investigate what we require in order to identify conflicts in contracts. This chapter will be split in two. We first investigate what kind of semantics we require and we extend the $\mathcal{CL}$ semantics in order to be able to analyse contracts for conflicts. We shall also show that the extended semantics is still correct (sound and complete) with respect to the original semantics. The second half of the chapter will focus on conflict freedom analysis. We define what a conflict is and we also define an algorithm that we prove correct with respect to the semantics of $\mathcal{CL}$ and the definition of a conflict.

5.1 Extending the $\mathcal{CL}$ Semantics

In this section we shall first see which semantics are required in order to perform conflict analysis. Then we shall see if the current semantics, as defined in previous papers about $\mathcal{CL}$ are enough for conflict analysis. After we discuss why the current semantics are not enough for conflict analysis, we extend the semantics in order to accomodate conflict analysis and prove the correctness of the extensions with respect to the original semantics.

There are currently two semantics given to $\mathcal{CL}$. The full semantics given in [55] and the trace semantics given in [34]. The first question to ask is which of these semantics would apply for conflict analysis.

5.1.1 Are full semantics required?

Do we require the full semantics in order to verify that a contract is conflict free? The “conflict-free” property may be described as ‘given any sequence of non-violating actions, the contract execution will not end up in a state where the contract is enforcing a conflict’. Using this definition we do not really require to know that there exists the possibility of performing two different actions at this point that will take us down two different branches, but rather, that both branches will not lead us into a conflicting state. Thus, for the ‘conflict-free’ property we do not require a branching semantics, a trace semantics should be enough.

Using the trace semantics given in [34] is unfortunately not enough either. These semantics were aimed at runtime monitoring of contracts and thus cannot be used to check that a contract is conflict free since we are losing all deontic information. Without deontic information we can only check if the contract is satisfiable[1] but we cannot check that it is conflict free. If we look at the semantics given in [34], it is defined over an infinite trace, of actions, which does not contain any deontic information. Furthermore, they treat permission as being the default behaviour unless a prohibition is enforced. For conflict analysis we would however desire to have any explicit permissions identified in order to ensure that they are not in conflict with prohibitions.

5.1.2 Augmenting the trace semantics

First of all we need to define formally what a conflict is. The informal definition described above suggests that we are required to know which deontic notions apply at any point in time, given that a certain sequence of actions that have not yet violated the contract occured. As the semantics are defined right now, one can know if a trace satisfies the contract or not but cannot know what deontic notions apply in the next step. Consider the contract $[a]O(b) \wedge [c]F(b)$. Using the current semantics we cannot find if there is a conflict in this contract. We cannot even check if there exists a trace that cannot be extended anymore since every extension will lead into a violation because as the definition is, the semantics only accepts or rejects infinite traces. We do not even have information about permissions. Even though using the trace semantics we cannot know if a permission is violated or not, we still may find conflicts due to permissions if we know when a permission is being enforced by the contract.

In order to define what a conflict is formally we need to have a way to know which deontic notions apply at what time. One way of preserving the deontic information is to change the semantics slightly. This is done by adding another trace, but instead of having a set of actions as elements of the trace (as the original trace is) it will have elements from the set D_a that is defined as $\{O_a | a \in A^B\} \cup \{F_a | a \in A^B\} \cup \{P_a | a \in A^B\}$ where O_a stands for 'we are obliged to do a', F_a stands for 'we are prohibited to do a' and P_a stands for 'we are permitted to do a'. We shall refer to this trace as σ_d and this trace will preserve the deontic information required for conflict analysis.

Consider the contract $[a]P(b) \wedge [c]F(b)$. A trace that satisfies this contract would be $[\{a, c\}, a, \ldots]$. The deontic trace that will be related to this trace would be $[\emptyset, \{F_b, P_b\}, \emptyset, \ldots]$. Using the deontic trace we can find that a conflict is present in this contract since there is a deontic trace that has not led to a violation and has an element set labelled with the prohibition and

[1] if it is not satisfiable there will be no σ that satisfies the contract

permission of performing the same action.

However, we still have a small issue. Consider the contract $[a]O(b+c) \wedge [b](F(b) \wedge F(c))$. This contract has a conflict since both the possible obliged actions are violated. However, in the deontic trace we need to be able to encode this choice, and make a difference between $O(b+c)$ and $O(b\&c)$ since $O(b+c) \wedge F(b)$ does not have a conflict whereas $O(b\&c) \wedge F(b)$ does have a conflict. In order to encode this in the deontic trace, instead of the elements of σ_d being sets of elements of D_a, they will be sets of the power set of D_a, where the elements of the set are possibilities. So if we consider the contract $[a]O(b+c) \wedge [b]F(b)$ a possible trace would be $[\{a,b\},\ldots]$[2] and the deontic trace would be $[\emptyset, \{\{O_a, O_b\}, \{F_b\}\}]$ whereas for $[a]O(b+c) \wedge [b]F(b)$, given the same action trace, the deontic trace would be $[\emptyset, \{\{O_a\}, \{O_b\}, \{F_b\}\}]$.

Before continuing with the extension of the semantics we shall define a number of operators on σ_d. We may concatinate two sequences using the ; operator. Any two deontic traces are pointwise (synchronously) joined using the combine operator where we shall use the $\cup$ symbol and defined as: $\sigma_d \cup \sigma'_d = \sigma_d(0) \cup \sigma_d(0)'; \sigma_d(1) \cup \sigma_d(1)' \ldots \sigma_d(n) \cup \sigma_d(n)'$. We are assuming that the length of σ_d is equal to σ'_d. In the case of ts2, by definition both traces are infinite and thus they are always of equal length. In the following semantics we will deal with finite traces. In the case that they are not of equal length, the shorter trace may be extended with empty elements in order to make them the same length. In the semantics we shall also use the $\cup$ operator between sets in which case the classical meaning will hold. Furthermore, if α is a set of atomic actions then we shall use O_α as a shortcut for the set $\{\{O_a\} | a \in \alpha\}$. $\sigma(1..)$ is used to represent the trace from the second item onwards.

We will make use of the relationship $\vDash^d_\infty$ where $\sigma, \sigma_d \vDash^d_\infty C$ is interpreted as the infinite action trace σ satisfies the contract C and the infinite deontic trace σ_d is consistent with contract C. A deontic trace σ_d is said to be consistent with a contract given a particular trace σ if all the deontic notions which apply to the trace are part of the deontic trace. So given the contract $[a]O(b)$, a trace that satisfies the contract is $[a, b, 1\ldots]$ whereas the corresponding deontic trace would be $[\emptyset, O_b, \emptyset, \ldots]$. The new semantics, which we shall refer to as ts2, are defined as follows:

[2] There is no infinite trace starting with action $a\&b$ that satisfies this contract. We are required to fix the semantics in order to be able to capture such traces and we shall be tackling this later in this section.

Definition 5.1.1. *Trace semantics ts2:*

$$\begin{array}{lll}
\sigma,\sigma_d & \vDash^d_\infty & \top \textit{ if } \sigma_d(0)=\emptyset \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty \top \\
\sigma,\sigma_d & \nvDash^d_\infty & \bot \\
\sigma,\sigma_d & \vDash^d_\infty & C_1 \wedge C_2 \textit{ if } \sigma,\sigma'_d \vDash^d_\infty C_1 \textit{ and } \sigma,\sigma''_d \vDash^d_\infty C_2 \textit{ and } \sigma_d=\sigma'_d\cup\sigma''_d \\
\sigma,\sigma_d & \vDash^d_\infty & C_1 \oplus C_2 \textit{ if } (\sigma,\sigma_d \vDash^d_\infty C_1 \textit{ and } \sigma,\sigma_d \nvDash^d_\infty C_2) \textit{ or} \\
 & & \qquad (\sigma,\sigma_d \vDash^d_\infty C_2 \textit{ and } \sigma,\sigma_d \nvDash^d_\infty C_1) \\
\sigma,\sigma_d & \vDash^d_\infty & [\alpha_\&]C \textit{ if } \sigma_d(0)=\emptyset \textit{ and} \\
 & & \qquad (\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty C, \textit{ or } \alpha_\& \nsubseteq \sigma(0))) \\
\sigma,\sigma_d & \vDash^d_\infty & [\beta;\beta']C \textit{ if } \sigma,\sigma_d \vDash^d_\infty [\beta][\beta']C \\
\sigma,\sigma_d & \vDash^d_\infty & [\beta+\beta']C \textit{ if } \sigma,\sigma_d \vDash^d_\infty [\beta]C \wedge [\beta']C \\
\sigma,\sigma_d & \vDash^d_\infty & [\beta^*]C \textit{ if } \sigma,\sigma_d \vDash^d_\infty C \wedge [\beta][\beta^*]C \\
\sigma,\sigma_d & \vDash^d_\infty & O_C(\alpha_\&) \textit{ if } \sigma_d(0)=\{O_{\alpha_\&}\} \textit{ and} \\
 & & \qquad ((\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty \top) \textit{ or } \sigma(1..),\sigma_d(1..)\vDash^d_\infty C) \\
\sigma,\sigma_d & \vDash^d_\infty & O_C(\alpha;\alpha') \textit{ if } \sigma,\sigma_d \vDash^d_\infty O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\sigma,\sigma_d & \vDash^d_\infty & O_C(\alpha+\alpha') \textit{ if } \sigma,\sigma'_d \vDash^d_\infty O_\bot(\alpha) \textit{ or } \sigma,\sigma''_d \vDash^d_\infty O_\bot(\alpha') \textit{ or} \\
 & & \qquad (\sigma_d(0)=(\sigma'_d(0)\cup\sigma''_d(0)) \textit{ and } \sigma,\emptyset;\sigma_d(1..)\vDash^d_\infty [\overline{\alpha+\alpha'}]C) \\
\sigma,\sigma_d & \vDash^d_\infty & F_C(\alpha_\&) \textit{ if } \sigma_d(0)=F_{\alpha_\&} \textit{ and } ((\alpha_\& \nsubseteq \sigma(0) \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty \top) \\
 & & \qquad \textit{or } (\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty C)) \\
\sigma,\sigma_d & \vDash^d_\infty & F_C(\alpha;\alpha')\sigma,\sigma_d \vDash^d_\infty F_\bot(\alpha) \textit{ or } \sigma,\sigma_d \vDash^d_\infty [\alpha]F_C(\alpha') \\
\sigma,\sigma_d & \vDash^d_\infty & F_C(\alpha+\alpha') \textit{ if } \sigma,\sigma_d \vDash^d_\infty F_C(\alpha)\wedge F_C(\alpha') \\
\sigma,\sigma_d & \vDash^d_\infty & [\overline{\alpha_\&}]C \textit{ if } \sigma_d(0)=\emptyset \\
 & & \qquad \textit{and } ((\alpha_\& \nsubseteq \sigma(0) \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty C) \textit{ or } \alpha_\& \subseteq \sigma(0)) \\
\sigma,\sigma_d & \vDash^d_\infty & [\overline{\alpha;\alpha'}]C \textit{ if } \sigma,\sigma_d \vDash^d_\infty [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C] \\
\sigma,\sigma_d & \vDash^d_\infty & [\overline{\alpha+\alpha'}]C \textit{ if } \sigma_d(0)=\emptyset \textit{ and } (\sigma\sigma_d \vDash^d_\infty [\overline{\alpha}]C \textit{ or } \sigma,\sigma_d \vDash^d_\infty [\overline{\alpha'}]C)
\end{array}$$

$$\begin{array}{lll}
\sigma,\sigma_d & \vDash^d_\infty & P(\alpha) \textit{ if } \sigma_d(0)=P_{\alpha_\&} \textit{ and } \sigma(1..),\sigma_d(1..)\vDash^d_\infty \top \\
\sigma,\sigma_d & \vDash^d_\infty & P(\alpha;\alpha') \textit{ if } \sigma,\sigma_d \vDash^d_\infty P(\alpha)\wedge[\alpha]P(\alpha') \\
\sigma,\sigma_d & \vDash^d_\infty & P(\alpha+\alpha') \textit{ if } \sigma,\sigma_d \vDash^d_\infty P(\alpha)\wedge P(\alpha')
\end{array}$$

This extension has the same behaviour as the original trace semantics with the difference that now we have added the deontic trace and have certain restrictions on this deontic trace. Most of the definitions do not involve the deontic trace and thus, are identical to the original trace semantics. However the definition of $\top$, $C_1 \wedge C_s$, $[\alpha_\&]C$, $O_C(\alpha_\&)$, $O_C(\alpha+\alpha')$, $F_C(\alpha_\&)$, $[\overline{\alpha_\&}]C$ and $P_C(\alpha_\&)$ define the deontic trace and hence, have this extension over the basic definition. Furthermore, we are also defining permission.

Satisfied, Violated: The simplest definition is that for $\top$, the trivially satisfiable contract. In the original semantics this was simply true by definition; however, in our extension we require that the deontic trace is empty since the contract is satisfied and thus, we have no more deontic requirements. Hence, any trace of actions will satisfy the contract $\top$; however, the deontic trace that satisfies the contract $\top$ is the trace with only empty elements ($\forall\, i : I \cdot \sigma_d(i) = \emptyset$). $\bot$ is the unsatisfiable contract and thus there does not exist any trace that can satisfy this contract.

$$\sigma, \sigma_d \models^d_\infty \top \quad \text{if} \quad \sigma_d(0) = \emptyset \text{ and } \sigma(1..), \sigma_d(1..) \models^d_\infty \top$$

Conjunction For the definition of the conjunction we require that the traces satisfy both clauses. One should note that the deontic trace cannot satisfy both clauses unless they have the same deontic requirements, in which case they would be identical. Consider $O(a) \wedge O(b)$. This would be split in two, $O(a)$ and $O(b)$ where the trace σ needs to satisfy both. However, the deontic trace, by definition, cannot satisfy both. The deontic trace consistent with $O(a)$ would be $[\{O_a\}, \emptyset \ldots]$ whereas for $O(b)$ would be $[\{O_b\}, \emptyset \ldots]$. This is tackled by having a deontic trace that satisfies each of the sub-clauses and then both deontic traces are joined pointwise using the combine operator. Thus in this case $[\{O_a\}, \emptyset \ldots] \cup [\{O_b\}, \emptyset \ldots]$ would be equal to $[\{O_a, O_b\}, \emptyset \ldots]$ and this is the expected deontic trace consistent with $O(a) \wedge O(b)$.

$$\sigma, \sigma_d \quad \models^d_\infty \quad C_1 \wedge C_2 \text{ if } \sigma, \sigma'_d \models^d_\infty C_1 \text{ and } \sigma, \sigma''_d \models^d_\infty C_2 \text{ and } \sigma_d = \sigma'_d \cup \sigma''_d$$

Exclusive Disjunction The exclusive disjunction is defined as having the traces satisfying only one of the two sub clauses.

$$\begin{aligned} \sigma, \sigma_d \quad \models^d_\infty \quad C_1 \oplus C_2 \text{ if } & (\sigma, \sigma_d \models^d_\infty C_1 \text{ and } \sigma, \sigma_d \not\models^d_\infty C_2) \text{ or} \\ & (\sigma, \sigma_d \models^d_\infty C_2 \text{ and } \sigma, \sigma_d \not\models^d_\infty C_1) \end{aligned}$$

Conditions Since any action may be represented in the normal form defined in [55], the concurrent operator may be pushed the the innermost level and thus we first define the conditions structurally. First we define the action built only from atomic actions and the concurrent operator and with this we define the sequence and choice. As described in the trace semantics, $[\alpha_\&]C$ is trivially satisfied if $\alpha_\&$ is not observed. If $\alpha_\&$ is observed then C must hold. For the negation the converse holds. Both for the positive and negative case, the restriction on the deontic trace is that the first element is the empty set. This is the desired behaviour since $[\alpha]C$ makes no deontic restrictions at the initial point in time but if the action α is observed, the clause C is required to hold in the following step, and thus the rest of σ_d will depend on C.

$$\begin{aligned}
\sigma, \sigma_d \ &\vDash^d_\infty \ [\alpha_\&]C \text{ if } \sigma_d(0) = \emptyset \text{ and} \\
&\qquad (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C, \text{ or } \alpha_\& \nsubseteq \sigma(0))) \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\beta;\beta']C \text{ if } \sigma, \sigma_d \vDash^d_\infty [\beta][\beta']C \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\beta+\beta']C \text{ if } \sigma, \sigma_d \vDash^d_\infty [\beta]C \wedge [\beta']C \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\beta^*]C \text{ if } \sigma, \sigma_d \vDash^d_\infty C \wedge [\beta][\beta^*]C \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\overline{\alpha_\&}]C \text{ if } \sigma_d(0) = \emptyset \\
&\quad \text{and } ((\alpha_\& \nsubseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \text{ or } \alpha_\& \subseteq \sigma(0)) \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\overline{\alpha;\alpha'}]C \text{ if } \sigma, \sigma_d \vDash^d_\infty [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C] \\
\sigma, \sigma_d \ &\vDash^d_\infty \ [\overline{\alpha+\alpha'}]C \text{ if } \sigma_d(0) = \emptyset \text{ and } (\sigma\sigma_d \vDash^d_\infty [\overline{\alpha}]C \text{ or } \sigma, \sigma_d \vDash^d_\infty [\overline{\alpha'}]C)
\end{aligned}$$

Obligations Similar to the definition of conditions, obligations are defined structurally on the action expressions. In the definition of the obligation, as one might expect, we find another requirement from the deontic trace. If we are obliged to perform $\alpha_\&$ then the deontic trace at that step needs to contain the set $O_{\alpha_\&}$ ($\sigma_d(0) = O_{\alpha_\&}$). For the obligation, we also need to define the obligation of a choice. As described before, a choice will be encoded as being a set of elements of D_a. Thus similar to what we have done for the definition of the $\wedge$ operator, we have two deontic traces, one for each possibility, which is then joined using the union. One should note that this time, we use the classical union since we are only requiring that $\sigma_d(0) = \sigma'_d(0) \cup \sigma''_d(0)$. We are not combining the complete traces since we have the possibility of satisfying one or the other and thus the continuation of the deontic trace might be equal to σ'_d, σ''_d or a combination of both, depending on the trace. This will be handled in the following steps.

$$\begin{aligned}
\sigma, \sigma_d \ &\vDash^d_\infty \ O_C(\alpha_\&) \text{ if } \sigma_d(0) = \{O_{\alpha_\&}\} \text{ and} \\
&\quad ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\sigma, \sigma_d \ &\vDash^d_\infty \ O_C(\alpha;\alpha') \text{ if } \sigma, \sigma_d \vDash^d_\infty O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\sigma, \sigma_d \ &\vDash^d_\infty \ O_C(\alpha+\alpha') \text{ if } \sigma, \sigma'_d \vDash^d_\infty O_\perp(\alpha) \text{ or } \sigma, \sigma''_d \vDash^d_\infty O_\perp(\alpha') \text{ or} \\
&\quad (\sigma_d(0) = (\sigma'_d(0) \cup \sigma''_d(0)) \text{ and } \sigma, \emptyset; \sigma_d(1..) \vDash^d_\infty [\overline{\alpha+\alpha'}]C) \\
\sigma, \sigma_d \ &\vDash^d_\infty \ F_C(\alpha_\&) \text{ if } \sigma_d(0) = F_{\alpha_\&} \text{ and } ((\alpha_\& \nsubseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \\
&\qquad \text{or } (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C))
\end{aligned}$$

Prohibitions The extension made to prohibition is similar to that done to obligation, with the difference that we do not have the issues with the prohibition of choice as we have with the obligation of choice. Prohibition of choice $F(\alpha + \alpha')$ is defined as $F(\alpha) \wedge F(\alpha')$ since being prohibited from doing either of two actions is effectivly being prohibited from performing both actions.

$$\begin{aligned}
\sigma, \sigma_d \ &\vDash^d_\infty \ F_C(\alpha_\&) \text{ if } \sigma_d(0) = F_{\alpha_\&} \text{ and } ((\alpha_\& \not\subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \\
&\qquad\qquad \text{or } (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C)) \\
\sigma, \sigma_d \ &\vDash^d_\infty \ F_C(\alpha; \alpha')\sigma, \sigma_d \vDash^d_\infty F_\bot(\alpha) \text{ or } \sigma, \sigma_d \vDash^d_\infty [\alpha]F_C(\alpha') \\
\sigma, \sigma_d \ &\vDash^d_\infty \ F_C(\alpha + \alpha') \text{ if } \sigma, \sigma_d \vDash^d_\infty F_C(\alpha) \wedge F_C(\alpha')
\end{aligned}$$

Permission Permission is similar to prohibition, where the permission to perform either of two actions is defined as the permission to perform both ($P(\alpha + \alpha') = P(\alpha)$ and $P(\alpha')$). One might want to define permission of choice as $P(\alpha)$ or $P(\alpha')$ in which case this would require a definition similar to what we have done for the obligation. We believe that the first instance is more natural and this is the definition that we shall use in the rest of the work; however, it is trivial if one would want to change the definition.

The permission operator is not explicitly defined in the original semantics since as discussed previously, given a trace one cannot know if the contract has been violated or not. If we look at the definition of permission, we do not place any restrictions over the action trace but only on the deontic trace.

$$\begin{aligned}
\sigma, \sigma_d \ &\vDash^d_\infty \ P(\alpha) \text{ if } \sigma_d(0) = P_{\alpha_\&} \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top \\
\sigma, \sigma_d \ &\vDash^d_\infty \ P(\alpha; \alpha') \text{ if } \sigma, \sigma_d \vDash^d_\infty P(\alpha) \wedge [\alpha]P(\alpha') \\
\sigma, \sigma_d \ &\vDash^d_\infty \ P(\alpha + \alpha') \text{ if } \sigma, \sigma_d \vDash^d_\infty P(\alpha) \wedge P(\alpha')
\end{aligned}$$

Using ts2 we now have a way to know which deontic operators occur at what time. In fact if we want we may define the satisfaction relation of σ with respect to σ_d rather than the contract C since the contract is encoded in σ_d. One might expect that this semantics is enough in order to check for conflict freedom. We will make use of the notation $\alpha \# \alpha'$ in order to represent that α and α' are mutually exclusive and thus cannot occur at the same time. We would need to check that for any trace such that $\sigma, \sigma_d \vDash^d_\infty C$, all elements D of $\sigma_d(i)$ for all i are conflict free. By conflict free we mean that for every D there exists a d, element of D that is not in conflict with the rest of the deontic notions. Thus if

$d = O_a$ then for all D' element of $\sigma_d(i)$, D' does not prohibit the action a ($D' \neq \{F_a\}$) and it does not give permission to perform a mutually exclusive action ($D' \neq \{P_b\}$ and $a\#b$) and if D' is an obligation or a choice of obligations, at least an element of D' must not oblige a mutually exclusive action ($D' \subseteq \{Oa | a \in A\} \rightarrow \exists Ob \in D'$ s.t. $\neg(a\#b)$)

$d = F_a$ then there does not exists d' element of D', element of $\sigma_d(i)$, such that it obliges or permits the action a ($d' = P_a$ or $d' = O_a$).

$d = P_a$ then there does not exist d' element of D', element of $\sigma_d(i)$, such that it prohibits action a or obliges a mutually exclusive action ($d' = F_a$ or $d' = O_b$ and $a\#b$).

Thus we are requiring that every element of the trace σ_d is conflict free. If the elements contain a single deontic notion, then we are required to ensure that it is not in conflict with any of the other deontic notions applying in this position. If it does not contain a single element, that means that we have a choice of obligations and thus, for it to be conflict free there needs to exist at least one of these elements that is not in conflict with any of the others.

Unfortunately, this method will only work for cases when we are permitted and forbidden to perform the same action and when we are permitted and obliged to perform conflicting actions. We need to ensure that the conflict was reached without any prior violations and thus, ensuring that $\sigma \vDash_\infty C$. Since permission will never result in the violation of a trace[3], there will be traces that satisfy the contract but still be identified as having conflicts. Consider the contract $[a]P(b) \wedge [b]F(b)$. The trace $[\{a, b\}, a, \ldots]$ will satisfy this contract and thus will be tested for a conflict, which in this case there is since the corresponding deontic trace is $[\emptyset, \{\{P_b\}, \{F_b\}\}, \emptyset, \ldots]$ and it has a conflict when $i = 1$.

However, for conflicts where obligations and prohibitions are concerned, there will be no trace that satisfies the contract and also leads to these conflicts. This is because once we reach these conflicts there will be no possible way to satisfy the contract and thus, there will be no infinite trace satisfying the contract that leads to the conflict. Because of this, we need to define another semantics. We are required to define the notion of 'has not violated yet'. We shall do this by defining a relation similar to the $\vDash_\infty$ relation but instead of being defined on infinite traces we define it for finite traces. We use the interpretation "has not violated yet" because if we extend

[3] When looking just at a trace we cannot know if a permission is satisfied or not since we are just looking at one possible sequence of action and thus, cannot know if there is the possibility to do the permitted action or not. In fact, in the original trace semantics, permission was not defined whereas in ts2 the permission was added just so that we add the information to σ_d

the finite trace we could end up violating the contract. Furthermore, we do not want to use the notion of having a finite trace that satisfies the contract since that would entail that all the clauses have been satisfied and that is not what we want.

Consider the contract $[a]O(b) \wedge [b]F(b)$. The deontic trace that would identify this conflict is $[\emptyset, \{\{O_b\}, \{F_b\}\}, \emptyset, \ldots]$. However there is no trace of actions that will satisfy the contract that will have this corresponding deontic trace. Satisfying traces, given that the set of actions is $\{a, b\}$ would be $[a, b, \cdots]$ and $[b, a, \cdots]$. Both these traces do not have a deontic trace identifying the conflict since both do not lead to a state of conflict. A trace that would lead to the conflict would start with the action $[\{a, b\}]$. Regardless of what follows, we will not be able to satisfy the contract. If we have the relation described above, we could reason that the trace $[\{a, b\}]$ has not yet violated the contract and so we need to check that it has not led to any deontic conflict and thus we will be able to identify that a conflict has occurred.

The new relation defined is $\vDash_f$ where $\sigma, \sigma_d \vDash_f C$ is interpreted as the finte trace σ has not yet violated the contract C and the finite deontic trace σ_d is consistent with C. We will refer to this semantics as ts3.

To define the new relation we are going to require a number of semantic tools. We define the function *len* as to return the size of a trace. Thus if trace $\sigma = [a, b], c$ then $len(\sigma) = 2$. In the new semantics we want that both σ and σ_d are finite. Furthermore we want that $len(\sigma) = len(\sigma_d)$. Hence, if $len(\sigma) \neq len(\sigma_d)$ then $\sigma, \sigma_d \nvDash_f C$.

One should note that if the trace is empty (i.e. $len(\sigma) = 0$) then it cannot violate any contract. Thus if $len(\sigma) = 0$ then $\sigma, \sigma_d \vDash_f C$ where C is any contract. Otherwise the same semantics hold. Most of the other rules are very similar to ts2. The full finite semantics ts3 is defined as follows.

Definition 5.1.2. *Trace Semantics ts3*

$$
\begin{array}{lll}
\sigma, \sigma_d & \not\models_f & C \textit{ if } len(\sigma) \neq len(\sigma_d) \\
 & & \\
\sigma, \sigma_d & \models_f & \top \textit{ if } len(\sigma) = 0 \textit{ or } \sigma_d(0) = \emptyset \textit{ and } \sigma(1..), \sigma_d(1..) \models_f \top \\
\sigma, \sigma_d & \not\models_f & \bot \\
\sigma, \sigma_d & \models_f & C_1 \wedge C_2 \textit{ if } \sigma, \sigma'_d \models_f C_1 \textit{ and } \sigma, \sigma''_d \models_f C_2 \textit{ and } \sigma_d = \sigma'_d \cup \sigma''_d \\
\sigma, \sigma_d & \models_f & C_1 \oplus C_2 \textit{ if } len(\sigma) = 0 \textit{ or } (\sigma, \sigma_d \models_f C_1 \textit{ and } \sigma, \sigma_d \not\models_f C_2) \textit{ or} \\
 & & \qquad (\sigma, \sigma_d \models_f C_2 \textit{ and } \sigma, \sigma_d \not\models_f C_1) \\
\sigma, \sigma_d & \models_f & [\alpha_\&]C \textit{ if } len(\sigma) = 0 \textit{ or } (\sigma_d(0) = \emptyset \textit{ and} \\
 & & \qquad (\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..), \sigma_d(1..) \models_f C, \textit{ or } \alpha_\& \not\subseteq \sigma(0)))) \\
\sigma, \sigma_d & \models_f & [\beta; \beta']C \textit{ if } \sigma, \sigma_d \models_f [\beta][\beta']C \\
\sigma, \sigma_d & \models_f & [\beta + \beta']C \textit{ if } \sigma, \sigma_d \models_f [\beta]C \wedge [\beta']C \\
\sigma, \sigma_d & \models_f & [\beta^*]C \textit{ if } \sigma, \sigma_d \models_f C \wedge [\beta][\beta^*]C \\
\sigma, \sigma_d & \models_f & O_C(\alpha_\&) \textit{ if } len(\sigma) = 0 \textit{ or } (\sigma_d(0) = O_{\alpha_\&} \textit{ and} \\
 & & ((\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..), \sigma_d(1..) \models_f \top) \textit{ or } \sigma(1..), \sigma_d(1..) \models_f C)) \\
\sigma, \sigma_d & \models_f & O_C(\alpha; \alpha') \textit{ if } \sigma, \sigma_d \models_f O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\sigma, \sigma_d & \models_f & O_C(\alpha + \alpha') \textit{ if } \sigma, \sigma'_d \models_f O_\bot(\alpha) \textit{ or } \sigma, \sigma'_d \models_f O_\bot(\alpha') \textit{ or} \\
 & & (\sigma_d(0) = (\sigma'_d(0) \cup \sigma''_d(0)) \textit{ and } \sigma, \emptyset; \sigma_d(1..) \models_f [\overline{\alpha + \alpha'}]C) \\
\sigma, \sigma_d & \models_f & F_C(\alpha_\&) \textit{ if } len(\sigma) = 0 \textit{ or } (\sigma_d(0) = F_{\alpha_\&} \textit{ and} \\
 & & \qquad ((\alpha_\& \not\subseteq \sigma(0) \textit{ and } \sigma(1..), \sigma_d(1..) \models_f \top) \\
 & & \qquad\; \textit{or } (\alpha_\& \subseteq \sigma(0) \textit{ and } \sigma(1..), \sigma_d(1..) \models_f C))) \\
\sigma, \sigma_d & \models_f & F_C(\alpha; \alpha') \textit{ if } \sigma, \sigma_d \models_f F_\bot(\alpha) \textit{ or } \sigma, \sigma_d \models_f [\alpha]F_C(\alpha') \\
\sigma, \sigma_d & \models_f & F_C(\alpha + \alpha') \textit{ if } \sigma, \sigma_d \models_f F_C(\alpha) \wedge F_C(\alpha') \\
\sigma, \sigma_d & \models_f & [\overline{\alpha_\&}]C \textit{ if } \sigma_d(0) = \emptyset \textit{ and} \\
 & & \qquad ((\alpha_\& \not\subseteq \sigma(0) \textit{ and } \sigma(1..), \sigma_d(1..) \models_f C) \textit{ or } \alpha_\& \subseteq \sigma(0)) \\
\sigma, \sigma_d & \models_f & [\overline{\alpha; \alpha'}]C \textit{ if } \sigma, \sigma_d \models_f [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C] \\
\sigma, \sigma_d & \models_f & [\overline{\alpha + \alpha'}]C \textit{ if } \sigma_d(0) = \emptyset \textit{ and } (\sigma \sigma_d \models_f [\overline{\alpha}]C \textit{ or } \sigma, \sigma_d \models_f [\overline{\alpha'}]C) \\
 & & \\
\sigma, \sigma_d & \models_f & P(\alpha_\&) \textit{ if } len(\sigma) = 0 \textit{ or } (\sigma_d(0) = P_{\alpha_\&} \textit{ and } \sigma(1..), \sigma_d(1..) \models_f \top) \\
\sigma, \sigma_d & \models_f & P(\alpha; \alpha') \textit{ if } \sigma, \sigma_d \models_f P(\alpha) \wedge [\alpha]P(\alpha') \\
\sigma, \sigma_d & \models_f & P(\alpha + \alpha') \textit{ if } \sigma, \sigma_d \models_f P(\alpha) \wedge P(\alpha')
\end{array}
$$

As mentioned before, an empty trace has not violated the contract and thus this is encoded in the definition of $\top$, $\oplus$, $[]$, O, F and P. The rest of the cases are defined using these base cases.

Consider the semantics given for the obligation. The obligation has not yet been violated if the trace is empty or if it has now been satisfied or else if violated, then the reparation is satisfied. Using this definition we can construct the trace stepwise and at each step know if the contract has been violated or not. The same goes for the rest of the definitions.

Using these trace semantics we shall be able to find all the types of conflicts. Lets us consider again the same example as before. The conflict in $[a]O(b) \wedge [b]F(b)$ cannot be found making use of the previous semantics. However, using these semantics we can find the conflict since the trace $[\{a, b\}]$ has not violated the contract yet $([\{a, b\}], [\emptyset] \vDash_f [a]O(b) \wedge [b]F(b))$ and thus we shall check if this trace leads to a conflict. We shall see how this is done in the following section.

After defining the finite trace semantics we shall prove that ts3 is sound and complete with respect to ts2. We shall not do the proof that ts2 is sound and complete. The outline of the proof would be the following. The definitions are exactly the same except for the deontic trace. Thus we can conclude that if $\sigma \vDash C$ then there exists a σ_d such that $\sigma, \sigma_d \vDash^d_\infty$ and that if $\sigma \nvDash C$ then there does not exists any σ_d such that $\sigma, \sigma_d \vDash^d_\infty C$. Also, if $\sigma, \sigma_d \vDash^d_\infty C$ then it also holds that $\sigma \vDash C$ and if for all possible deontic traces, $\sigma, \sigma_d \nvDash^d_\infty C$ holds then $\sigma \nvDash C$.

Correctness of ts3

At the end of the previous section we explain why we did not formally prove the correctness of ts2 with respect to the original semantics and also gave a very brief outline on how this would have been done. In this section we shall depict the proof of correctness of ts3 with respect to ts2. We shall do this by proving that ts3 is sound and complete with respect to ts2. We say that ts3 is sound with respect to ts2 if when a trace satisfies a contract using the semantics of ts2 then it will also satisfy the contract using ts3. We say that ts3 is complete with respect to ts2 if when a trace satisfies a contract using the samantics of ts3 it will also satisfy the contract using the semantics of ts2. The main problem in this proof is that ts2 makes use of infinite traces whilst ts3 is finite and thus the relation is not "satisfies the contract" but rather "has not violated yet".

The first step is to prove that an empty trace will not violate any contract. We define $\sigma_\emptyset$ as being an infinite sequence of $\emptyset$.

Theorem 5.1.1. *An empty trace can never violate a contract:*

$$\forall C, len(\sigma) = len(\sigma_d) = 0 \Rightarrow \sigma, \sigma_d \vDash_f C$$

Proof. We shall prove this by structural induction on the formula. The base cases are the following and result directly from the definition:

$$len(\sigma) = len(\sigma_d) = 0 \Rightarrow \\ \begin{array}{rcc} \sigma, \sigma_d & \vDash_f & \top \\ \sigma, \sigma_d & \vDash_f & [\alpha_\&]C \\ \sigma, \sigma_d & \vDash_f & O_C(\alpha_\&) \\ \sigma, \sigma_d & \vDash_f & F_C(\alpha_\&) \\ \sigma, \sigma_d & \vDash_f & P(\alpha_\&) \\ \sigma, \sigma_d & \vDash_f & C_1 \oplus C_2 \end{array}$$

The rest of the cases are defined using these base cases.

Case $C_1 \wedge C_2$

$$\begin{array}{ll} & len(\sigma) = len(\sigma_d) = 0 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_1 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_2 \\ \Rightarrow & \{\text{conjunction introduction}\} \\ & \sigma, \sigma_d \vDash_f C_1 \text{ and } \sigma, \sigma_d \vDash_f C_2 \\ \Rightarrow & \{\text{definition}\} \\ & \sigma, \sigma_d \vDash_f C_1 \wedge C_2 \end{array}$$

Case $[\beta; \beta']C$

$$\begin{array}{ll} & len(\sigma) = len(\sigma_d) = 0 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_1 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_2 \\ \Rightarrow & \{\text{let } C_1 = [\beta]C_2 \text{ and } C_2 = [\beta']C\} \\ & \sigma, \sigma_d \vDash_f [\beta][\beta']C \\ \Rightarrow & \{\text{definition}\} \\ & \sigma, \sigma_d \vDash_f [\beta; \beta']C \end{array}$$

The rest of the cases are similar. Refer to Appendix A Theorem A.1 for the complete proof. □

Proposition 5.1.2. *Any finite trace using the semantics of ts3 will satisfy the trivial contract* $\top$

$$\forall \sigma \ \sigma, \sigma_\emptyset(0..len(\sigma)) \vDash_f \top$$

Proof. We shall prove this by induction on the length of σ.

Base Case n=0 from Theorem 5.1.1

Inductive Hypothesis n=k

$$\forall\sigma\ \sigma(0..k), \sigma_{\emptyset}(0..k) \vDash_f \top$$

Inductive Case n=k+1

$$\begin{array}{ll} & \sigma(0..k+1), \sigma_d(0..k+1) \vDash_f \top \\ \Rightarrow & \{\text{definition}\} \\ & \sigma(1..k+1), \sigma_d(1..k+1) \vDash_f \top \\ \Rightarrow & \{\text{the definition does not consider the elements in the trace, only the length. Thus the definition cannot differentiate between } \sigma(1..k+1) \text{ and } \sigma(0..k)\} \\ & \sigma(0..k), \sigma_d(0..k) \vDash_f \top \\ \Rightarrow & \{\text{Inductive hypothesis}\} \end{array}$$

□

Proposition 5.1.3. *Any infinite trace using the semantics of ts2 will satisfy the trivial contract* $\top$

$$\forall\sigma\ \sigma, \sigma_{\emptyset} \vDash^d_\infty \top$$

Proof. The definition of $\sigma, \sigma_d \vDash^d_\infty \top$ is recursive where we chop off the first element from σ and σ_d and then check that $\sigma(1..), \sigma_d(1..) \vDash^d_\infty \top$. The only check on the first element is that $\sigma_d(0) = \emptyset$ thus $\sigma(0)$ is free to be any set of possible actions. Furthermore, we know that $\sigma_d(0) = \emptyset$ since by definition, $\sigma_{\emptyset}$ is the infinite sequence of $\emptyset$ and thus, this proposition is true. □

Proposition 5.1.4. *For any infinite trace which satisfies the trivial contract* $\top$*, the deontic trace will be empty, as defined before,* $\sigma_d = \sigma_{\emptyset}$

$$\sigma, \sigma_d \vDash \top \Rightarrow \sigma_d = \sigma_{\emptyset}$$

Proof. The definition of $\sigma, \sigma_d \vDash^d_\infty \top$ is recursive where we chop off the first element from σ and σ_d and then check that $\sigma(1..), \sigma_d(1..) \vDash^d_\infty \top$. By definition we require that $\sigma_d(0) = \emptyset$ and then we keep on recursively checking the rest of the trace recursively, removing the first element and checking that $\sigma_d(0) = \emptyset$ and thus for $\sigma, \sigma_d \vDash^d_\infty \top$, $\sigma_d = \sigma_{\emptyset}$ □

Theorem 5.1.5. *A trace* σ_d *is equal to the combination of* σ'_d *and* σ''_d *(*$\sigma_d = \sigma'_d \cup \sigma''_d$*) iff any subtrace of* σ_d *is equal to the combination of the subtraces of* σ'_d *and* σ''_d *given that they are of the same length.*

$$\sigma_d = \sigma'_d \cup \sigma''_d \Leftrightarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$$

.

Proof. We shall split this proof in two cases, proving the two directions of the implication separately.

Case $\sigma_d = \sigma'_d \cup \sigma''_d \Rightarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$. The definition of $\sigma_d = \sigma'_d \cup \sigma''_d$ is $\sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n)$. We shall use induction on n in order to prove this case.

Base Case n=0

$$\begin{array}{ll} & \sigma_d = \sigma'_d \cup \sigma''_d \\ \Rightarrow & \{\text{definition}\} \\ & \sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n) \\ \Rightarrow & \{\text{n=0, so we chop } \sigma_d \text{ and take only required sequence}\} \\ & \sigma_d(0) = \sigma'_d(0) \cup \sigma''_d(0) \end{array}$$

Inductive Hypothesis n=k

$$\sigma_d(k) = \sigma'_d(0..k) \cup \sigma''_d(0..k)$$

Inductive Case n=k+1

$$\begin{array}{ll} & \sigma_d = \sigma'_d \cup \sigma''_d \\ \Rightarrow & \{\text{definition}\} \\ & \sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n) \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma_d(k) = \sigma'_d(0..k) \cup \sigma''_d(0..k) \\ \Rightarrow & \{\text{definition}\} \\ & \sigma_d(k) = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(k) \cup \sigma''_d(k) \\ \Rightarrow & \{\text{by definition we may add the next element by adding to sequence}\} \\ & \sigma_d(k+1) = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \\ & \qquad \sigma'_d(k) \cup \sigma''_d(k), \sigma'_d(k+1) \cup \sigma''_d(k+1) \\ \Rightarrow & \{\text{definition}\} \\ & \sigma_d(k+1) = \sigma'_d(k+1) \cup \sigma''_d(k+1) \end{array}$$

Case $\sigma_d = \sigma'_d \cup \sigma''_d \Leftarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$

$\sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$ holds for all N thus if $n = len(\sigma_d)$ then $\sigma_d(0..n) = \sigma_d, \sigma'_d(0..n) = \sigma'_d$ and $\sigma''_d(0..n) = \sigma''_d$

□

Lemma 5.1.6. *If the infinite traces σ, σ_d satisfy a contract C, then any finite prefix of these traces will not violate the contract.*

$$\sigma, \sigma_d \vDash_{\infty}^{d} C \Rightarrow \forall n : N, \sigma(0..n), \sigma_d(0..n) \vDash_f C$$

Proof. $\sigma, \sigma_d \vDash_{\infty}^{d} C$ means that the traces σ, σ_d do not violate the contract C. So if the entire infinite trace does not violate the contract then even any finite prefix would not violate the contract, or at least would not have violated the contract yet. This is the definition of the $\vDash_f$ relationship and so we may conclude that for any finite prefix of the infinite traces that satisfy a contract C will not violate the contract.

We shall use structural induction in order to prove this formally. First of all we have the rule that $\sigma, \sigma_d \nvDash_f C$ if $len(\sigma) \neq len(\sigma_d)$. In our case the lengths of σ and σ_d are identical by definition.

Case $\top$

$$\begin{aligned}
& \sigma, \sigma_d \vDash_{\infty}^{d} \top \\
\Rightarrow\ & \{\text{Proposition 5.1.4}\} \\
& \sigma_d = \sigma_\emptyset \\
\Rightarrow\ & \{\sigma_d = \sigma_\emptyset \text{ and Proposition 5.1.2}\} \\
& \forall n : N, \sigma(0..n), \sigma_d(0..n) \vDash_f \top
\end{aligned}$$

Case $C_1 \wedge C_2$

$$\begin{aligned}
& \sigma, \sigma_d \vDash_{\infty}^{d} C_1 \wedge C_2 \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma, \sigma_d' \vDash_{\infty}^{d} C_1 \text{ and } \sigma, \sigma_d'' \vDash_{\infty}^{d} C_2 \text{ and } \sigma_d = \sigma_d' \cup \sigma_d'' \\
\Rightarrow\ & \{\text{inductive hypothesis and Theorem 5.1.5}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n)' \vDash_f C_1 \text{ and } \forall n : N\ \sigma(0..n), \sigma_d(0..n)'' \vDash_f C_2 \\
& \ \text{and } \forall n : N\ \sigma_d(0..n) = \sigma_d(0..n)' \cup \sigma_d(0..n)'' \\
\Rightarrow\ & \{\text{associativity of } \forall \text{ over conjunction}\} \\
& \forall n : N\ (\sigma(0..n), \sigma_d(0..n)' \vDash_f C_1 \text{ and } \sigma(0..n), \sigma_d(0..n)'' \vDash_f C_2 \\
& \ \text{and } \sigma_d(0..n) = \sigma_d(0..n)' \cup \sigma_d(0..n)'') \\
\Rightarrow\ & \{\text{definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f C_1 \wedge C_2
\end{aligned}$$

Case $O_C(\alpha_\&)$

$$\begin{aligned}
& \sigma, \sigma_d \vDash^d_\infty O_C(\alpha_\&) \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma_d(0) = O\alpha \text{ and} \\
& ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma_d(0) = O\alpha \text{ and } (\alpha_\& \subseteq \sigma(0) \text{ and } \forall n : N\ \sigma(1..n), \sigma_d(1..n) \vDash_f \top) \text{ or} \\
& (\forall n : N\ \sigma(1..n), \sigma_d(1..n) \vDash_f C) \\
\Rightarrow\ & \{\text{No free variable n, thus can move } \forall n : N \text{ outside}\} \\
& \forall n : N\ \sigma_d(0) = O\alpha \text{ and } (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..n), \sigma_d(1..n) \vDash_f \top) \text{ or} \\
& (\sigma(1..n), \sigma_d(1..n) \vDash_f C) \\
\Rightarrow\ & \{\text{an empty trace cannot violate a contract(Theorem 5.1.1)}\} \\
& \forall n : N\ len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
& ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash_f \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash_f C) \\
\Rightarrow\ & \{\text{Definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha_\&)
\end{aligned}$$

The rest of the cases are similar. Refer to Appendix A Theorem A.6 for the complete proof. □

Lemma 5.1.7. *If all the finite prefixes of an infinite trace do not violate a contract C then the infinite traces satisfy the contract.*

$$\sigma, \sigma_d \vDash^d_\infty C \Leftarrow \forall n : N \sigma(0..n), \sigma_d(0..n) \vDash_f C$$

Proof. We shall use structural induction to show that if for any n, $n : N\sigma(0..n), \sigma_d(0..n) \vDash_f C$ then for the equivalent infinite trace, $\sigma, \sigma_d \vDash^d_\infty C$.

Case $\top$

$$\begin{aligned}
& \forall n : N, \sigma(0..n), \sigma_d(0..n) \vDash_f \top \\
\Rightarrow\ & \{\text{Proposition 5.1.3}\} \\
& \sigma, \sigma_d \vDash^d_\infty \top
\end{aligned}$$

Case $C_1 \wedge C_2$

$$
\begin{array}{ll}
 & \forall n : N, \sigma(0..n), \sigma_d(0..n) \vDash_f C_1 \wedge C_2 \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N, (\sigma(0..n), \sigma'_d(0..n) \vDash_f C_1 \text{ and } \sigma(0..n), \sigma''_d(0..n) \vDash_f C_2 \text{ and} \\
 & \qquad \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)) \\
\Rightarrow & \{\text{inductive hypothesis and Theorem 5.1.5}\} \\
 & \sigma, \sigma'_d \vDash^d_\infty C_1 \text{ and } \sigma, \sigma''_d \vDash^d_\infty C_2 \text{ and } \sigma_d = \sigma'_d \cup \sigma''_d \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty C_1 \wedge C_2
\end{array}
$$

Case $[\alpha_\&]C$

$$
\begin{array}{ll}
 & \forall n : N \ \sigma(0..n), \sigma_d(0..n) \vDash_f [\alpha_\&]C \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & len(\sigma) = 0 \text{ or } (\sigma_d(0) = \emptyset \text{ and } (\alpha_\& \subseteq \sigma(0) \text{ and} \\
 & \qquad \forall n : N \ \sigma(1..n), \sigma_d(1..n) \vDash_f C, \text{ or } \alpha_\& \nsubseteq \sigma(0)))) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & len(\sigma) = 0 \text{ or } (\sigma_d(0) = \emptyset \text{ and } (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C, \text{ or} \\
 & \qquad \alpha_\& \nsubseteq \sigma(0)))) \\
\Rightarrow & \{\sigma \text{ is infinite so } len(\sigma) \neq 0\} \\
 & \sigma_d(0) = \emptyset \text{ and } (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C, \text{ or } \alpha_\& \nsubseteq \sigma(0))) \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty [\alpha_\&]C
\end{array}
$$

Case $O_C(\alpha_\&)$

$$\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha_\&) \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
 & ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..n), \sigma_d(1..n) \vDash_f \top) \text{ or } \sigma(1..n), \sigma_d(1..n) \vDash_f C) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
 & ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\Rightarrow & \{\sigma \text{ is infinite so } len(\sigma) \neq 0\} \\
 & \sigma_d(0) = O\alpha \text{ and} \\
 & ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty O_C(\alpha_\&)
\end{array}$$

The rest of the cases are similar. Refer to Appendix A Theorem A.7 for the complete proof. □

Lemma 5.1.8. *ts3 is correct with respect to ts2*

$$\sigma, \sigma_d \vDash^d_\infty C \Leftrightarrow \forall n : N \sigma(0..n), \sigma_d(0..n) \vDash_f C$$

Proof. Directly from Lemma 5.1.6 and Lemma 5.1.7. □

Lemma 5.1.8 concludes our proof. For the complete proof please refer to Appendix A.

5.1.3 Canonical form and the Kleene Star

Before we look at the automation of conflict analysis we have to take a look at the canonical form. The canonical form has been only specified for actions defined using the Concurrency, Choice and Sequence operators but no Kleene star. If we look at the trace semantics it assumes that action concurrency is the innermost operator and thus, the concurrency operator will only be acting on atomic actions. However, the syntax allows us to define a formula of the following form $[a\&(b^*)]C$ that cannot be described directly using the semantics. How can we handle such situations?

One should note that the Kleene star poses many of problems when defining normal forms. There is no standard normal form for regular expressions and neither for Propositional Dynamic Logic. One way is by unfolding the Kleene star, but this may lead to non-termination.

Let us consider the simplest case, $a\&(b^*)$. Remember that $b^* = 1 + b \cdot b^*$, so substituting this into the original formula will result in $a\&(1 +$

$b \cdot b^*$) that with a bit of massaging, by taking out the choice and sequence operator (similar to what is done in the canonical form), we end up with $a+(a\&b \cdot b^*)$. We have, thus, successfully taken the Kleene star outside of the concurrancy operator. Remember that the Kleene star may be repetitively opened ending up with an infinite definition. Thus regardless of the length of action with which we are combining it, we can always generate a longer definition. Consider for example $(a \cdot b)\&c^*$, which is equal to $(a \cdot b)\&(1 + c \cdot c^* + c \cdot c \cdot c^*)$ that with a bit of massaging will end up defined as $(a \cdot b) + (a\&c \cdot b) + (a\&c \cdot b\&c \cdot c^*)$.

A slightly more complicated situation arises when combining the Kleene star with the sequence operator. When it is to the right of the sequence operator, there is no problem; however, when it is to the left it is not as straight forward. The semantics suggest the following way to process the Kleene star: $[\beta^*]C = C \wedge [\beta][\beta^*]C$ thus if $C = [a]C'$ then this will end up being $[a]C' \wedge [\beta][\beta^*][a]C'$. Also, remember that $[a \cdot b]C = [a][b]C$. Thus $[a^* \cdot b]C = [b]C \wedge [a \cdot a^* \cdot b]C$. So the action $a^* \cdot b$ should be equal to $b + a \cdot a^* \cdot b$ and not as one might expect to be equal to $1 + a \cdot b\&a^*$. Thus, if we have $a\&(b^* \cdot c)$ this will have to be translated to $(a\&c) + (a\&b \cdot b^* \cdot c)$.

Another question would be how to simplify the concurrency of two Kleene stars, $a^*\&b^*$. This could be opened as $1 + a \cdot a^* + b \cdot b^* + a\&b \cdot a^*\&b^*$. However, this still leaves the concurrency operator between two non-atomic actions. Thus when we add the Kleene star, we cannot always end up with the concurrency operator between atomic actions. This representation is however equivalent to $(a\&b)^*$ which can be processed.

In this work we shall restrict that the syntax will not contain concurrency of Kleene star as seen in the previous paragraph. We believe that these could still be represented using an equivalent representation with the Kleene star outside of the concurrency operator; however, since this was not the focus of our work we opted to restrict the syntax. The definition of the canonical form with the addition of the Kleene star will be part of our future work.

5.2 Conflict Analysis

In this section we shall first define what a conflict is using the extension to the initial $\mathcal{CL}$ trace semantics we defined in the previous section. After defining formally what a contract is we shall define the decision procedure that will automatically analyse a contract for conflicts.

5.2.1 Contract Conflict Freedom

After proving the soundness and completeness of ts3, we need to find a method that will allow us to verify that a contract is conflict free. We require that the trace leading to a conflict has not already violated the contract. If a contract is violated then the contract has been voided and external legal action is to be taken. As discussed before, we are concered with finding conflicts which are being brought about by the contract and thus we require that the contract has not been already violated. Thus, in order to find a conflict we require that given a contract C, $\sigma, \sigma_d \vDash_f C$ and that σ_d has a conflict.

Consider the contract $[a]O(b) \wedge [b]F(b)$. If a is observed, then we are required to satisfy both $O(b)$ and $F(b)$. As described earlier this is one case of a conflict. We also require that $\sigma, \sigma_d \vDash_f C$ and that the conflict is found in σ_d. In this case $[a], [\emptyset] \vDash [a]O(b) \wedge [b]F(b)$ however σ_d has not yet reached the conflict. In the following step, whatever action we perform but, we are not able to satisfy the contract and thus one would expect that $[a, b], [\emptyset, \{O_b, F_b\}] \nvDash [a]O(b) \wedge [b]F(b)$ since by the definition of prohibition, in order to satisfy the prohibition of b we cannot observe b. However, remember that $F(b)$ is shorthand for $F_\perp(b)$ and thus by the definition of prohibition, if it is violated then we are required to satisfy the reparation, in which case it is the unsatisfiable clause $\perp$. Thus $[a, b], [\emptyset, \{O_b, F_b\}] \vDash [a]O(b) \wedge [b]F(b)$, however any extention of these traces will not satisfy the contract since no trace may satisfy the $\perp$ clause.

Definition 5.2.1. *A contract C is said to be conflict free if for all σ and σ_d such that $\sigma, \sigma_d \vDash_f C$ every element D of $\sigma_d(i)$, for $i = 0$ up till $len(\sigma_d)$, is not in conflict with any other deontic notion in the same trace position. Let D' be any element of $\sigma_d(i)$. D is not in conflict if there exists at least one element d of D such that if*

$$\begin{array}{ll} d = O_a & \textit{then } \forall\, D' \ (D' \neq \{F_a\} \textit{ and } (D' \neq \{P_b\} \textit{ and } a\#b) \textit{ and} \\ & \qquad D' \subseteq \{Oa | a \in A\} \rightarrow \exists Ob \in D' \textit{ s.t. } a\not\#b) \\ d = F_a & \textit{then } \nexists\, d' \in D', \textit{ s.t. } d' = P_a \textit{ or } d' = O_a \\ d = P_a & \textit{then } \nexists\, d' \in D', \textit{ s.t. } d' = F_a \textit{ or } (d' = O_b \textit{ and } a\#b) \end{array}$$

Using this definition of conflict, we shall only be able to find Strong conflicts. In order to find weak conflicts we need to treat a set of possibilities as if they all need to be satisfied. Once we have defined what a conflict is, we can turn to defining a method that will automatically check if a contract is conflict free or not.

5.2.2 Automating conflict analysis

We shall split the automated conflict analysis in two steps. First we shall generate an automaton that accepts all and only traces which satisfy a contract. Then, we shall apply conflict analysis on this automaton.

Generating the automaton

Given a clause C we can construct automaton $A(C) = \langle S, A_{\&}, s_0, T, V, l, \delta\rangle$ where S is the set of states, $A_{\&}$ is the set of concurrent actions, s_0 is the initial state, $T = S \times A_{\&} \times S$ is the set of labelled transitions, V is a special violation state, l is a labelling function labelling states with the $\mathcal{CL}$ clause that holds in that state ($l : S \to \mathcal{CL}$) and δ is a labelling function labelling states with the set of deontic notions that hold in that state.

We define a construction function f which given a $\mathcal{CL}$ formula C and an action α will return the clause that needs to hold in the following step. f is defined in Table 5.1. We need to define the binary operator / that given two actions, will chop the second action from the front of the first action. If the prefix of the first action matches the second action we will remove it. This operator is required since $\mathcal{CL}$ formulas do not allow disjunction between Obligations. This operator will allow us to perform what function f is doing but instead of on a $\mathcal{CL}$ clause, on an action.

Definition 5.2.2. *Given that φ is a compound action we define / as*

$$\begin{aligned}\alpha_{\&}/\varphi &= \emptyset \textit{ if } \alpha_{\&} = \varphi, \textit{ otherwise } 0\\ \alpha \cdot \alpha' &= \alpha/\varphi \cdot \alpha' \textit{ if } \alpha/d \neq \emptyset, \textit{ otherwise } \alpha'\\ \alpha + \alpha' &= \alpha/\varphi + \alpha'/\varphi\end{aligned}$$

The construction function is defined in Table 5.2. We initially create a state s_0 where $l(s_0) = C$ and pass that state to function f_c that will generate the rest of the automaton. If the label of the state (i.e. the contract that needs to be satisfied in that state) is equal to 1, that means that the contract has been satisfied and thus, we add a new transition from that state to itself with action being 1 (i.e. for all actions). This is because once a contract is satisfied any action that may occur will still not violate the contract.

If on the other hand the label is 0, then that means that it is the violation state and thus, we make the violation state equal to this state and add a transition to itself with action 1 that will match any action. If a contract is violated no action may satisfy the contract and thus we shall remain in this violation state regardless of what action is observed.

If the label of the state is neither 1 nor 0 we shall go through all possible actions ($\forall a \in A_{\&}$) in order to add all possible transitions out of this state. We first check if there already exists a state (s') in S that is labelled with what would be required to be satisfied after the transition is taken. This is computed by function f (l(s')=f(l(s),a)). If this is the case we add the transition from the current state to this state labelled with a ($T \cup (s, a, s')$). Checking that there is no state already labelled with what is required to be satisfied in the following state will ensure that we have at most only one state representing satisfaction and violation and will also allow termination of the algorithm.

Given that $\varphi \in A_{\&}$

$$
\begin{array}{rcl}
f(1,\varphi) & = & 1 \\
f(0,\varphi) & = & 0 \\
f(C_1 \wedge C_2,\varphi) & = & f(C_1,\varphi) \wedge f(C_2,\varphi) \\
f(C_1 \oplus C_2,\varphi) & = & \begin{cases} 1 & \text{if } (f(C_1,\varphi) = 1 \wedge f(C_2,\varphi) = 0) \vee \\ & (f(C_1,\varphi) = 0 \wedge f(C_2,\varphi) = 1) \\ 0 & \text{if } (f(C_1,\varphi) = 1 \wedge f(C_2,\varphi) = 1) \vee \\ & (f(C_1,\varphi) = 0 \wedge f(C_2,\varphi) = 0) \\ f(C_1,\varphi) \oplus f(C_2,\varphi) & \text{otherwise} \end{cases} \\
f([\alpha_{\&}]C,\varphi) & = & \begin{cases} C & \text{if } \alpha_{\&} \subseteq \varphi \\ 1 & \text{otherwise} \end{cases} \\
f([\beta \cdot \beta']C,\varphi) & = & f([\beta][\beta']C,\varphi) \\
f([\beta + \beta']C,\varphi) & = & f([\beta]C \wedge [\beta']C,\varphi) \\
f([\beta^*]C,\varphi) & = & f(C \wedge [\beta][\beta^*]C,\varphi) \\
f(O_C(\alpha_{\&}),\varphi) & = & \begin{cases} 1 & \text{if } \alpha_{\&} \subseteq \varphi \\ C & \text{otherwise} \end{cases} \\
f(O_C(\alpha \cdot \alpha'),\varphi) & = & f(O_C(\alpha) \wedge [\alpha]O_C(\alpha'),\varphi) \\
f(O_C(\alpha + \alpha'),\varphi) & = & \begin{cases} 1 & \text{if } f(O_0(\alpha),\varphi) = 1 \text{ or } f(O_0(\alpha'),\varphi) = 1 \\ C & \text{if } f(O_0(\alpha),\varphi) = 0 \text{ and } f(O_0(\alpha'),\varphi) = 0 \\ O_C(\alpha + \alpha'/\varphi) & \text{otherwise} \end{cases} \\
f(F_C(\alpha_{\&}),\varphi) & = & \begin{cases} C & \text{if } \alpha_{\&} \subseteq \varphi \\ 1 & \text{otherwise} \end{cases} \\
f(F_C(\alpha \cdot \alpha'),\varphi) & = & f([\alpha]F_C(\alpha'),\varphi) \\
f(F_C(\alpha + \alpha'),\varphi) & = & f(F_C(\alpha) \wedge F_C(\alpha')) \\
f([\overline{\alpha_{\&}}]C,\varphi) & = & \begin{cases} C & \text{if } \alpha_{\&} \nsubseteq \varphi \\ 1 & \text{otherwise} \end{cases} \\
f([\overline{\alpha \cdot \alpha'}]C,\varphi) & = & f([\overline{\alpha}][\overline{\alpha'}]C,\varphi) \\
f([\overline{\alpha + \alpha'}]C,\varphi) & = & \begin{cases} C & \text{if } f([\overline{\alpha}]C,\varphi) = C \text{ or } f([\overline{\alpha'}]C,\varphi) = C \\ 1 & \text{if } f([\overline{\alpha}]C,\varphi) = f([\overline{\alpha'}]C,\varphi) = 1 \\ [\overline{\alpha + \alpha'/\varphi}]C & \text{otherwise} \end{cases}
\end{array}
$$

Table 5.1: Definition of the construction function

```
$f_c(s) =$  if $l(s) = 1$ then
              $T := T \cup (s, 1, s)$
            if $l(s) = 0$ then
              $V := s$
              $T := T \cup (V, 1, V)$
            otherwise
              $\forall a \in A_\&$
                if $\exists\, s' \in S$ s.t. $l(s') = f(l(s), a)$ then
                  $T := T \cup (s, a, s')$
                otherwise
                  new $s'$
                  $l(s') := f(l(s), a)$
                  $S := S \cup s'$
                  $T := T \cup (s, a, s')$
                  $d(s') := f_d(l(s'))$
                  $f_c(s')$
```

Table 5.2: Definition of the Construction function

$$\begin{array}{lcl}
f_d(C_1 \wedge C_2) & = & \{f_d(C_1)\} \cup \{f_d(C_2)\} \\
f_d(O(\alpha_\&)) & = & \{O_{\alpha_\&}\} \\
f_d(F(\alpha_\&)) & = & \{F_{\alpha_\&}\} \\
f_d(P(\alpha_\&)) & = & \{P_{\alpha_\&}\} \\
f_d(O(\alpha + \alpha')) & = & f_d(O(\alpha)) \cup f_d(O(\alpha')) \\
f_d(\text{ otherwise }) & = & \emptyset
\end{array}$$

Table 5.3: Definition of the deontic labelling function f_d

If there does not exist such a state, we shall create this new state that is labelled with what has to be satisfied next ($l(s') = f(l(s), a)$). We add this state to S and add the transition (s, a, s') to T. We label the deontic label by calling the function f_d defined in Table 5.3. This function will label the state with all the deontic notions that apply in this state. One should note that this state has been labelled with a $\mathcal{CL}$ clause that passed through the function f; thus, the clauses would have been opened and the deontic notions would have been isolated and joined only with conjunction (e.g. $F(a + b)$ would be translated to $F(a) \wedge F(b)$). We shall then call the construction function on this new state s'.

Once we create the automaton we check that a contract is conflict free by ensuring that every state generated is conflict free:

Definition 5.2.3. *An automaton $A(C)$ is said to be conflict free if for every state s in S, D which is an element of $f_d(s)$ is not in conflict with any other* *element of $f_d(s)$. Let D' be any element of $f_d(s)$. D is not in conflict if there exists at least one element d of D such that if*

$$\begin{array}{ll}
d = O_a & \textit{then}\ \forall\, D'\ (D' \neq \{F_a\}\ \textit{and}\ (D' \neq \{P_b\}\ \textit{and}\ a\#b)\ \textit{and} \\
 & \qquad D' \subseteq \{Oa | a \in A\} \rightarrow \exists Ob \in D'\ \textit{s.t.}\ a\not\#b) \\
d = F_a & \textit{then}\ \nexists\ d' \in D',\ \textit{s.t.}\ d' = P_a\ \textit{or}\ d' = O_a \\
d = P_a & \textit{then}\ \nexists\ d' \in D',\ \textit{s.t.}\ d' = F_a\ \textit{or}\ (d' = O_b\ \textit{and}\ a\#b)
\end{array}$$

5.2.3 Correctness of Algorithm

In order to prove corectness of the algorithm, we first need to prove that the algorithm will accept all and only the traces which satisfy the contract. Then we shall prove that the algorithm will identify correctly if a contract is conflict free or not.

Lemma 5.2.1. *Given a $\mathcal{CL}$ expression C, we may build an automaton $A(C)$ that will accept all and only the traces σ that satisfies the contract, that is $\sigma \vDash C$. This can be achieved by proving*

$$\sigma \vDash C \Leftrightarrow \sigma(1..) \vDash f(C, \sigma(0))$$

We shall prove this by applying induction on the structure of the formula.

Proof. **Case** $C_1 \wedge C_2$

$$\begin{array}{ll}
 & \sigma \vDash C_1 \wedge C_2 \\
\Leftrightarrow & \{\text{Definition of Trace Semantics}\} \\
 & \sigma \vDash C_1 \text{ and } \sigma \vDash C_2 \\
\Leftrightarrow & \{\text{Inductive hypothesis}\} \\
 & \sigma(1..) \vDash f(C_1, \sigma(0)) \text{ and } \sigma(1..) \vDash f(C_2, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of trace semantics } (\wedge)\} \\
 & \sigma(1..) \vDash f(C_1, \sigma(0)) \wedge f(C_2, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of f}\} \\
 & \sigma(1..) \vDash f(C_1 \wedge C_2, \sigma(0))
\end{array}$$

Case $[\alpha_{\&}]C$

$$
\begin{aligned}
& \sigma \vDash [\alpha_{\&}]C \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \alpha_{\&} \nsubseteq \sigma(0) \text{ or } \sigma(1..) \vDash C \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \alpha_{\&} \nsubseteq \sigma(0) \text{ or } \sigma(2..) \vDash f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of trace semantics } ([\,])\} \\
& \sigma(1..) \vDash [\alpha_{\&}]f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of } [\,]\} \\
& \sigma(1..) \vDash (\alpha_{\&} \nsubseteq \sigma(0) \text{ or } f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f([\alpha_{\&}]C, \sigma(0))
\end{aligned}
$$

Case $O_C(\alpha_{\&})$

$$
\begin{aligned}
& \sigma \vDash O_C(\alpha_{\&}) \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \alpha_{\&} \subseteq \sigma(0) \text{ or } \sigma(1..) \vDash C \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \alpha_{\&} \subseteq \sigma(0) \text{ or } \sigma(2..) \vDash f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of Obligation}\} \\
& \sigma(1..) \vDash (O_{f(C,\sigma(1))}(\alpha_{\&})) \\
\Leftrightarrow\ & \{\text{Opening of Obligation}\} \\
& \sigma(1..) \vDash (\alpha_{\&} \subseteq \sigma(0) \text{ or } f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(O_C(\alpha_{\&}), \sigma(0))
\end{aligned}
$$

The rest of the cases are similar. Refer to Appendix B Theorem B.1 for the complete proof. □

After proving that the automaton $A(C)$ accepts all and only runs that satisfy the contract we need to prove that the automaton identifies that a contract is conflict free if and only if C is actually conflict free. An accepted run of the automaton $A(C)$ is a sequence of states r such that for $i =$ 0 to $len(r)$ $r_i \neq V$. With this definition of a run we accept all traces which have either satisfied the contract or have not yet violated the contract. Furthermore, we do not require to check that every state in the run is not the violating state since by construction we cannot leave the violating state and thus, we need only to check the final state.

From the definition, $A(C)$ is conflict free if for every run r that is accepted by $A(C)$, every element D of $\delta(r_i)$, for $i = 0$ up till $len(r)$, is not in conflict with any other deontic notion in the same state. Let D' be any element of $\delta(r_i)$. D is not in conflict if there exists at least one element d of D such that if

$$\begin{array}{ll} d = O_a & \text{then } \forall\, D' \; (D' \neq \{F_a\} \text{ and } (D' \neq \{P_b\} \text{ and } a\#b) \text{ and} \\ & \qquad\quad D' \subseteq \{Oa | a \in A\} \rightarrow \exists Ob \in D' \text{ s.t. } a\not\#b) \\ d = F_a & \text{then } \nexists\, d' \in D',\ \text{s.t. } d' = P_a \text{ or } d' = O_a \\ d = P_a & \text{then } \nexists\, d' \in D',\ \text{s.t. } d' = F_a \text{ or } (d' = O_b \text{ and } a\#b) \end{array}$$

This is equivalent to the definition with $ts3$. Also, the labelling function f_d is equivalent to how the labelling is done in ts3. However, our algorithm checks that all the states do not have a conflict rather than checking all possible satisfying runs. In order to prove that this is correct we shall need to prove that all and only reachable states are generated.

Proposition 5.2.2. *All and only reachable states are generated by the construction function f_c*

Proof. By Lemma 5.2.1 we conclude that the construction will construct all required reachable states. Furthermore, from the construction, only reachable states are generated since the states are generated recursively by creating the state and a transition to that state from an already reachable state. □

Lemma 5.2.3. *A contract is identified to be conflict free by the automaton iff it is actually conflict free*

Proof. By Lemma 5.2.1 the automaton can accept all satisfying traces and thus, by ensuring that no run has a conflicting state is equivalent to ensuring that every deontic trace does not have any conflicting labels. However our algorithm does not check every trace but goes through all the states. But by Proposition 5.2.2 we can conclude that it is equivalent since all the states created are reachable and thus, all the states will be passed through by at least one run. □

5.3 Related Work

When it comes to conflict analysis, no work was found that does this automatically as in this work. However, in [34] we find a method for generating an automaton from the trace semantics aimed at monitoring. The automaton generated is different from the one generated in this work and cannot be used for conflict analysis. Furthermore, as we shall see in Chapter 6 we may create a monitor directly from the automaton generated in this work.

In [48], a Labelled Transitions System (LTS) is generated in an ad hoc manual manner in order to be model checked with LTL formulas. We may translate the automaton generated from a $\mathcal{CL}$ contract into an LTS automatically. Thus we automate the generation of an LTS from a $\mathcal{CL}$ contract thus the method specified in [48] all automated. However, in [48] the full branching semantics is used and by using this method, we may not translate automatically the branching semantics. In this case, the branching portions of the formula would have to be done manually.

5.4 Conclusion

This chapter was split into two parts. In the first part we showed why the trace semantics presented in [34] were not enough for determining that a contract is conflict free. We then proceeded with the extension of this semantics, first to ts2 and then to ts3 once we had shown why ts2 was still not adequate.

The second part of the chapter started with a formal definition of what a conflict free contract is. This was defined making use of the semantics specified in the previous section. We then defined a function which would create an automaton that accepts all on only traces which satisfy the contract and defined a method to ensure that a contract is conflict free on this automaton. We finally proved that the automaton does in fact accept all and only non violating traces and that the conflict analysis is correct.

In the next chapter we shall show how by using the automaton generated in this chapter we may make further analysis on the contract. Thus, we not only are able to perform conflict analysis on a contract but by analysing the automaton we can verify other properties on the contract.

Chapter 6

Other Analysis

Once we have created the automaton in Chapter 5 in order to find conflicts, we can make use of the same automaton to perform other analysis on the contract. Remember that we are essentially generating an automaton that accepts all possible traces that do not violate the contract. We also add certain information to the automaton that will enable us to perform conflict analysis. All this can be used in order to perform other types of analysis.

After the work taken to generate the automaton, it is logical to try and apply as many different analysis as possible. As mentioned in previous chapters, analysing for conflict analysis could be seen as model checking the contract for a particular property; thus, now we are adding more diverse properties we can model check for.

6.1 Simulation

Suppose that we are drafting a contract. We would like to check that the contract behaves as expected in certain situations. We could manually *execute* the contract. This will ensure that the behaviour of the contract is as expected in these limited number of situations. Thus, this will help in gaining confidence that the behaviour of a system satisfying the contract is as expected.

However, when doing this manually, one could easily make mistakes, or overlook certain clauses; thus, this will result in possibly overlooking mistakes the contract might have that could then be abused by certain signatories. However, we can automate the simulation process leaving the choice of which trace to follow to the user while also ensuring that the contract is being satisfied.

Once we have generated the automaton from the contract, simulation will boil down to the execution of the automaton. If we are given the contract in automaton form, simulation will constitute in moving from one state to another depending on the desired actions. This simulation could be done

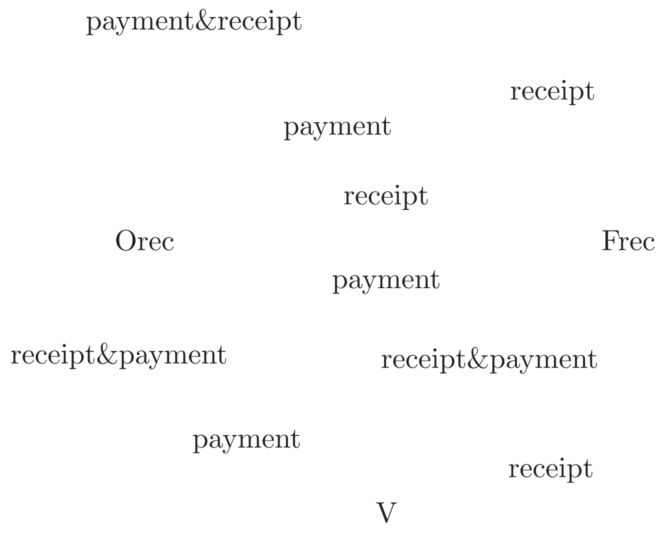


Figure 6.1: Automaton generated from $[1^*][\mathit{payment}]O(\mathit{receipt}) \wedge [1^*][\overline{\mathit{payment}}]F(\mathit{receipt})$

manually directly from the automaton; however, we could aid the user even more if we provide a simpler interface to the automaton. The interface could be similar to the UPPAAL [3] interface that is used when simulating the model.

Consider the simple clause 'You are obliged to issue a receipt on payment, but you are forbidden to issue a receipt before payment is made'. This could be represented in $\mathcal{CL}$ as $[1^*][\mathit{payment}]O(\mathit{receipt}) \wedge [1^*][\overline{\mathit{payment}}]F(\mathit{receipt})$. The automaton that would be generated from this clause may be as seen in Figure 6.1. The simulation tool would provide the user with the actions that could be performed starting from the first state. Thus if the user chooses the payment action, he will advance a step in the automaton, and the interface will instruct the user that he cannot accept another payment since that would lead to the violating state and thus, he needs to issue a receipt. This is exactly the behaviour that the user is expecting in this situation. However, the user might also notice that in the first step, the simulator is allowing the user to issue a receipt. This is not the desirable behaviour since the issueing of a receipt should only be allowed after payment has been received. This would identify the problem in the contract, where we would need to add the close which forbids the issue of a receipt in the first step.

Even though this is quite a simple analysis, it would aid during the drafting and negotiation of contracts. Using simulation we can ensure that a certain sequence of actions would violate a contract or what obligations will the contract enforce after a certain sequence of actions. This clearly is very helpful during the design and drafting of the contract. It would also be a helpful tool to help the signatories of the contract understand the contract and be more confident that they are signing a contract that ensures the desired behaviour.

6.2 Monitoring

Monitoring is a technique of observing a live system ensuring that certain behaviour is adhered to. For example, one could be monitoring internet traffic in order to identify when a Denial of Service attack is being performed in order to attempt in counter acting the attack. Monitoring of $\mathcal{CL}$ contracts could be seen as being quite similar to simulation. The difference is that instead of the user entering the actions we are observing a live system. Monitoring is a very desirable option once a contract is represented using $\mathcal{CL}$. It would also be more desirable to have an automated manner to construct the monitor that will make sure that the contract is abided to. Such a monitor would ensure that if any obligations are broken then the reparations are enacted and if not, detect this violation and inform the user.

Since we have the automaton constructed already, what we need is to make a means to observe the actions from the real-time application. This may be done by making use of a monitoring framework. Furthermore, this can be seen as real-time verification since we are verifying that the parties are not violating the contract while it is enacted.

In our implementation, we provided monitoring by translating the automaton into Larva [15] code. Larva is a real-time verification system. This will allow us to write contracts restricting Java programs from which we automatically generate a monitor. Larva is a framework used to generate code seamlessly in order to monitor an application written in Java. Using Larva we can specify the events that are to be monitored. Then, using the Larva compiler, the Larva model will be translated automatically into the required Java code. The events will be in the form of method calls, where when specific methods are called, the event will be triggered which will correspond in the monitor automaton moving to the following state. Larva makes use of AspectJ [30] in order to observe the events; thus. you could easily monitor certain specific method calls.

The Larva code will piggy back onto the application, and you specify an automaton or model which will do the actual monitoring. The monitor will start in its initial state and move from one state to another depending on what events are observed. If the monitor ends in a bad state, it is then up to the user to decide what action he should take. From $\mathcal{CL}$ we will be generating the Larva model automatically.

The definition of a Larva model is split into two parts. We have the

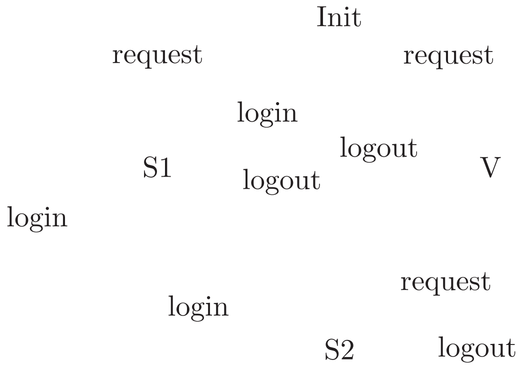


Figure 6.2: Automaton generated from the $\mathcal{CL}$ clause $[\overline{login}^*]F(request) \wedge [1^*][logout][\overline{login}^*]F(request) \wedge F(request)$ using our implementation. One should note that this is not the minimal automaton possible. We require to perform formula simplification in order to have the formulas required to be satisfied in states 1 and 3 to be identical and thus be able to join them together

specification of the events where we tie the events to particular method calls. The events may also have conditions attached to them and also variables. The second part is the specification of the monitor automaton which is made up of states and transitions. Transitions in Larva not only have events which are observed but also conditions which need to be met in order to take the transition and also actions which are performed when taking a transition.

From $\mathcal{CL}$ we automatically generate the specification of the monitor automaton. One would then have to specify the events by tying the events to particular methods of the implementation. One should note that Larva does not support concurrent actions. We will be monitoring a real implemented system and thus due to practical limitations we cannot really observe a concurrent call to different methods. Because of this we should make all actions mutually exclusive in order to have transitions with single actions. We may still represent concurrent actions in $\mathcal{CL}$ however we would eventually still need to tie particular concurrent actions to a single method call or timer event.

Consider that we want to ensure that a user logs into a system before being able to request data and also we allow the user to log out. This would be written in $\mathcal{CL}$ as:

$$[\overline{login}^*]F(request) \wedge [1^*][logout][\overline{login}^*]F(request) \wedge F(request)$$

The automaton generated would be as seen in Figure 6.2. From this automaton we generate the Larva code of Listing 6.1.

```
PROPERTY clcontract {
    STATES {
        BAD { V }
    NORMAL { S1 S2 }
    STARTING { Init }
    }
    TRANSITIONS {
        Init -> S1 [login]
        Init -> V [request]
        Init -> S2 [logout]
        S1 -> S1 [login]
        S1 -> S1 [request]
        S1 -> S2 [logout]
        S2 -> S2 [logout]
        S2 -> V [request]
        S2 -> S1 [login]
    }
}
```

Listing 6.1: This code is generated automatically from the automaton in Figure 6.2 which is generated from a $\mathcal{CL}$ clause

The step which would be left is tying the actions to method calls. This will need to be done manually since knowledge of the system is required. Lets assume that in order to login into the system a method named “login” is called on some object, in order to logout of the system a method named “logout” would be called on some object whereas the method “requestItem” is called in order to request information. We specify the events using the code in Listing 6.2. Using aspect programming the monitoring framework will be notified when the methods login, logout or requeast are called on any object.

```
EVENTS{
    login = {*.login()}
    logout= {*.logout()}
    request= {*.request()}
}
```

Listing 6.2: We specify the events in order to tie the $\mathcal{CL}$ actions with actual method calls.

Thus in this example, the monitor will start in the initial state. Once a method ‘login’ is called on any object, the monitor automaton will move from state *Init* to state $S1$. If this is followed by a call to a ‘logout’ method followed by a ‘request’ method, the monitor will end up in state V which is a bad state and it will be up to the user to implement what code to be executed by the monitor on the behalf of the application. It is important to note that when implementing the system, one would not need to have any knowledge that a monitor is going to be ensuring properties on the execution og the system thus the development of the application will not incure any penalties. Furthermore, the monitor can be seen as an extra precaution which could eventually also be seemlessly removed if deemed not necessary.

Another ideal place for a monitor is as a module inside Enterprise Service Bus as initially discussed in the Preface. This would directly apply in the world of service oriented architecture. We have not implemented such a module yet however we will discuss this in the case study in Chapter 8.

6.3 Contract Queries

It would be an attractive feature to have the ability to ask/query the contract for certain information. Simulation can be seen as a form of a simple query. Given that the contract is in a certain state and that a certain action has been performed, in what state will we be then? This, even though practical, is quite limited, and using the automaton we have generated we can obtain more information.

Consider we have a contract in place and we know that in certain situations we would be obliged to pay a fine as some form of reparation. It would be very useful to check which are the possible traces that would lead to being obliged to pay this fine. Given that we have the automaton already generated, we can obtain an automaton that will generate all the traces that would lead to the contract obliging the signatories to pay the fine. This can be done by first finding all the states that oblige the payment of the fine, and go through the automaton in reverse order until the start state is reached.

The query could be made starting from a different state rather than the start state, and ask for example how, given that we have already performed a number of actions, shall we satisfy the contract, and this will return all the possible traces that would lead in the contract being satisfied. Thus querying the contract is helpful both during the initial drafting of the contract but also while the contract is already in effect.

Lets take as an example a betting company which is requesting a software house to develop an online betting solution to the following specification:

1. 24 hours before the match, the link for betting on the match should be made available on the main page.

2. After the link is made available, every bet request should be accepted and processed.

3. 1 hour before the start of the match, the link for the betting on the match should be removed.

This would be translated in $\mathcal{CL}$ as:

1. $[24hoursbefore]O(placeLink)$

2. $[placeLink][1^*][betRequest]O(processBet)$

3. $[1hourbefore]O(removeLink)$

This contract would then translated into an automaton and then we may query the contract. A particular query which one would make is to list all traces which lead in the application being obliged to process a bet. This would result in traces of the form $\{24hoursbeforegame \cdot placeLink \cdot betRequest\}$ which are correct, however there are others which suggest an undesirable behaviour. Another trace returned would be $\{placeLink \cdot betRequest\}$ which shows an undesirable situation, where the link is being placed earlier than 24 hours before the match. This could be fixed by adding $[\overline{24hoursbefore}^*]F(placeLink) \wedge F(placeLink)$. After this adjustment we apply the same query. From the traces returned we see that the previous problem has been solved, however we see traces of the form $\{24hoursbeforegame \cdot placeLink \cdot 1hourbefore \cdot removeLink \cdot betRequest\}$. This is also an undesired behaviour. Even though the link has been removed from the page, somebody might still have the link visible on the web page and click on it to place a bet less than one hour before the match. The contract needs to also enforce that once the link is removed, we are forbidden to process bets. To solve this problem we need to change the second clause to:

$$\begin{aligned}[placeLink] \quad & ([\overline{removeLink}^*][betRequest]O(processBet) \wedge \\ & [removeLink][1^*]F(processBet))\end{aligned}$$

One should note that using only quering will not ensure that a contract is correct. In large contracts, the traces returned could be quite long and numerous to analyse manually. An even better solution would be model checking contracts, however up till now, this has not been implemented for $\mathcal{CL}$. However, this example shows that it is a viable help during the drafting of a contract.

6.4 Reachability Analysis

A desirable property that a contract should have is that all the obligations, prohibitions and permissions that are explicitly specified in the contract can actually be enacted. Consider the contract $F(a) \wedge [a]O(b)$. The only way to ever be obliged to perform b is by first observing a, however this would break the prohibition of performing a and since it has no reparation the contract is violated. Thus we could never be obliged to perform b since the contract would have already been violated.

Such a situation is not desirable and would typically mean that there is a mistake in the contract. Thus it would be desirable to ensure that all the obligations, prohibitions and permissions in the contract are reachable without violating the contract, if not, either that normative requirement is not needed, or we have a mistake in the other deontic notions preventing we to reach the situation where the norm is enacted.

We could ensure that this property holds by making use of the automaton generated. Since we only generate the states that are reachable without violating the contract and we are labelling the states with the deontic notions that apply in that particular state, we could ensure this property by listing these norms and go through the automaton ensuring that there exists at least one state labelled with each of the norms. The automaton generated for $F(a) \wedge [a]O(b)$ in Figure 6.3 will identify that the obligation to perform b is never reached since there is no state labelled Ob.

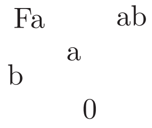


Figure 6.3: The automaton generated for $F(a) \wedge [a]O(b)$ has no state labelled Ob. Thus not all the normative notions required by the contract are reachable

At first glance one might think that this type of analysis would be enough. However, consider the contract $F(a) \wedge [a]O(b) \wedge [b]O(b)$. This would generate the automaton in Figure 6.4. If we use the method as described before, the list of deontic notions in the contract would be $F(a)$ and $O(b)$ and thus, the algorithm will search the automaton that there exists at least a state that has the labels Fa and Ob. Since there does exist a state labelled with Fa and also Ob, the algorithm will erroneously say that all the deontic notions are reachable missing that the obligation to perform b after observing a is not reachable.

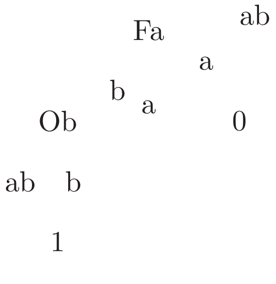


Figure 6.4: Using the automaton generated for $F(a) \wedge [a]O(b) \wedge [b]O(b)$ as is cannot be used to identify that the obligation that should be enacted after observing a is not reachable

In order to solve this problem, we shall need to make some minor changes to how we construct the automaton. We need to identify between the obligation to perform b after observing a b and the obligation to perform b after observing an a. This could be done by adding an index to the normative notions that are identical in order to be able to distinguish between them. Thus we would now have $F(a)$, $O(b)_1$ and $O(b)_2$ that will be labelled accordingly and now using the automaton we can find that there is no reachable state that is labelled with Ob_1.

6.5 Superfluous Clauses

The first question to answer is what is meant by superfluous clauses. Consider the contract $[a]O(b) \wedge [a\&b]O(b)$. It should be clear that the clause $[a\&b]O(b)$ is not required since $[a]O(b) \rightarrow [a\&b]O(b)$. This is not desirable since a contract with such clauses is more complicated than necessary and is surely not minimal. Furthermore, such situations may cause confusion when interpreted or might also be a mistake. Consider the contract "If the luggage weighs more than 15kg then the client is obliged to pay extra. If the client has a priority boarding and his luggage weighs more than 15kg then then client is obliged to pay extra". The equivalent contract would be $[> 15]O(pay) \wedge [priority\& > 15]O(pay)$ which would similarly have the second part of the contract redundant. The problem with this is that it might confuse readers. The readers will try and understand why there is this redundancy and start to wonder if they understood well or not since they would expect that there are no repetitions in the contract or whether if with priority boarding you have a different price list.

Such an analysis could also aid in finding a mistake during the drafting since this redundancy could have been a mistake. Even though these examples were quite simple and easy to spot, in a real situation the contract would be more complicated and thus it would be ideal if we could perform this check in an automated way.

This could be achieved using the automaton generated automatically from the clause using the extension we discussed in the previous section. This mainly entails that we index the deontic notions in order to be able to distinguish between the different parts of the contract that have the identical deontic requirements.

Consider the initial contract we were using as an example, $[a]O(b) \wedge [a\&b]O(b)$. This would generate the automaton in Figure 6.5. We should note that typically the states labelled Ob_1 and Ob_{12} would typically be grouped as one and the information encoded. We split them up in this example in order to simplify the explanation.

In order to find superfluous clauses we need to go through the states. If all the states that are labelled by a certain label are also labelled with

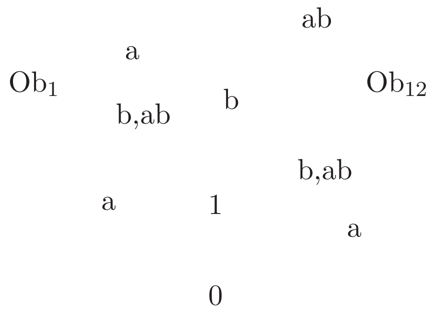


Figure 6.5: Automaton generate from $[a]O(b) \wedge [a\&b]O(b)$.

another label that has the same semantic meaning but with a different index then that clause is redundant. Considering this example. All the states that are labelled with Ob_2 are also labelled Ob_1; however, not the converse. Thus now we know that Ob_2 is contained in Ob_1. We could now notify the user and leave it up to the user to decide between fixing it manually, or automatically remove all the sub clauses that contains the second obligation.

Furthermore, we can also check that the change has preserved the overall behaviour, that by removing the superfluous clause, the same behaviour is still kept. This can be done by comparing the new automaton generated and it should be identical to the original one without the presence of the superfluous clause label.

6.6 Overlapping Clauses

Superfluous clauses may be seen as a special case of overlapping clauses. Overlapping clauses are clauses that enforce the same action at the same time. However, unlike superfluous clauses, one clause does not necessarily contain the other, so unlike superfluous clauses, we cannot simply remove the contained clause. An example of an overlapping clause would be $[a]O(b) \wedge [b]O(b)$. The presence of overlapping clauses does not necessarily mean that the contract should be fixed; however, in certain circumstances it would make the contract easier to read, for example $[a]O(b) \wedge [b]O(b)$ may be translated into $[a + b]O(b)$. Although this has the same meaning it is much easier to read.

Figure 6.6 shows the automaton constructed for $[a]O(b) \wedge [b]O(b)$. Please note that typically, the nodes bounded by the dotted line would be joined in a single node since they have the exact same formulas yet to be processed. The information about the overlapping would be encoded inside the node. It is clear from the image where the overlapping is occurring since it is labelled

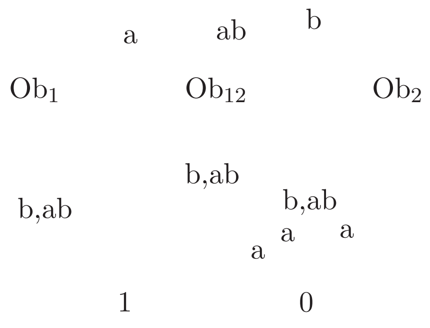


Figure 6.6: Automaton constructed for $[a]O(b) \wedge [b]O(b)$.

with the obligation from two different parts of the contract.

This tool would again aid during the drafting of the contract. Consider the Contract “If one has a business class ticket we are permitted to visit the VIP lounge. If one has a gold card you are permitted to visit the VIP lounge”. In this case we have an overlap if the traveller has both a business class ticket and a gold card. This shows that this case of overlap cannot be described as wrong (unlike the special case of overlap where one clause is contained in another), however, the contract can be made smaller and possibly more clear by changing it to “If one has a business class ticket or a gold card, you are permitted to visit the VIP lounge”.

We should note that a superfluous clause is a special case of overlapping clauses where one clause overlaps entirely another clause. We make this distinction between superfluous and overlapping clauses because typically superfluous clauses identify a mistake in the contract whereas overlapping clauses identify possibilities to improve the representation of the contract but most not an error in contract.

6.7 Related Work

The trace semantics used in order to create the conflict analysis was obtained from [34] in which they proved that it is possible to translate a $\mathcal{CL}$ clause into an automaton. However, they have not implemented the solution and they focused on monitoring instead of conflicts. The theory they used in order to generate the automaton, even though fundemantally similar to eachother is taken from a different point of view. Furthermore, they do not identify any other forms of analysis apart from the generation of the monitor.

6.8 Conclusion

In this chapter we saw a number of other tools and analysis we can obtain once we generate the automaton from the contract. The first three sections describe techniques where we make use of the automaton to get further information from the contract about its behaviour. This information could be used by humans in order to simulate the contract or query for certain properties or by machine in order to automatically monitor the contracts.

The last three examples are tools to identify certain undesirable properties that will help the user to simplify and clarify the contract. It is important to note that this simplification is being automatically done when we translate the contract into automaton form. In the reachability analysis, situations that can never be reached are removed from the automaton. In superfluous clauses and overlapping clauses analysis the states are joined together automatically so that no extra states are produced. Thus, these three simplifications would have been gained automatically if we could translate the automaton back into contractual form. So we could see that the automaton is actually the canonical form of the contract and thus, if we could generate the clauses from the automaton we would effectively be simplifying the contract. We are not claiming that we would get the minimal contract possible, but as could be seen in the previous sections, we would simplify certain situations. However, this analysis was not part of the study, even though we believe that it is possible.

Chapter 7

$\mathcal{CL}$ and Other Logics

From the semantics given in [55] we know that $\mathcal{CL}$ can be represented using an extension of the μ-calculus called Cμ. However even though this helps us with defining the semantics it still poses a difficulty when it comes to representing and understanding these semantics. A useful question to answer is "Is there any other logic that could be used to reprsesent $\mathcal{CL}$?". If so, we might even be able to verify other properties or make use of tools that have already been implemented for these logics.

As seen in the Chapter 2, μ-calculus can be referred to as a temporal logic. Furthermore, there are less expressive temporal logics like CTL* and LTL. So a question that we might ask is what subset of $\mathcal{CL}$ can these logics represent since if the properties we would like to describe fall into these subsets then we can translate the $\mathcal{CL}$ clauses into these logics and then leverage on the tools already present for these logics. For example we could make use of LTL model checking if we are able to translate the $\mathcal{CL}$ clauses into LTL. First lets start by looking at CTL* being more expressive than LTL.

7.1 The Relationship between $\mathcal{CL}$ and CTL*

Using CTL* we believe that we can represent the trace semantics of $\mathcal{CL}$. Furthermore, we can also represent the branching part of the full semantics of $\mathcal{CL}$. The following is a translation function from $\mathcal{CL}$ to CTL*.

$$
\begin{array}{rcl}
f(\bot) & \rightarrow & false \\
f(\top) & \rightarrow & true \\
f(\alpha_{\&}) & \rightarrow & \bigwedge \alpha_p \\
f(\overline{\alpha}_{\&}) & \rightarrow & \neg \bigwedge \alpha_p \wedge \bigvee Prep \\
f(C_1 \wedge C_2) & \rightarrow & f(C_1) \wedge f(C_2) \\
f(C_1 \vee C_2) & \rightarrow & f(C_1) \vee f(C_2) \\
f(C_1 \oplus C_2) & \rightarrow & (f(C_1) \wedge \neg f(C_2)) \vee (f(C_2) \wedge \neg f(C_1)) \\
f([\alpha_{\&}]C) & \rightarrow & (f(\alpha_{\&}) \wedge \mathbf{X} f(C)) \vee f(\overline{\alpha}_{\&}) \\
f([\beta \cdot \beta']C) & \rightarrow & f([\beta] f([\beta']C)) \\
f([\beta + \beta']C) & \rightarrow & f([\beta]C) \wedge f([\beta']C) \\
f([\alpha^*]C) & \rightarrow & f(C) \mathbf{U} \neg f(\alpha) \\
f(\langle \alpha_{\&} \rangle C) & \rightarrow & \mathbf{EX}(f(\alpha_{\&})) \wedge \mathbf{X} f([\alpha_{\&}]C)) \\
f(\langle \beta \cdot \beta' \rangle C) & \rightarrow & f(\langle \beta \rangle f(\langle \beta' \rangle C)) \\
f(\langle \beta + \beta' \rangle C) & \rightarrow & f(\langle \beta \rangle C) \vee f(\langle \beta' \rangle C)
\end{array}
$$

$$
\begin{array}{rcl}
f(O_C(\alpha_{\&})) & \rightarrow & \mathbf{X}(f(\alpha_{\&}) \vee (\neg(f(\alpha_{\&})) \wedge f(C))) \\
f(O_C(\alpha \cdot \alpha')) & \rightarrow & f(O_C(\alpha)) \wedge f([\alpha] O_C(\alpha')) \\
f(O_C(\alpha + \alpha')) & \rightarrow & f(O_{\bot}(\alpha)) \vee f(O_{\bot}(\alpha')) \vee (\neg(f(\alpha) \vee f(\alpha')) \wedge f(C)) \\
f(F_C(\alpha_{\&})) & \rightarrow & \mathbf{X}(\neg f(\alpha_{\&}) \vee (f(\alpha_{\&}) \wedge f(C))) \\
f(F_C(\alpha \cdot \alpha')) & \rightarrow & f(F_{\bot}(\alpha)) \vee f([\alpha] F_C(\alpha')) \\
f(F_C(\alpha + \alpha')) & \rightarrow & f(F_C(\alpha)) \wedge f(F_C(\alpha')) \\
f(P(\alpha_{\&})) & \rightarrow & \mathbf{EX} f(\alpha_{\&}) \\
f(P(\alpha \cdot \alpha')) & \rightarrow & f(P(\alpha)) \wedge f([\alpha] P(\alpha')) \\
f(P(\alpha + \alpha')) & \rightarrow & f(P(\alpha)) \wedge f(P(\alpha'))
\end{array}
$$

As one can see we can also define Permission and also $\langle\rangle$. The syntax of $\mathcal{CL}$ has been changed since the first publication where the semantics in μ-Calculus was given and no new full semantics has been given. If we take the trace semantics together with the original definition of Permission and $\langle\rangle$ we can show that they accept the same trees of actions. One should note however, that in the original branching definition the deontic notions were placed after the action and not before. This was initially done in order to have a deterministic model; however, we moved the deontic notions before the action that will satisfy/violate the notion. This occurs when we created the automaton while still keeping the model deterministic.

7.2 The Relationship between $\mathcal{CL}$ and LTL

Let us first start by defining the modal operators. Obligation and prohibition are quite straight forward. if $O(x)$ then x must hold whilst if $F(x)$ then $\neg x$ must hold. Permission is slightly more tricky, but we can see permission as the negation of prohibition, so $P(x)$ can be defined as $\neg F(x)$ but $F(x)$ is $\neg x$ so $P(x)$ is $\neg\neg x$ which is x. Unfortunately here we see that now $O(x)$ and $P(x)$ are the same thing, but as discussed earlier, if one has the permission to do x it does not necessarily mean that he is also obliged to do x.

Permission adds some form of branching notion. If we have permission to do action x in a certain state we want to have at least one path that does x but we do not need that every path does x like an obligation would demand. LTL cannot be used to describe such notions and thus LTL cannot be used to describe $\mathcal{CL}$ since it is not able to describe branching. However, we believe that the trace semantics of $\mathcal{CL}$ can be represented using LTL. The following is a function that will translate the trace semantics of $\mathcal{CL}$ into LTL.

$$\begin{array}{rcl}
f(\bot) & \rightarrow & false \\
f(\top) & \rightarrow & true \\
f(\alpha_{\&}) & \rightarrow & \bigwedge \alpha_p \\
f(\overline{\alpha}_{\&}) & \rightarrow & \neg \bigwedge \alpha_p \wedge \bigvee Prep \\
f(C_1 \wedge C_2) & \rightarrow & f(C_1) \wedge f(C_2) \\
f(C_1 \vee C_2) & \rightarrow & f(C_1) \vee f(C_2) \\
f(C_1 \oplus C_2) & \rightarrow & (f(C_1) \wedge \neg f(C_2)) \vee (f(C_2) \wedge \neg f(C_1)) \\
f([\alpha_{\&}]C) & \rightarrow & (f(\alpha_{\&}) \wedge \mathbf{X} f(C)) \vee f(\overline{\alpha}_{\&}) \\
f([\beta \cdot \beta']C) & \rightarrow & f([\beta] f([\beta']C)) \\
f([\beta + \beta']C) & \rightarrow & f([\beta]C) \wedge f([\beta']C) \\
f([\alpha^*]C) & \rightarrow & f(C)\mathbf{U}\neg f(\alpha)
\end{array}$$

$$\begin{array}{rcl}
f(O_C(\alpha_{\&})) & \rightarrow & \mathbf{X}(f(\alpha_{\&}) \vee (\neg(f(\alpha_{\&})) \wedge f(C))) \\
f(O_C(\alpha \cdot \alpha')) & \rightarrow & f(O_C(\alpha)) \wedge f([\alpha]O_C(\alpha')) \\
f(O_C(\alpha + \alpha')) & \rightarrow & f(O_\bot(\alpha)) \vee f(O_\bot(\alpha')) \vee (\neg(f(\alpha) \vee f(\alpha')) \wedge f(C)) \\
f(F_C(\alpha_{\&})) & \rightarrow & \mathbf{X}(\neg f(\alpha_{\&}) \vee (f(\alpha_{\&}) \wedge f(C))) \\
f(F_C(\alpha \cdot \alpha')) & \rightarrow & f(F_\bot(\alpha)) \vee f([\alpha]F_C(\alpha')) \\
f(F_C(\alpha + \alpha')) & \rightarrow & f(F_C(\alpha)) \wedge f(F_C(\alpha'))
\end{array}$$

7.3 The Relationship between $\mathcal{CL}$ and CTL

Following the previous chapter CTL seems a better candidate for $\mathcal{CL}$ since it allows branching. We shall start again with defining the modal operators.

$$O(x) \triangleq \mathbf{AX}(x) \tag{7.1}$$

$$F(x) \triangleq \mathbf{AX}(\neg x) \tag{7.2}$$

$$P(x) \triangleq \mathbf{EX}(x) \tag{7.3}$$

The next step is to check that the Equations 7.1 and 7.2 still hold. Equation 7.1 holds since if it is always true that in the next step x is true then it is also true that there exists at least one path where x is true. Furthermore the converse is not necessarily true since if there exists a path which leads to x it does not necessarily mean that all paths lead to x. Equation 7.2 is also true since there exists one path that leads to x being true means the same as it is not true that all paths do not lead to x. Thus these deontic notions have successfully been described using CTL.

Unfortunately, CTL cannot specify fairness and liveness constraints unlike what can be done in $\mathcal{CL}$. There is no way in CTL to represent the following formula $\Box\Diamond O(\alpha)$. Thus even though it would be helpful to translate certain $\mathcal{CL}$ formulas into CTL in order to model check, we believe that it would be misleading when using formulas using **AGAF** and **AGAF** since they would not have the equivalent meaning to $\mathcal{CL}$. So, we shall refrain from giving a translation function to the limited subset of $\mathcal{CL}$ that could be represented using CTL.

7.4 Contradiction Analysis using LTL

As seen in Section 7.2, LTL is not as expressive as full $\mathcal{CL}$; however, it is expressive enough in order to represent the trace semantics. As we already know, the trace semantics is enough to check for conflicts in contracts; thus, we can draw some parallels between conflict analysis and the LTL representation of the trace semantics of $\mathcal{CL}$. Remember that there are many tools today that model check LTL so we could leverege some of the tools already present for LTL.

As described before, there are also two types of contradictions that may occur. First, a contract may always lead to a contradicting state; thus, the contract can never be satisfied. This type of contradiction is generally a question of satisfiability. The other contradiction is slightly more complicated. This is when the contract does not necessarily lead to a contradiction, but it will only in certain situations. In this case, we need to check that every path will never lead to a contradiction. It is clear that the former type of a contradiction is a special case of the latter where all paths lead to a contradiction.

7.4.1 Contracts that are not Satisfiable

Contracts that cannot be satisfied are ones that can never be obeyed since they always lead to a contradiction. Consider the following contract made up of these three clauses.

$$\Box[x]O(y) \tag{7.4}$$
$$\Box[x]F(y) \tag{7.5}$$
$$\Diamond O(x) \tag{7.6}$$

This would be translated in LTL as

$$\mathcal{G}(x \rightarrow \mathcal{X}(y)) \tag{7.7}$$
$$\mathcal{G}(x \rightarrow \mathcal{X}(\overline{y})) \tag{7.8}$$
$$\mathcal{F}(x) \tag{7.9}$$

When we join these clauses together it is clear that this contract can never be satisfied since if we satisfy clause 7.6 we do action x and once we do action x we are both obliged and forbidden to do y. This is a contradiction and thus, this contract can never be satisfied.

We can make use of a model checker to automate the search for this contradiction. What we do is check that the conjunction of all the clauses does not imply false on a free model. So if we make use of SMV we shall need to declare two boolean variables, x and a, and verify that the following assertion is true.

$$G((G(x \rightarrow X(a)) \wedge G(x \rightarrow X(\neg a)) \wedge F(x)) \rightarrow 0)$$

Using SMV, this assertion was verified to be true. This means that the contract is not satisfiable and thus will always lead to a violation. However this is not always the case as described above. Consider removing clause 7.6. Now the contract is satisfiable and will not always lead to a contradiction since if we never perform action x we will never end up in the contradiction. If we try to verify the implication without $F(x)$ the model checker will return a trace where x is never true and thus it returns a valid trace. We cannot use this technique in order to check for the existance of a contradiction.

7.4.2 Satisfiable Contracts with Contradictions

As seen in the previous section, there are situations where the contract can be satisfied in certain situations but still poses undesirable contradictions that can crop up in certain situations. In order to find such contradictions we need to augment our translation from $\mathcal{CL}$ to LTL with our modal operators since in the previous translation we have lost them.

We can add a variable for every obligation and prohibition that is true whenever there is such an obligation or prohibition. So if we consider the previous example, we need to add three variables, O_x,O_a and F_a. We want these variables to be true only when there is a modality being enforced and are not free like the other variables (a and x in this case). However, it is not easy to control these variables and the LTL formulas start to become cumbersome.

The first step is to translate the obligation and prohibition to two different variables so that now we do not have a contradiction in the LTL formula. So if we have action a we shall have two variables F_a and O_a and depending on these variables we assign a value to a. If however both F_a and O_a are true we then have a contradiction and will set some variable to signify that the state is in contradiction. We then need to verify that this variable is never set to true.

Even though the idea behind it is quite simple, the creation of the clauses are not that simple as it was before since now, unlike before, these variables inside the obligations are not free anymore. We cannot state that unless told otherwise F_a and O_a are false. We need to explicitly say this in the LTL formula. Furthermore, if we have more than one clause dictating the values of O_a and F_a we have to be careful of overlaps.

$\mathcal{CL}$ by default assumes that unless otherwise stated we are not obliged to do anything. One can also asume that things dictated by the contract are by default forbidden whilst things that are out of the control of the contract are permitted. This gives us a kind of hierarchy which to represent in LTL can become cumbersome.

7.4.3 Automata Theoretic Approach

In order to look for contradictions we can make use of an automata theoretic approach. We should be able to create a Büchi automata since we are translating the $\mathcal{CL}$ formula into LTL. During this translation we then can look for states where we have contradicting atomic propositions.

Usually when LTL formulas are translated into Büchi automata for model checking these states are discarded since no such language can be accepted. In our case we have to keep these contradictions in order to analyse which clauses lead to these contradictions.

Care however must be taken since certain contradictions need to be removed when dealing with the disjunction. The way the algorithm works, when a node is found with a disjunction it is split into two nodes, one with each possibility. In this case there could be the situation that one of the conditions leads to a contradiction but since we have a disjunction we can simply remove that state since overall the formula is still true.

Translating $\mathcal{CL}$ into LTL for Contradiction analysis

As discussed before, LTL cannot be used to describe $\mathcal{CL}$ formulas; however, in certain situations, LTL could be enough. As described before, $\mathcal{CL}$clauses cannot be translated into LTL because of its branching features, the P and $\langle\rangle$ operators. However, in the case of contradiction analysis we can omit branching as discussed in Section 5.1.1. We are analysing the contract for possibilities which lead to contradictions. We do not need to check that when we have $P(a)$ there exists a transition which performs a but what will happen when we take this transition; will it lead to a contradiction or not. The same holds for the $\langle\rangle$ operator. We do not care that there exists a transition that can do that sequence of actions but we want to check that if they are taken there will not be any contradictions.

This means that we can translate $P(a)$ to a or not a and $\langle a\rangle x$ as $(a \rightarrow \mathbf{X}x)$. So adding these translations to the one described in Section 7.2 we can now translate any $\mathcal{CL}$ formula into LTL. Even though we are loosing certain semantic meaning from the original formula, we are retaining enough for our purpose.

Furthermore, we need to preserve the deontic information in order to be able to find conflicts in satisfiable contracts. Similarly as we have done with the trace semantics of $\mathcal{CL}$ we need to augment the LTL trace semantics with the deontic notions; thus, we shall add variables in order to represent the deontic information. These variables will take the form of $O\alpha_{\&}, F\alpha_{\&}$ and $P\alpha_{\&}$ which signify that in this state one is obliged, prohibited or permitted to perform these actions. Thus, the translation into LTL will be slightly modified where

$$\begin{array}{rcl}
f(O_C(\alpha_{\&})) & \rightarrow & O\alpha_{\&} \wedge \mathbf{X}(f(\alpha_{\&}) \vee (\neg(f(\alpha_{\&})) \wedge f(C))) \\
f(F_C(\alpha_{\&})) & \rightarrow & F\alpha_{\&} \wedge \mathbf{X}(\neg f(\alpha_{\&}) \vee (f(\alpha_{\&}) \wedge f(C))) \\
f(P(\alpha_{\&})) & \rightarrow & P\alpha_{\&}
\end{array}$$

Furthermore, these variables will have a default value of false, unlike in typical LTL model checking where unspecified variables are given values non-deterministically. This notion of a default will move us away from the standard LTL and that is why we have described this as an extension.

Looking for Contradictions

After translating to LTL we need to generate a Büchi automaton that accepts only the paths that satisfy the LTL formula. This can be done using stardard techniques used in model checking. After we translate into the Büchi automatom, we search for all states that have contradicting deontic notions. Once we find these contradictions we need to return a path showing what is the sequence of events which lead to a contradiction and also mark in the original $\mathcal{CL}$ formula where the contradiction is.

Lets consider the example $[x]O(a) \wedge [x]F(a)$. In this case if x is never true, the contradiction never occurs. This would be translated from $\mathcal{CL}$ into LTL as $x \rightarrow Oa \wedge \mathbf{X}(a) \wedge x \rightarrow Fa \wedge \mathbf{X}((\neg a))$. This is then translated into negated normal form, in this case removing the implication. $Fa \wedge Oa \wedge (\neg x \vee x \wedge \mathbf{X}(a)) \wedge (\neg x \vee x \wedge \mathbf{X}((\neg a)))$. From this, a graph will be generated as seen in Figure 7.1 and we can see that when the action x occurs this will lead to a state where a contradiction occurs.

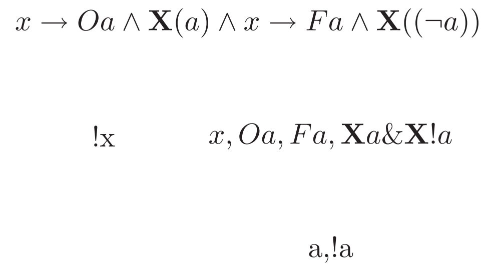

$x \rightarrow Oa \wedge \mathbf{X}(a) \wedge x \rightarrow Fa \wedge \mathbf{X}((\neg a))$
!x
$x, Oa, Fa, \mathbf{X}a\&\mathbf{X}!a$
a,!a

Figure 7.1: The formula $[x]Oa \wedge [x]Fa$ will be translated into this automaton.

We now need to go through the states and look for states that have contradictions, and in the case of Figure 7.1 we can find a state where both Oa and Fa are true; thus, this is a conflicting state. We need to show a path that leads to this conflicting state and also which parts of the original $\mathcal{CL}$ formula that have been violated in order to aid the user adjust the contract. Because of this we need to keep a reference between the different steps of the translation.

Viability of this method

Initially, we implemented a prototype using the method as seen in the previous section. Unfortunately, the initial syntax changed, moving away from the temporal operators that now would need to be defined using the dynamic logic operators. This meant that before translating to an extension of LTL, we needed to translate all the dynamic logic behaviour into its equivalent LTL formulas making it quite inefficient. Instead we opted to translate $\mathcal{CL}$ directly into an automaton rather than go through this intermediate step, however, the knowledge obtained during the implementation of the prototype was very helpful.

7.4.4 Model checking $\mathcal{CL}$ using LTL

As described before, $\mathcal{CL}$ cannot be fully expressed in LTL since it is not just linear. However, this does not mean that if we have a model $\mathcal{M}$ we cannot check that it has $\mathcal{CL}$ properties using the LTL language. When there is no branching involved we can use the previous translations and model check using that. However, when branching is involved it will take a number of steps to model check.

In order to check for permission, we need to have at least a single path that lets us do the permitted activity. This cannot be directly described in LTL but what we can do is change that permission into a prohibition and check the model for that. If the verification fails, the model checker will give us a path showing that there does exist a path that permits us to do the permitted action and thus we know that we are permitted.

The other branching operator is the dynamic logic diamond operator, $\langle\rangle$. Similar to the permission operator, we first need to check that there does actually exist a path that allows us to perform this action and this is done in the same way as the permission operator. Then, what we do is change the diamond operator to the [] operator and model check this new formula that can now be translated into LTL. One could translate $\langle\alpha\rangle C$ into $P(\alpha) \wedge [\alpha]C$ which has the same requirements, that there exists a path performing α and if that path is taken then C must hold.

Even though this "hacking" may permit us to verify that a model obeys a contract written in $\mathcal{CL}$, we believe that it would be much more desirable if we do not have to manually be checking parts of the formula but rather just directly model check the model. This full model checking will be part of our future work.

7.5 Conclusion

In this chapter we have seen how $\mathcal{CL}$ could be represented using other logics. In the case of CTL* we could translate a $\mathcal{CL}$ clause into CTL* where both formulas will accept the same traces, however, when translating into CTL* we are loosing deontic information and thus we will not be able to perform analysis which requires this information, for example conflict analysis. LTL could be used in order to represent the trace semantics of $\mathcal{CL}\dot{\text{W}}$e shown how one could model check certain properties and also how one would perform conflict analysis using this translation into LTL. This however would require the encoding of the deontic information, thus making it less feasable than the suggested solution in Chapter 5.

In the next Chapter we will give a brief overview of the implementation and also tackle a case study in order to depict how one would use the tool and techniques discussed in this work.

Chapter 8

Implementation and Case Study

In this chapter we shall go through the implementation of the automatic conflict analysis of $\mathcal{CL}$. After going through the basic implementation we shall see how we can improve on our initial implementation of the construction of the automaton from the clauses. This is then followed by the implementation of a number of different analysis techniques that we mentioned in the previous chapters. We then take a brief look at the tool we have implemented and finally end by going through a case study.

In this case study we show how contracts written in $\mathcal{CL}$ could be used in a real life situation. We also show how automating the analysis of the contract for certain properties may help during the drafting of the contract. We shall also see how useful monitoring of the system would be in real life situations, especially when the monitor is generated automatically from the $\mathcal{CL}$ contract.

8.1 Implementing conflict analysis

From the definition of the algorithm, the automaton we will generate will be defined as $A = \langle S, s_0, A_{\&}, T, l, d \rangle$ where

S is the set of states

s_0 is an element of S and it is the intial state

$A_{\&}$ is the set of possible concurrent actions (as defined in previous sections)

T is the set of transitions in the form of $S \times A_{\&} \times S$

l is the labelling function labelling the state with the clauses that need hold at this state.

d is a labelling function that labels a state with a subset of D_a (as defined at the end of Section 5.1.1)

We shall also have a special state V that is the violating state; thus, any trace that will lead to this state will not satisfy the contract.

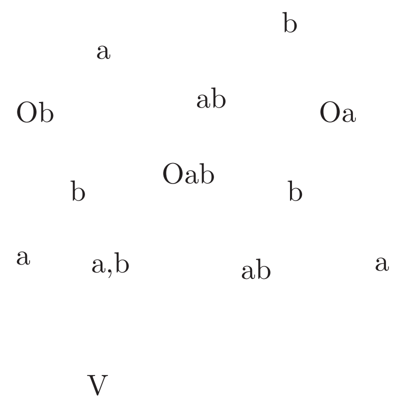


Figure 8.1: Automaton representing the contract $[a]O(b) \wedge [b]O(a)$

We shall first see the data structures we use in the implementation followed by the actual implementation.

8.1.1 The Data Structures

The main data structure used is the "Node" structure, which can be seen as a state that is still being processed[1]. The node structure will be defined as follows

Definition 8.1.1. *A Node n will contain the fields* $\langle N, O, T, D \rangle$ *where:*

N *is a set of* $\mathcal{CL}$ *formulas that still need to be processed*

O *is a set of* $\mathcal{CL}$ *formulas that have already been processed*

T *is a set of outbound transitions where an outbound transition can be seen as the tuple* $A_{\&} \times N$*. Thus it will contain an action and the node that the transition will lead to.*

D *is the set of the deontic notions that apply in this state, and thus will be a subset or equal to* D_a*.*

[1] A node that is fully processed will be translated into a state.

The algorithm will also make use of a stack U that will contain unprocessed nodes and a set S that will contain the processed nodes. We make use of these stacks instead of using a recursive call as in the original algorithm in order to avoid any potential stack overflow. Remember that the heap space is typically much larger than the stack.

8.1.2 The Algorithm

The initialisation and main loop of the algorithm can be seen in Listing 8.1. The algorithm is initialised by creating a new node and adding the full contract[2] to this new node (Lines 2-3). This node is then pushed into the stack that contains the unprocessed nodes (Line 4). After this initialisation we start the main loop (Lines 5-8) that while the stack U contains nodes, will pop a node and process the node. Once the unprocessed stack is empty, that would mean that the processing of the contract is ready after which we can translate the nodes into states and obtain the final automaton.

```
 1 GenerateAutomaton(CLContract C){
 2     n=new Node
 3     n.N={C}
 4     U.push(n)
 5     while U.hasItems{
 6         curNode=U.pop
 7         curNode.process(U,S)
 8     }
 9     return translateToAutomaton(S)
10 }
```

Listing 8.1: The algorithm starts by creating a new node that contains the full contract and pushes it into the unprocessed stack. While the unprocessed stack is not empty, nodes are popped and processed

Before we go into how we are going to process the node we wish to comment about the structure of compound actions. In [34, 54] the authors show and proove that any compound action may be translated into canonical form. For convenience we shall define it again here:

Theorem 8.1.1. *For any compound action defined in terms of* $+, \&$ *and* $\cdot$ *there exists an equivalent action called the canonical form of the form:* $+_{i \in I} \alpha^i_\& \cdot \alpha^i$*, where each* $\alpha^i_\& \in A^\&_B$ *and* α^i *is another action in canonical form.*

Furthermore we discussed the extention of the canonical form for the case of the Kleene star in Section 5.1.3. Once in Canonical form we can look at an action as a disjunction of actions that must occur now and for each of these a compound action that needs to hold in the next step. This is a

[2] A full contract is the conjunction of all the clauses making up the contract

very helpful way of looking at a compound action while processing the node since we can translate a compound action α into an array of possibilities α_i where for each entry we have the atomic actions that need to hold now ($\alpha_i.now$) and the possible compound or empty actions that need to follow at the next instance($\alpha_i.next$). Let us consider the compound action $(a\&(d + (b \cdot a))) + (c \cdot (b + (c \cdot a)))$ that can be translated into the canonical form $(a\&d) + ((a\&b) \cdot a) + ((c \cdot (b + (c \cdot a))))$ which then can be translated into array form $[ad|, ab|a, c|b + (c \cdot a)]$ where for compactness each element in the array is of the form $now|next$. This will help us split the concerns; thus, we can pass on elements that do not need to hold now to other nodes which come further down the trace. With regards to the implementation, we will assume that all compound actions will be represented in this form. It should be straightforward to see that any compound action written in canonical form can be traslated into this form.

We shall now look at how we shall process the nodes. The pseudocode can be found in Listing 8.2. The first thing we do when we start to process a node is check if the set of new formulas N is empty (Line 2). If it is empty that means that there are no more formulas to process and thus we can add this node to the set of finished nodes S (Line 3). If on the other hand N is not empty, that means that we have formulas still to process.

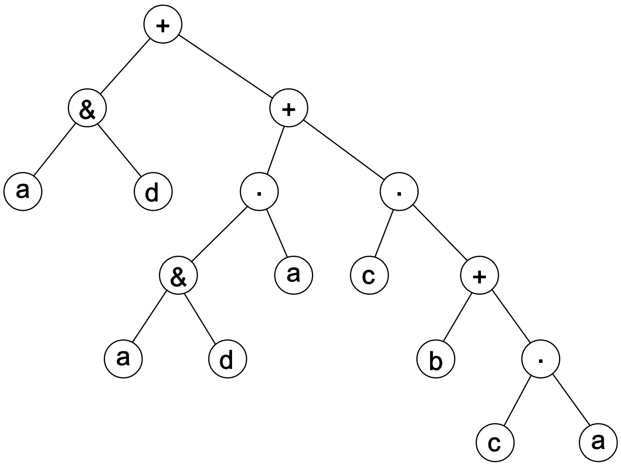


Figure 8.2: This is how the compound $(a\&d) + ((a\&b) \cdot a) + ((c \cdot (b + (c \cdot a))))$ would be parsed

In order to keep the algorithm simple we generate all the possible transitions and create new nodes at the end of these transitions. This is done with the method call on Line 5, which will create a transition for each element in the set $A_{\&}$. Once this is done, we get an element from the set N of formulas and check which is the root element of the formula. We are talking about the root element since the formulas have been parsed into a parse tree. We shall not discuss the details of the algorithm since it is a standard technique.

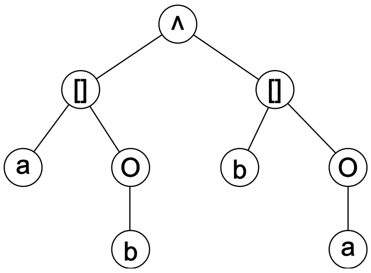


Figure 8.3: This is how the formula $[a]O(b) \wedge [b]O(a)$ would be parsed

In Image 8.2 we see an example of a parsed compound action and similarly Image 8.3 we see an example of a parsed $\mathcal{CL}$ formula.

```
1  Node.process(Stack U, Set S){
2      if(N.empty()){
3          S.add(this)
4      }else{
5          generatePossibleTransitions()
6          while(!N.empty()){
7              c=N.get()
8              if(c.isConjunction){
9                  N.add(c.leftBranch)
10                 N.add(c.rightBranch)
11             }else
12                 if (c.isObligation){
13                     addDeonticInfo(O(c.rightBranch))
14                 elseif (c.isProhibition){
15                     addDeonticInfo(F(c.rightBranch))
16                 }elseif (c.isPermission)
17                     addDeonticInfo(P(c.rightBranch))
18
19                 forall t in N.T
20                     t.State.N.add(process(c,t))
21
22                 N.groupNodes()
23                 N.checkAlreadyProcessed()
24                 N.addNodestoU()
25             }
26         }
27     }
28 }
```

Listing 8.2: Algorithm for Processing the node

The simplest operator to process is the conjunction. When the root of the formula being processed is the conjunction operator we split the formula in two (take the left branch and right branch of the parse tree) and add both formulas to the set N (Lines 8-10). Thus if the set N is equal to

$\{[a]O(b) \wedge [b]O(a)\}$ once the formula is processed it will then contain two formulas instead of the initial one and will be equal to $\{[a]O(b), [b]O(a)\}$. We can intuitively see why this is the correct behaviour since we want that both formulas hold in this state.

For the rest of the operators we shall have to call another function. If the operators are either Obligation, Prohibition or Permission, we add deontic information to the set D (Lines 13, 15 and 17 respectively).

After this we go through all the possible transitions always calling the process function and adding the return value to the node to which the transition leads to. The return value from this function is the formula that is left to be satisfied in the node following the transition. The code of the process function can be found in Listing 8.3.

```
Node.process(Clause c,Transition t){
    Clause toSatisfy1 , toSatisfy2
    Clause tmpClause, reparation
    if(c.isConjunction){
        toSatisfy1=process(c.leftBranch ,t)
        toSatisfy2=process(c.rightBranch ,t)
        if(toSatisfy1==0 or toSatisfy2==0){
            return 0
        }elseif(toSatisfy1==toSatisfy2==1){
            return 1
        }else{
            return toSatisfy1 and toSatisfy2
        }

    }elseif (c.is[]){
        $\alpha_i$=c.leftBranch
        tmpClause=c.rightBranch
        forall a in $\alpha_i${
            if($a.now \subseteq t.Action$)
                if($a.next \neq \emptyset$)
                    return $[a.next] tmpClause$
                else
                    return tmpClause
            else return 1
        }
    }elseif (c.isObligation){
        $\alpha_i$=c.rightBranch
        reparation=c.leftBranch
        $new\alpha_i$=new Array()
        satisfied=false
        forall a in $\alpha_i${
            if($a.now \subseteq t.Action$)
                if($a.next \neq \emptyset$)
                    $new\alpha_i$.add($[a.next]$)
                else
                satisfied=true
        }
        if(satisfied)
            return 1//Obligation satisfied by taking this transition
        elseif($new\alpha_i$.size>0)
            return O($new\alpha_i$,Reparation)
        else
            return reparation
}elseif (c.isProhibition){
        $\alpha_i$=c.rightBranch
        reparation=c.leftBranch
        $new\alpha_i$=new Array()
        violated=false
        forall a in $\alpha_i${
            if($a.now \subseteq t.Action$)
                if($a.next \neq \emptyset$)
                    $new\alpha_i$.add([$a.next$])
                else
                    violated=true
        }
        if(violated)
            return reparation
        elseif($new\alpha_i$.size>0)
            return F($new\alpha_i$,Reparation)
        else
            return 1\\Prohibition has been satisfied
}elseif (c.isPermission){
        $\alpha_i$=c.rightBranch
        $new\alpha_i$=new Array()
        forall a in $\alpha_i${
            if($a.now \subseteq t.Action$)
                if($a.next \neq \emptyset$)
                    $new\alpha_i$.add([$a.next$])
        }
        return P($new\alpha_i$)
}elseif (c.isExclusinveOr){
        toSatisfy1=process(c.leftBranch,t)
        toSatisfy2=process(c.rightBranch,t)
        if(toSatisfy1==toSatisfy2==0 or toSatisfy1==toSatisfy2==1){
            return 0
        }elseif((toSatisfy1==1 and toSatisfy2==0)or(toSatisfy1==0 and toSatisfy2==1)){
            return 1
        }else{
            return toSatisfy1 xor toSatisfy2
        }
}
```

```
82 }
```

Listing 8.3: Algorithm for Processing clauses

Looking at Listing 8.3 we would realise that we are again processing the conjunction operator (Lines 4-14). The first reaction would be to conclude that, we do not really need both and thus we could remove this and leave the one in the other function (Listing 8.2). We shall see why we need it in this function once we take a look at how we process the exclusive or. We process the conjunction by first processing the left branch by calling the same function but passing the left branch as a parameter and then process the right branch. Then, if either result is 0 (i.e. violation) then the whole conjunction has been violated and thus, we can return 0 as well. If both are satisfied then we can return 1 (the conjunction has been satisfied) otherwise, that means that there are still items left to satisfy and thus we return the conjunction of both. One should note that the overall result of how we process the conjunction here is identical to how we process the conjunction in Listing 8.2. However, here, we keep the structure of the conjunction (this will be needed when processing the exclusive or). What one should note as well is that the algorithm would still work if we removed the part where we processed the conjunction in Listing 8.2 (Lines 8-10).

If the operator to process is the square brackets ($[\alpha]C$) then the left branch will contain an action and the right branch will contain a clause. The action has been translated into canonical form and then into the array form; thus, from α we get α_i. To process this (Lines 15-25) we shall need to go through all the transitions and for each transition we go through all of the elements in α_i and if $\alpha_i[n].now$ is a subset or equal to the label of the transition then we add $[\alpha_i[n].next]C$ to the set N of the node to which the transition leads. If $\alpha_i[n].next]$ is empty then we simply add C instead of adding the empty square brackets. Image 8.4 shows an example of the structure generated when processing the square brackets.

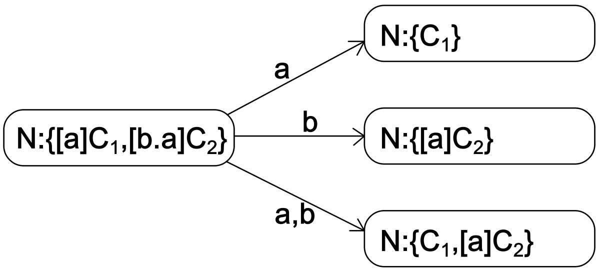


Figure 8.4: When processing the [] operator we go through all the transitions and check which satisfy the requirement inside the square brackets and add accordingly to the nodes what is left to be satisfied

The next operator we shall look at is the obligation. When an obligation is parsed its right branch will be the action that is obliged and the left branch will be the reparation. Processing an obligation is more demanding than the square brackets (Lines 26-43). Again, we shall need to go through all the possible transitions and check if by taking the transition the obligation is satisfied in which case we would not need to add anything else to the node. If however it does not satisfy the obligation then we will need to add the reparation to the node. If there is no reparation than a violation has occured thus we add the special marker 0.

However, it is not as simple since we have compound actions. We shall have to go again through all the items in the array representation, however this time we need to keep additional information. To do this we create a boolean variable "satisfied" that is set to false (Line 30) and an empty action array "$new\alpha_i$". If the transition satisfies one of the elements of α_i that does not have anything left to satisfy, then we set the "satisfied" variable to true (Line 36); thus, signifying that by taking this transition the obligation has been satisfied. If however, there are actions left to be satisfied then we add these to "$new\alpha_i$" (Line 34) since the obligation has not been fully satisfied but has still parts of the obligation to be fullfilled. Once we go through all the elements of α_i we first check if "satisfied" is true, in which case the obligation has been fulfilled and we do not need to add anything to the state reached with this transition. If not, we then check if $new\alpha_i$ contains any elements. If so, we add to the node pointed to by the transition the formula $O(new\alpha_i)$ while keeping also the original reparation. If however $new\alpha_i$ is empty, that means that this transition has not satisfied (or partially satisfied) neither of the possible requirements and thus, this transition would lead to the violating state.

We first check if "satisfied" is set before check if $new\alpha_i$ has elements since there is a disjunction between the elements of α_i and so if just one element is fully satisfied then the whole compound action has been satisfied.

Prohibition is similar to Obligation (Lines 44-61). Instead of defining a "satisfied" variable we define a "violated" variable that will be set if an element of $\alpha_i.now$ is satisfied by the label on the transition and the element's next field is empty. If however the next element is not empty then we add this to the array $new\alpha_i$. Once all the elements of α_i have been processed we first check if the "violation" variable has been set to true, in which case we add the violation marker. If not, we then check if the array $new\alpha_i$ is empty. If it is not empty then we add $F(new\alpha_i)$ together with the corresponding reparation to the node pointed to by the transition. If $new\alpha_i$ is empty then that would mean that the prohibition has been satisfied.

Permission is slightly simpler than obligation and prohibition since when using the trace semantics we cannot know if a permission has been violated and thus a permission will never lead to a violation. The code handling the permission can be found between Lines 62 and 70. It will go through every transition and check which values of the 'now' elements of α_i are satisfied by the transition label and add their next value to $new\alpha_i$. Then add $P(new\alpha_i)$ to the node that is pointed to by the transition.

The exclusive disjuction is a slightly trickier operator and as discussed in Section 3.3 has a number of issues however here we will simply try to model the trace semantics given to the exclusive or. The semantics of the exclusive disjunction in the original trace semantics were:

$$\sigma \vDash C_1 \oplus C_2 \text{ if } (\sigma \vDash C_1 \text{ and } \sigma \nvDash C_2) \text{ or } (\sigma \nvDash C_1 \text{ and } \sigma \vDash C_2)$$

In order to process this we do the same as we did with the conjunction. We process the left and right branches on their own. Then depending on the values returned, we either return satisfaction, violation or else return a new formula that needs to be satisfied. So, if the results from processing the right and left formulas are either both 1s or 0s then the exclusive or has been violated and thus, we return 0. If however, one branch is satisfied and the other is not, then we have successfully satisfied the exclusive or and thus, we can return 1. If one of the branches has still to be satisfied/violated (i.e. when processing it, the result was neither 0 nor 1) then we shall take both results, join them with an exclusive or and return them since that is the new formula that has to be satisfied in that node.

It is because of this behaviour that we need to process the conjunction in this way (preserve the conjunction structure). Consider the following formula $C_1 \oplus (C_2 \wedge C_3)$[3]. We cannot simply split the conjunction since it is inside the exclusive or and thus, we need to keep this structure. When we process the exclusive or, we first see what is returned when we process C_1 and then process the right branch, which contains a conjunction effectively processing C_2 and then C_3 and getting back the result of their conjunction.

Once we have seen how we process the formulae we go back to Listing 8.2 where we continue with analysing what nodes have been created. In order to simplify the automaton and reduce the ammount of processing we have to do, we can join nodes with identical values in the set N into one node. This entails the algorithm to go through the set of transitions and compare the nodes to which they point to. When two identical nodes are found, one of the nodes is dropped and the transition which pointed to that node is made to point to the node that was not discarded.

After this, we do the same but this time check if we have already processed a node with the same formulae. This is done by going through the set S of already processed nodes and check if the set O contains the same values as in the set N of any of the nodes pointed to by any of the transitions. If there exists such a node, it is dropped and the transition made to point to the old one.

[3] The syntax allows the exclusive or to reside only between two obligation clauses or two permission clauses however as discussed in Section 3.3 we may end up having to process such clauses

Finally we add the nodes that are left to be processed to the stack U.

8.1.3 Translating Action into Array form

As discussed previously, we are translating an action into this array form in order to make the main algorithm simpler. We are splitting a compound action into an array of possible actions, and each element of this array is split into two, the action that needs to be observed/satisfied now, and the rest of the action that needs to be satisfied in the next step.

The algorithm assumes that it is given an action that is already in canonical form. The code can be found in Listing 8.4.

```
 1  actionToArray(Action a){
 2     \\a is already Parsed and in Canonical Form
 3     Stack pStack=new Stack
 4     Array actionArray=new Array
 5     Processing.Push(a)
 6     while(!Processing.isEmpty){
 7        Action a=pStack.Pop
 8        if(a.isDisjunction){
 9           pStack.Push(a.LeftBranch)
10           pStack.Push(a.RightBranch)
11        }else if(a.isSequence){
12           ActionItem item=new ActionItem
13           item.now=a.LeftBranch
14           item.next=a.rightBranch
15           actionArray.add(item)
16        }else if(a.isConcurrent){
17           ActionItem item=new ActionItem
18           item.now=a
19           item.next=∅
20           actionArray.add(item)
21        }else if(a.isStar){
22           ActionItem item=new ActionItem
23           item.now=a
24           item.next=∅
25           actionArray.add(item)
26        }
27     return actionArray
28  }
```

Listing 8.4: Algorithm for translating an action into an Array

The algorithm starts by creating a Stack (pStack) that will contain sub parts of the action which still need to be processed and the Array (actionArray) which will be the array that is returned with the final result. The algorithm starts by initialising the array and stack followed by the pushing of the action to the processing stack (Lines 3-5).

While the stack is not empty, we shall pop an action and process it. If the root element is a disjuction (+) then we push both the left and right branches into the pStack (Lines 8-10). The disjunction means that both branches are possible to satisfy the action and that each item in the array will stand for a possibility; thus, one can see a disjunction between each entry in the array. We push the branches back into the stack rather than processing them since it might be the case that there is another disjunction.

If the root element is a sequence then we create a new array element *item* and make $item.now = a.LeftBranch$ and $item.next = a.RightBranch$ (Lines 11-15). We should note that the way we translate into canonical form restricts sequences to only have a concurrent action in the left branch. Again, we are preserving the semantics of the actions. There, the action occuring before the sequence symbol is placed in the now element and the rest in the next element.

When the root element is a concurrency operator, we create a new array element *item* and make $item.now = a$ and make $item.next = \emptyset$ (Lines 16-20). This is quite straightforward since all the concurrent actions must occur now and thus, the next step is left empty.

If the root element is a Kleene Star we simply add it to the array and it is left up to the rest of the algorithm to process the Star.

8.2 Improving Implementation

In this section we shall see a number of improvements that we have made to our basic implementation. These improvements will not only allow us to analyse larger contracts but also in shorter time.

8.2.1 Encoding Actions

During the construction of the automaton we are repetitively comparing concurrent actions together. Consider that we are creating an automaton for the clause $[a]O(b)$. This will create the first state labelled $[a]O(b)$. Then we generate all possible transitions and check if a is a subset or equal to the label on the transition. In this case we do three comparisons between actions. However note that as the number of actions increases the number of comparisons will increase exponentially since the possible action combinations will increase exponentially.

By making this comparison as efficient as possible we shall improve the speed of the algorithm. If we keep actions in the form of a parse tree, a comparison between concurrent actions would require us to traverse the tree and compare between the leaves. This will be quite a heavy computation since we need to list the leaves of one concurrent action and check if the are present in the other action.

In order to improve on this point we encode concurrent actions as integers. We do this by assigning a bit position to each atomic action and set that bit position to 1 when that action is present. Thus if the list of possible atomic actions is $\{a, b, c, d\}$, we would represent action $b\&d$ as 0101, which is equal to 5. This will clearly reduce the size of representation since now we do not have to store the entire tree structure but instead only the number representing the concurrent action. Let f_e be the function that will encode concurrent actions in this way. Thus given a concurrent action f_e will return an integer ($f_e : A_{\&} \rightarrow I$).

Comparing between actions is also simplified. The comparison that is mostly used is the subset or equal to ($\subseteq$). Suppose we want to check if $\alpha_{\&} \subseteq \alpha'_{\&}$. We perform a bitwise conjunction between $f_e(\alpha_{\&})$ and $f_e(\alpha'_{\&})$ and if the result is equal to $f_e(\alpha_{\&})$ then $\alpha_{\&}$ is a subset or equal to $\alpha'_{\&}$. Otherwise it is not.

Furthermore, we do not need to stay encoding the actions just before comparison but we can do this once we translate the contract into canonical form and thus, will have concurrent actions already encoded.

This encoding will also improve the generation of all possible transitions. Instead of creating the formulas of all possible combinations of actions we can simply loop from 1 till the maximum number of combinations ($|A|^2$) and this will go through all possible actions. This generation of all possible transitions typically occurs when processing every state.

With this encoding we shall reduce the size of the space required to generate the automaton and will also make the generation of the automaton faster.

8.2.2 Generating only required transitions

When processing a node we create all possible transitions and then add what is left to be satisfied in the following state. The problem is that the number of transitions will increase exponentially to the number of atomic actions. Also, each state will have a constant (maximal) number of transitions leaving the state. Thus, if we create an automaton for a clause that has five different atomic actions and will result in having five states, we shall have 125 transitions. If instead we have 10 different atomic actions and 10 states, we shall have 1000 transitions. Most of the time this results in having more transitions than actually required because in most states there will be actions that will not affect the outcome of the transition. Furthermore, there will be a number of transitions from one state to another all labelled with different concurrent actions that will result in the same behaviour.

Consider processing the formula $[a]Ob \wedge [b]Oc$. If we generate all the transitions the result would be the automaton in Figure 8.5. We should note that out of each state we have all possible transitions (we grouped transitions as ‘otherwise’ in order to make the automaton more readable).

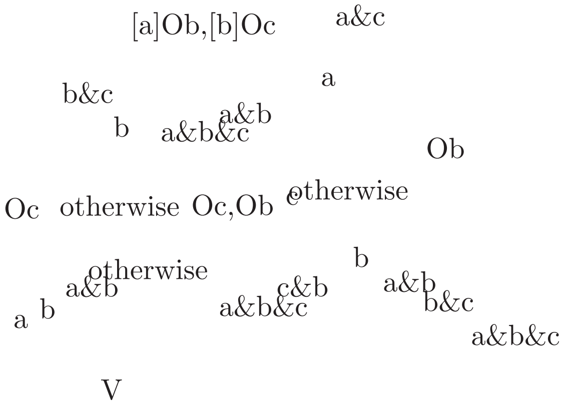


Figure 8.5: Automaton generated for $[a]Ob \wedge [b]Oc$ with all the transitions being created.

It would be desirable if we could group transitions that go from the same initial and destination state into one; thus, reducing the maximum number of transitions. Thus, in the first case we shall have a maximum of 25 transitions whereas in the second case we would have a maximum of 100 transitions. In the case of the $[a]Ob \wedge [b]Oc$, instead of generating the automaton in Figure 8.5 which has a total of 54 transitions we would generate the automaton in Figure 8.6 which has a total of 12 transitions.

This will however require us to possibly label transitions not only with concurrent actions but we shall also have to encode choice. This will also make the automaton more complex to execute. Instead we can go for a solution which is a compromise between the two. We will reduce the number of transitions while still keeping transitions labeled only with concurrent actions (i.e. integers since we are encoding the concurrent actions).

Instead of creating all the transitions before we start processing the node, we generate only the required transitions and states as we process the node. Consider processing the initial node where $P = [a]Ob, [b]Oc$. We start without any transitions and add new transitions as required while processing the sub formula. When processing the first sub formula we add a transition with a and place Ob in the new node. However when we process the second sub formula we cannot simply add a transition with action b and add Oc to the new node. We shall have to go through all the transitions already created and create a transition that is a combination of both actions. In this case we shall need to create a transition $a\&b$ and add both Ob and Oc to the new node. The result can be seen in Figure 8.6.

When traversing the automaton we shall now need to take the transition with the maximal possible match. Consider that we are in the first state of the automaton in Figure 8.6 and the action $a\&b$ is observed. Even though

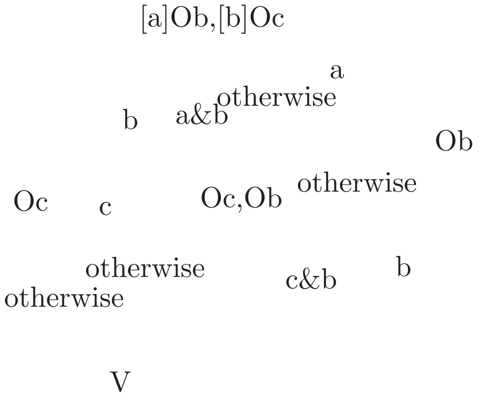


Figure 8.6: By creating only the necessary transitions the corresponding automaton for $[a]Ob \wedge [b]Oc$ is much smaller.

actions b and a are a substet of $a\&b$ we cannot take the transitions labeled with these actions since there is another transition labelled with $a\&b$ that is a better match.

If we want to encode choice, this could be done by labelling each trace with an array of integers rather than with only one integer and thus the array will represent all the possibilities that would satisfy the transition. This is also a viable possibility however we did not do it this way since when processing actions the choice is always split (eg. $[a + b]O(c) \to [a]O(c) \wedge [b]O(c)$) and thus this would require us to rejoin the actions together later which means further processing, however if the size of the automaton of the contracts one is trying to analyse, this would be a viable solution.

8.2.3 On-the-fly conflict analysis

Another viable possibility when the automaton for the contract is too large to analyse is to make the conflict analysis on the fly. Instead of generating the automaton first and then analysing the contract for conflicts we may check for conflicts once a node has been processed since it would already be labelled with the deontic requirements in that state. Furthermore we do not need to store transitions between states. We only need to store the sub formulae which have been processed and the formulas which are yet to be processed. We need to store the sub formulae that have already been processed in order for the algorithm to terminate.

If the contract has a conflict, it may identify that the contract has a conflict before all the reachable sub formulae have been analysed. However, since we have not generated the automaton we shall only have a listing of the conflicting sub formula and we are not able to generate traces that lead to the conflict. This method would fit well in order to ensure that a contract is conflict free. However, the types of analyses are limited when compared to generating the entire automaton.

8.3 The Tool

From the description given in the previous section, the tool developed performs conflict analysis, superfluous clause analysis and also generation of Larva code. The tool implemented has got a graphical user interface in order to make it accessible even for non computer scientists who are used to have this kind of friendlier interface. In Figure 8.7 we can see a screen shot of the application. One can add clauses to the contract by entering the clause in the top text box and clicking on the add clause button. The clause will then be placed in the list view shoing all the clauses. In order to remove from the list, select and click on the remove clause. Once the contract is ready for analysis click on Analyse. The results will be displayed in the text area below together with other information like a textual representation of the graph and also the code generated for LARVA.

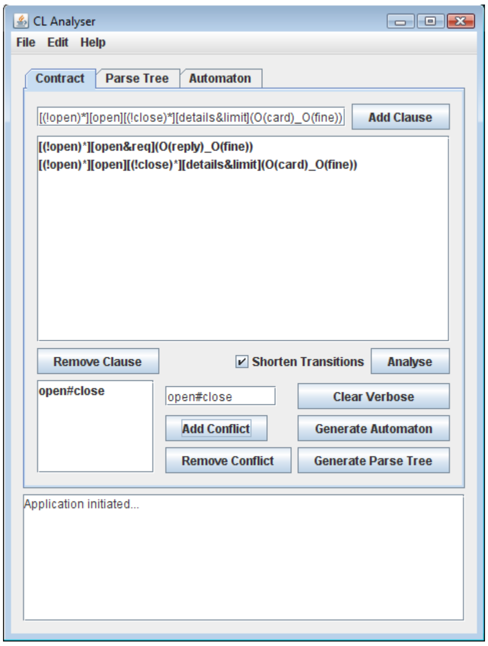
To add conflicts you enter the conflict using the '#' symbol as a seperator between actions, and similar to the conflicts use the appropriate buttons ot add and remove mutually exclusive actions. There is also the option to shorten transitions which will group certain actions into an 'otherwise' group in order to reduce the number of actions visible.

After we click the analyse button we may choose to generate the automaton or the parse tree. These will be displayed on the appropriate tabs an example of which can be seen in Figure 8.8. This is the automaton generated for the contract $[c]O(b) \wedge [a]F(b)$ which has a conflict and the state in which the conflict is present is marked red.

The appropriate syntax to be used is can be found in Table 8.1.

$\mathcal{CL}$ syntax	Tool equivalent
$C \wedge C$	C^C
$C \oplus C$	C-C
$O_C(a)$	O(a)_C
$F_C(a)$	F(a)_C
$P(a)$	P(a)
$[a]C$	[a]C
$[a \cdot b]C$	[a.b]C
$[a + b]C$	[a+b]C
$[\overline{a}]C$	[!(a)]C

Table 8.1: Syntax to be used when defining contracts for application



```
[(!open)*][open][(!close)*][details&limit](O(card)_O(fine))
```

```
[(!open)*][open&req](O(reply)_O(fine))
[(!open)*][open][(!close)*][details&limit](O(card)_O(fine))
```

```
open#close
```

```
open#close
```

```
Application initiated...
```

Figure 8.7: Screen shot of implemented tool

8.4 Case Study

With this case study we are going to show how straight forward it is to create a contract using $\mathcal{CL}$. We also show how valuable it is to look for conflicts, superfluous clauses and reachability all of which are automated once the description is in $\mathcal{CL}$. Furthermore, we shall show how queries and simulation can be used in order to help the understanding of the contract and to increase the confidence in the contract. We will finally see how monitoring could be applied once the contract has been signed and put into action.

The approach taken is that we start with a draft of a contract written in english. From this contract we create a $\mathcal{CL}$ representation of this same contract. Using the tool and techniques described in this work we analyse the contract and incrementally improve the contract in order to fix any errors the contract might have. Once we are satisfied with that the $\mathcal{CL}$ contract is what is actually required by the signatories, we then translate it back into its english, textual representation.

8.4.1 The Scenario

We are going to take a look at the airline industry. Consider that we have two players in this scenario, the airline company and a company taking care of the ground crew (mainly the check-in process) This is a typical scenario

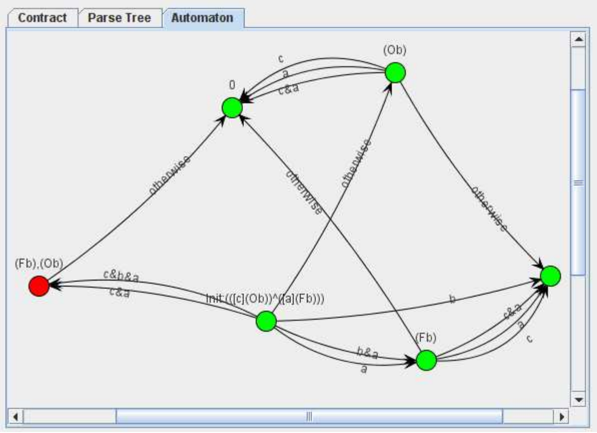


Figure 8.8: Automaton generated for $[c]O(b) \wedge [a]F(b)$

for small airline companies when operating from large airports.

In such a situation each party would like to protect itself and this is done by writing a contract. The airline company would like to give enough time for its clients to check-in so that the travellers can comfortably check-in in a reasonable amount of time without incurring having to wait in long queues. Also, the airline wants to have the plane ready for takeoff on time since any delays would possibly mean having to pay liabilities to the travellers that miss connecting flights. In order for the luggage to be loaded on time the loading crew would need to get the information of all luggage weights and the luggage themselves in time to balance the containers with the luggage and load the luggage themselves. All these requirements will fall on the check-in company. The check-in company is required to open the check-in desk two hours before the flight leaves giving passengers ample time to check-in, without having the problem of long queues that end up delaying the flight. Furthermore, they are required to close the check-in desk and send the luggage information to the loading personnel 20 minutes before the flight has to leave.

After this brief introduction we are going to see what points should be written in the contract. The airline would require the ground crew to open check-in desk two hours before the flight leaves and the ground crew would require that the airline provides them with the passenger information once the check-in desk is open. Furthermore, the airline would require the ground crew to check and confirm that the passport details match the passenger details on the ticket and that the luggage weighs within the required amount. If the luggage weighs more, then they are obliged to charge the passenger extra. The ground crew is also obliged to send all the luggage information and luggage 20 minutes before the flight leaves. The reparation to each of these requirements would be in monetary form (payment of a fine).

At the centre of this system is a messaging hub (Enterprise Service Bus), managing the routing of information between the three parties. When opening the check-in desk, a request is made to the hub to get all the passenger information from the airline system. Once we close the check-in process the passenger information and luggage information is sent to both the airline system and the loading system. Since all these messages are passing through the message hub, we can place a monitor inside the Service Bus which could make sure that the contract is adhered to and will automatically detect violations and will then inform the respective central offices in order to impose the monetary reparations.

8.4.2 The Contract

The first step would be to draft a candidate contract. This will be followed with the analysis of the contract in order to ensure that it has no conflicts and also to refine and improve the contract. We will then see how we would generate a monitor directly from the contracts.

Contract between airline and ground crew

1. *The ground crew is obliged to open the check-in desk and request the passenger manifest two hrs before the flight leaves.*

2. *The airline is obliged to reply to the passenger manifest request made by the ground crew with the passenger manifest*

3. *After the check-in desk is open the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass*

4. *If the luggage weighs more than the limit, the crew are obliged to collect payment for the extra weight after which they are obliged to issue the boarding pass*

5. *The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave.*

6. *After closing check-in, they are obliged to send the luggage information to the airline.*

7. *If any of these obligations are violated a fine is to be paid*

This contract will regulate on how the ground crew will handle the clients of the airline agency. One should find all the information discussed in the description of the case study described in some form or another in the contract.

Before giving the $\mathcal{CL}$ representation, we need to define the actions that are going to be used in the contract. Up until now, we have not been doing this when defining contracts in $\mathcal{CL}$ since all the examples given where straightforward. However, defining the actions is required when defining contracts using $\mathcal{CL}$ since the actions are to be defined by the users. The definitions of the actions may be found in Table 8.2.

Action	Definition
$2hrs$	This action is observed once it is 2 hours before the flight leaves
$fine$	A monetary fine is to be paid
$openCheckIn$	Ground crew opens the check-in desk
$requestInfo$	Ground crew requesting passenger information from airline
$replyInfo$	The airline sends the passenger information to the groud crew
$correctDetails$	A traveller with the matching passaport and ticket details is checking-in
$luggageInLimit$	The weight of the luggage is within the desired limit
$boardingCard$	Ground crew issues a boarding card to the traveller
$collectPayment$	Ground crew collects payment due to the luggage being over the weight limit
$20mins$	This action is observed once it is 20 minutes before the flight leaves
$closeCheckIn$	Ground crew closes the check-in desk
$sendLuggageInfo$	Ground crew sends the information about the checked-in luggage and passengers to the airline

Table 8.2: Action definition of contract being used for the case study

The contract could be represented in $\mathcal{CL}$ as follows:

1. $[2hrs]O_{O(fine)}(openCheckIn \& requestInfo)$

2. $[requestInfo]O_{O(fine)}(replyInfo)$

3. $[openCheckIn][correctDetails \& luggageInLimit]$
   $O_{O(fine)}(boardingCard)$

4. $[openCheckIn][correctDetails \& \overline{luggageInLimit}]$
   $O_{O(fine)}(collectPayment \cdot boardingCard)$

5. $[20mins]O_{O(fine)}(closeCheckIn)$

6. $[closeCheckIn]O_{O(fine)}(sendLuggageInfo)$

8.4.3 Analysing the Contract

After producing the first draft of the contract, we shall make use of the tools we developed in order to analyse the contract and fix any problems in the contracts that the tools may detect.

The first step would be to input the contract into our tool and analyse it for conflicts. This will then be followed by analysing the contract for conflicts and for reachability of all the deontic restrictions. This step is performed after every change made to the contract. As is, this contract does not have any conflicts which is good. However, that does not mean that it has the desired behaviour. If we query for non violating traces which have an action which is $boardingCard$ (which stands for issuing a boarding card) we realise that we have traces where a boarding card is issued without checking that the client has the correct details. This is a huge security risk which surely is not desired. In order to fix this issue we need to add a clause which ensures that we cannot issue a boarding card without first confirming the correct details. We will thus add the clause $[\overline{correctDetails}]F(boardingCard)$. When we execute the same query again in order to check that the problem has been solved we realise that this however has not solved the security issue. We need to enforce that this clause holds in all the states not just the first one and thus we need to modify it to $[1*][\overline{correctDetails}]F(boardingCard)$. This is again not enough since we will notice traces which immediately start witch issuing the boarding card. In the first state we are not actually prohibiting the issue of the boarding card. We only have a formula which states that if the correct details are not shown then a boarding card cannot be issued. We will need to add the prohibition of issuing a boarding card in the first state and thus we will need to change the clause to $[1*][\overline{correctDetails}]F(boardingCard) \wedge F(boardingCard)$.

When we check this modified contract for conflicts we realise that with these modifications we have introduced a conflict. We get the following trace that leads to the conflict $openCheckIng \cdot correctDetails\&luggageOverLimit \cdot collectPayment$ and the state is labelled with $O_{boardingCard}$ and $F_{boardingCard}$. We have just forbid the handing of boarding cards except exactly after the correct details have been supplied. We will thus have to make changes to the prohibition to allow the possibility of collecting the payment before issuing the boarding card. However, an easier approach which will keep the contract written in $\mathcal{CL}$ cleaner would be to make the collection of the payment and the issue of the boarding pass a concurrent action. Thus now Clause 4 becomes:

$$\begin{gathered}[openCheckIn][correctDetails\&\overline{luggageInLimit}] \\ O_{O(fine)}(collectPayment\&boardingCard)\end{gathered}$$

We now query for traces leading to a state where the crew is obliged to issue a boarding card. One thing we should realise is that there is no loop in the trace; the crew is being obliged to issue a boarding card only once. We would like that this will happen repeatedly once the cash desk is open and not just do it for the first client thus we will need to change Clauses 3 and 4 to

$$[openCheckIn][1*][correctDetails\&luggageInLimit]O_{O(fine)}(boardingCard)$$

and

$$[openCheckIn][1*][correctDetails\&\overline{luggageInLimit}]$$
$$O_{O(fine)}(collectPayment\&boardingCard)$$

When we check again the traces that lead to the crew to be obliged to issue a boarding card we now see that as desired we have a loop thus they will be obliged to process not just the first customer. Unfortunately we see another fault. Customers are being checked in after the check-in desk has been closed. One might say that this clause should be taken for granted since it should be obvious that once the check-in desk is closed no more people should be allowed to check-in. However, if it is not explicitly written in the contract, a ground crew member might pity a late passenger and check-in his bags after check-in has closed thus giving rise to the possibility of the bag not making it to the plane on time. In order to prevent any check-in to occur after the check-in desk is closed we add the following clause $[1*][closeCheckIn][1*]F(boardingCard)$. This however introduces a conflict since after the check-in is opened it is always the case ($[1*]$) that if the client has the correct details then the crew is obliged to issue a boarding card. This must be changed to while the desk is open ($[\overline{closeCheckIn}^*]$). Thus the Clauses 3 and 4 become

$$[openCheckIn][\overline{closeCheckIn}^*][correctDetails\&luggageInLimit]$$
$$O_{O(fine)}(boardingCard)$$

and

$$[openCheckIn][\overline{closeCheckIn}^*][correctDetails\&\overline{luggageInLimit}]$$
$$O_{O(fine)}(collectPayment\&boardingCard)$$

This however still has not solved all the conflicts. Consider the situation where we first open the check-in desk and then we close it and receive correct details concurrently. In the following step we are both obliged and forbidden from issuing a boarding card. To solve this we will make the closing of the desk and the receiving of the passenger details as mutually exclusive actions thus prohibiting that these actions occur at the same time.

In order to increase the confidence in the contract we may decide to simulate the contract. We realise that there is the possibility that before check-in opens, if a client issues the correct details then the crew may issue the boarding card. This again is undesirable since we want cards only to be issued after the check-in desk is opened, thus we need to add $[\overline{openCheckIn}^*]F(boardingCard)$. Again using the simulation we may realise that we are permitted to open the check-in desk after it has already been open. This might seem redundant but it does not really pose any danger. However what about being able to open the check-in desk after it has already been closed? Right now this is still permitted however this still would not allow us to check-in any luggage and there is no conflict either since after the check-in is closed we are no longer obliged to check-in luggage. However, for clearness sake this should be prohibited and thus the clause which prohibits us to issue any more boarding cards after the check-in desk is closed should be changed to $[1*][closeCheckIn][1*]F(boardingCard + openCheckIn)$. Please note that this is equivalent to

$$[1*][closeCheckIn][1*](F(boardingCard) \wedge F(openCheckIn))$$

which when translated into textual form has a clearer and more direct representation.

Now we turn our focus to the first two clauses. First of all, after the experience already obtained from the previous cases, we should immediately realise that we will need to add some form of Kleene star to the beginning of the clauses. Thus the first clause should be changed to $[\overline{2hrs}^*][2hrs]O_{O(fine)}(openCheckIn\&requestInfo)$. One should note that $\mathcal{CL}$ does not have a notion of time and that $2hrs$ is just a normal action thus the contract allows that action to appear more than once and it is because of this that we used $[\overline{2hrs}^*]$ instead of $[1^*]$ since this ensure that the obligation to open the check-in occurs only once after the first $2hrs$ action is observed. We will not explicitly prohibit the $2hrs$ action not to occur after the first occurrence since it will not affect the overall contract and in real life it is taken for granted. It would also be desired to restrict that the info is requested only when opening check-in and not anytime later thus we may change the clause to $[\overline{2hrs}^*][2hrs](O_{O(fine)}(openCheckIn\&requestInfo) \wedge [1][1^*]F(requestInfo))$.

With respect to clause two we will need to decide when the airline is obliged to reply with the flight information. In this case it would make sense to wait until the check-in desk is open and only after the check-in desk is open do we make the airline obligated to reply. We thus make clause two equal to

$$[\overline{openCheckIn}^*][openCheckIn \cdot requestInfo]O_{O(fine)}(replyInfo)$$

When analysing the contract we realise that an unreachable obligation has been detected. We will never be obliged to reply with the information. In this case we have a mistake, not just a case of removing the unreachable deontic notion. From clause two we are waiting for the request to arrive after the opening of the check-in, whereas from clause one we are obliging the request at the time when opening the check-in. This would not have been a problem since we could still be free to request for the information one step afterwards, however we have prohibited that thus ending in a situation where we miss the request and thus never obliged to reply. This can be changed by making clause two equal to

$$[\overline{openCheckIn}^{*}][openCheckIn\&requestInfo]O_{O(fine)}(replyInfo)$$

One should note that if the prohibition was not there, the obligation to reply with the information would not have been flagged as an unreachable state, however with the analysis tools available we could have still detected such a situation. If we requested the traces which would lead to us being obliged to reply with the information, we will realise that the request is not actually being obliged but it is only being permitted.

If we query to see traces that do not violate the contract and one of the actions is $closeCheckIn$ we will realise that this action may occur before even opening the check-in. This is obviously undesirable. Thus we will need to add the clause $[\overline{openCheckIn}*^{]}\, F(closeCheckIn) \wedge F(closeCheckIn)$. This could again be described as an obvious clause since we know the typical behaviour of how a check-in desk would work. However, the contract does not have this "universal" knowledge. This clause, even though it should be added to the contract written in $\mathcal{CL}$ it is not necessary to add to the contract written in textual format since it might be seen as overkill. This change to the contract however leads to a conflict. The check-in crew is obliged to close the desk 20 minutes before departure, however in the trace returned we realise that we are observing the $20mins$ action before the $2hrs$ action. Again we should "two hours before the flight" occurs before "20 minutes before the flight". We will thus add $[\overline{2hrs}^{*}]F(20mins)$. This clause will not be translated into the final textual representation of the contract and is only a clause to add the correct behaviour of these two actions.

We also realise that we will have a deontic directive which is not reachable. Clause five states that after the check-in is closed then the ground crew are obliged to send the Luggage Information:

$$[closeCheckIn]O_{O(fine)}(sendLuggageInfo)$$

The mistake is that we have not added $[\overline{closeCheckIn}^{*}]$ to the front without which this clause would only apply to the initial step and in the initial step it is prohibited to close the check-in since it has not been opened yet.

In order to improve on the speed of the analysis we may add information about mutually exclusive actions since when we make actions mutually exclusive we reduce the number of possible actions. We may add $openCheckIn\#closeCheckIn$ and also $2hrs\#20mins$.

The final contract would be:

1. $[\overline{2hrs}^{*}][2hrs](O_{O(fine)}(openCheckin\&requestInfo)\wedge$ $[1][1^{*}]F(requestInfo))$
2. $[\overline{openCheckin}^{*}][openCheckin\&requestInfo]O_{O(fine)}(replyInfo)$
3. $[\overline{openCheckin}^{*}][openCheckin][\overline{closeCheckin}^{*}]$ $[correctDetails\&luggageInLimit]\ O_{O(fine)}(BoardingCard)$
4. $[\overline{openCheckin}^{*}][openCheckin][\overline{closeCheckin}^{*}][correctDetails\&$ $\overline{luggageInLimit}]\ O_{O(fine)}(CollectPayment\&BoardingCard)$
5. $[1*][\overline{correctDetails}]F(BoardingCard)$
6. $[\overline{openCheckin}^{*}](F(BoardingCard)\wedge F(closeCheckin))$
7. $[\overline{20mins}*](F(closeChecking)\wedge[20mins]O(closeCheckin))$
8. $[\overline{closeCheckin}^{*}][closeCheckin]O_{O(fine)}(sendLuggageInfo)$
9. $[1^{*}][closeCheckin][1^{*}](F(BoardingCard)\wedge F(openCheckin))$

10. $[\overline{2hrs}]F(20mins)$

This could be represented in textual form as:

1. 2 hours before flight time, the ground crew are obliged to open the check-in desk and request from the airline the passenger information. After this they are forbidden from requesting the passenger information again.
2. The airline are obliged to reply to the passenger information request which occurred with the opening of the check-in desk with the passenger manifest.
3. While the check-in desk is open, if the details of the passport match the details on the ticket and the luggage is within the weight limit the ground crew are obliged to issue a boarding card.
4. While the check-in desk is open, if the details of the passport match the details on the ticket however the luggage is over the weight limit the ground crew are obliged to collect payment for the extra weight and issue a boarding card.
5. The ground crew are prohibited from issuing any boarding cards without inspecting that the details are correct beforehand.
6. The ground crew is prohibited from closing the check-in before 20 minutes before the flight at which time they are obliged to close the check-in desk.

7. Once the check-in desk is closed the ground crew are obliged to send the luggage information to the airline.
8. Once the check-in desk is closed the ground crew are forbidden from issuing any boarding cards or reopening the check-in desk again.

Once the contract is signed by both parties we can generate the monitor directly from the contract in order to ensure that the contract is adhered to between the two parties. Certain aspects of the contract, like for example that that the ground crew personnel check the details of the passengers cannot be observed by the enterprise service bus in most airports, however this might soon change when we have our biometric information encoded in the passport and we would be required a thumb scan or retinal scan.

Monitoring the System

In this case, we have two systems communicating with eachother. We have the airline system that has the information about the passengers and that will receive information about which passengers have checked in, and we have the check in system that logs the information about each traveller that is checking in and then sends the information to the airline system. In this case, using Larva in for monitoring would not be ideal since you would have two monitors on both systems and there is also the requirement that they are both written in Java.

In such scenario, there is typically an enterprise service bus (ESB) residing seamlessly between the two systems. The enterprise service bus will aid systems using different protocols to communicate with eachother. Lets consider that the airline system will communicate through web services wheras the check-in desks will get the passenger information using JDBC (getting information from a database) and use JMS (Java Messenging Service) in order to send the check in information and notify the opening and closing of the desk. Thus when the enterprise service bus receives the jms message that the check in desk has been opened, it will translate the message into a web service request and retrieve the passenger information which is then translated into a database insert query that is executed on the check in database. The JMS messages sent by the check in desks are aggregated at the ESB and sent as a web service request once the close check-in desk message is received.

As one can note, this ESB allows for a highly decoupled design, and allows the same airline and check-in system to interact with other systems even if they provide different interfaces.

The ESB, does not have access to the internal code of niether of the systems, however, since it is observing all the messages passing to and from the airline and check-in system it can quite effectively monitor the behaviour. Thus it would be ideal to have a monitoring module inside the ESB which generates the monitor directly from the $\mathcal{CL}$ contract. One would also need to define the actions, which in this case would be the events of receiving and sending of messages through the ESB.

By placing the monitor on the ESB, we can now ensure for example that the check-in desks open and close on time and do not check-in any luggage once closed. We can also ensure that payment is collected when luggage that is checked-in is overweight. Furthermore we can ensure that the airline system responds with the correct passenger details. Unfortunately, in this case, we cannot monitor the entire contract since the system cannot be sure that the check-in crew has ensured that the details on the passport match the details on the ticket. However it is clear that with this method, we could ensure that most of the contract is adhered to.

8.5 Conclusion

In this chapter we first went through the implementation of the tool, where we saw how the main part of the algorithm was implemented. This was then followed by a discussion of improvements to the basic implementation which will allow us to analyse even larger contracts. This is then followed by the case study, where we showecased our tools and techniques in a realistic scenario.

In the case study we have shown how one would go about drafting a contract. Then using our tool we can analyse the contract for conflicts and reachability analysis from which we can then fix errors in the contracts. We also shown how quering and simulating the contract would also help in identifying and fixing errors.

Chapter 9

Conclusion

In this work we have successfully automated conflict analysis of a contract written in $\mathcal{CL}$. In order to automate conflict analysis we were required to generate an automaton that accepts all and only traces which satisfy the contract. We proved that the automaton generated will in fact accept all and only traces satisfying the contract and also proved that the conflict analysis is correct. Once we have generated the automaton we have also shown how we could analyse the contract further. We have shown how we could check a contract for superfluous clauses, query the contract, simulate the contract and also generate the monitor. Using the theory developed in this work, we have also implemented a tool that will automatically analyse contracts for conflicts.

In this work we also portrayed $\mathcal{CL}$ as a specification language and as a modelling language. We compared $\mathcal{CL}$ with other methods of specification. Furthermore, by making use of $\mathcal{CL}$ in case studies we have shown that the $\mathcal{CL}$ language is mature enough to be used in real life situations. We have also shown how attractive it is to use such a deontic logic. We have also shown how we could aid in the drafting of contracts using the tools developed in this work together with the other possibilities mentioned in this work .

9.1 Future Work

The future work on this subject may be split in two. First of all, there is the work directly concerned with the refinement of the $\mathcal{CL}$ language. Secondly, there is work concerned with the development of techniques in order to automatically analyse contracts. With regards to the language there are a number of possibilities that we would like to explore further:

- Right now, the actions are left underspecified and we add our own definition to the action. However, in these contracts we do not have the notion of signatories of the contracts. So if we have the formula

$O(pay)$ the contract in itself has no knowledge of who is to do the payment and to whom. This issue has been avoided by encoding this information inside the action itself. Thus for example, action “pay” is defined to mean that the client paid a certain amount to the service provider. However we believe that it would be useful that the deontic notions would have information about who is doing what, for example an obligation will become a triple rather than just a single action where it will have the information about who is being obliged and on whom the action is being made. This will open up a number of new possibilities with respect to possible analysis that we may perform on the contracts.

- The semantics of $\mathcal{CL}$ do not differentiate between actions. However we believe that actions may have different features or qualities and also different defaults. In contracts, actions are usually either permitted or forbidden by default. Typically, actions that are in the control of the signatories are forbidden by default whereas actions that cannot be controlled (i.e. actions being done by non signatories) are usually permitted by default. As one might expect, there are many exceptions to this rule. However, we believe that we should be able to specify the default behaviour of actions. Thus we have the default deontic notions which apply to the action and then we have a higher level where we specify how these defaults are overidden.

- Another point that would be interesting to investigate, that is slightly based on the previous item, is that we have a layered contract. We first give the most basic clauses and then we refine these general clauses in higher and higher layers. This is usually the way contracts are written, where first we give the general rules example “Clients are forbidden to make any sort of damage to the property”. Then we have exceptions “in case of fire clients are permitted to break glass”. In this work, the situation would be marked as a conflict and in order to fix this we have to explicitly write that we are forbidden to make any sort of damage to the property except in the case of fire.

- We need to give more thought to the Exclusive Or operator as discussed in Section 3.3. The current syntax and semantics allow us to define clauses which when certain traces are followed we will have to satisfy clauses which we and the original authors of $\mathcal{CL}$ believe should not be allowed. Consider the clause $\top \oplus O(a)$. In this case we are not allowed to satisfy the obligation to perform a thus effectively we have a negation of an obligation which in [55] is deemed as undesirable.

- We need to make changes in the definition of the prohibition of choice as defined in Section 3.4 or explicitely make this issue clear. The current semantics define $F_C(\alpha + \alpha')$ as $F_C(\alpha) \wedge F_C(\alpha')$. If the length of α and α' is identical, then there is no problem, since if both are observed (thus violating the clause) the reparation will be enacted once at the same instance. However, if they are not of the same length and the clause is violated by both α and α' we will have to satisfy the reparation twice. We believe that if both ways of violating the prohibition occurs, the reparation is only applied once. If the original behaviour is required, that the reparation is satisfied twice, one can still define this using the conjunction of the prohibition of both actions.

When it comes to automation the following are interesting prospects on which we shall continue to work:

- In this works we have suggested a number of different analysis which could be done using the automaton generated from a $\mathcal{CL}$ contract. Because of time restrictions not all of these have been implemented in the tool. It would be an interesting project to implement more of these analysis techniques and possibly investigate further possibilities.

- Full model checking of the trace semantics. Once we have generated an automaton from the contract written in $\mathcal{CL}$ it is now possible using automata theoretic approaches to model check. Thus given a model, we may verify that it will always satisfy the contract. The spirit of the approach would be to negate the automaton generated from the $\mathcal{CL}$ contract and then obtain the product of this new automata with the model and this should result in an empty automaton. This is, however very inefficient and so it would be interesting to imvestigate this further.

- Develop a normative automata approach to $\mathcal{CL}$ where we give the semantics of $\mathcal{CL}$ using an automata rather than using an extension in the μ-Calculus. This will not only give a clearer semantics but we shall also have a start towards model checking of the full $\mathcal{CL}$. There is already some work on this but nothing concrete yet.

- Once we extend the $\mathcal{CL}$ with the notion of actors and possibly the layering of contracts we would have a number of new tools possible. We would have to extend again the conflict analysis method, monitoring and even the superfluous clause analysis.

- An interesting project would be to implement the module described in the Preface to fit inside an enterprise service bus in order to be able to apply $\mathcal{CL}$ in live projects rather than just theoretical case studies. We see an example of the possibilities of such a module in the case study of Chapter 8.

9.2 Concluding remarks

As we have discussed in the introduction, the industry is moving towards a service oriented architecture where now the code is distributed not only across machines in the same company but also across the organisational's borders as well. Because of this dependence on other organisations, companies make use of contracts in order to safeguard against any possible adverse behaviours performed by the other companies. This has lead to the need of a way to be able to describe such contracts in a formal manner in order to give the possibility for more automation and securty. $\mathcal{CL}$ is a language designed to fill this void.

$\mathcal{CL}$ as a language is still quite young; however, it still can be applied to current real life situations. We have seen in this work $\mathcal{CL}$ being applied to the CoCoME case study and also to an airline case study. Both in these situations $\mathcal{CL}$ performed well allowing us to elegantly describe the requirements of the parties involved and also how exceptional behaviour is handled.

Also, by providing tools that automatically analyse contracts we add value to $\mathcal{CL}$. We have augmented the trace semantics of $\mathcal{CL}$ with deontic information in order to be able to find conflicts in contracts. We then also defined a construction to generate an automaton from $\mathcal{CL}$ and automatically check if the contract is conflict free or not. Making use of automatic contract analysis allows us to draft sound contracts in a much shorter time since we can automatically identify undesirable situations. In this work we focused on identifying conflict free contracts, however we also show how we can make further analysis using the techniques described here. Furthermore, in the airline case study we have shown how one would go about drafting a contract and how usefull the tools developed in this masters are to identify possible problems in the contract.

As discussed in the previous section, there is still more work to be done on $\mathcal{CL}$ however, it is already in a state which could be applied and used in real projects. Furthermore, if $\mathcal{CL}$ has also a vast toolset to enable users to both create, analyse and deploy contracts, it would make $\mathcal{CL}$ a very suitable solution for contracting, especially in a service oriented architecture.

Appendix A

Correctness of ts3

Theorem A.1. *An empty trace can never violate a contract:*

$$\forall C, len(\sigma) = len(\sigma_d) = 0 \Rightarrow \sigma, \sigma_d \vDash_f C$$

Proof. We shall prove this by structural induction on the formula. The base cases are the following and result directly from the definition:

$$\begin{array}{rrl} len(\sigma) = len(\sigma_d) = 0 \Rightarrow & & \\ & \sigma, \sigma_d \vDash_f & \top \\ & \sigma, \sigma_d \vDash_f & [\alpha_\&]C \\ & \sigma, \sigma_d \vDash_f & O_C(\alpha_\&) \\ & \sigma, \sigma_d \vDash_f & F_C(\alpha_\&) \\ & \sigma, \sigma_d \vDash_f & P(\alpha_\&) \\ & \sigma, \sigma_d \vDash_f & C_1 \oplus C_2 \end{array}$$

The rest of the cases are defined using these base cases.

Case $C_1 \wedge C_2$

$$\begin{array}{ll} & len(\sigma) = len(\sigma_d) = 0 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_1 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f C_2 \\ \Rightarrow & \{\text{conjunction introduction}\} \\ & \sigma, \sigma_d \vDash_f C_1 \text{ and } \sigma, \sigma_d \vDash_f C_2 \\ \Rightarrow & \{\text{definition}\} \\ & \sigma, \sigma_d \vDash_f C_1 \wedge C_2 \end{array}$$

Case $[\beta;\beta']C$

$$\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\; & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f C_1 \\
\Rightarrow\; & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f C_2 \\
\Rightarrow\; & \{\text{let } C_1 = [\beta]C_2 \text{ and } C_2 = [\beta']C\} \\
& \sigma, \sigma_d \vDash_f [\beta][\beta']C \\
\Rightarrow\; & \{\text{definition}\} \\
& \sigma, \sigma_d \vDash_f [\beta;\beta']C
\end{aligned}$$

Case $[\beta + \beta']C$

$$\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\; & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f C_1 \\
\Rightarrow\; & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f C_2 \\
\Rightarrow\; & \{\text{conjunction introduction}\} \\
& \sigma, \sigma_d \vDash_f C_1 \wedge C_2 \\
\Rightarrow\; & \{\text{let } C_1 = [\beta]C \text{ and } C_2 = [\beta']C\} \\
& \sigma, \sigma_d \vDash_f [\beta]C \wedge [\beta']C \\
\Rightarrow\; & \{\text{definition}\} \\
& \sigma, \sigma_d \vDash_f [\beta + \beta']C
\end{aligned}$$

Case $O_C(\alpha;\alpha')$

$$\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\; & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\Rightarrow\; & \{\text{definition}\} \\
& \sigma, \sigma_d \vDash_f O_C(\alpha;\alpha')
\end{aligned}$$

Case $O_C(\alpha + \alpha')$

$$
\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f O_\perp(\alpha) \\
\Rightarrow\ & \{\text{disjunction introduction}\} \\
& \sigma, \sigma_d \vDash_f O_\perp(\alpha) \text{ or } \sigma, \sigma_d \vDash_f O_\perp(\alpha') \text{ or } (\sigma_d(0) = (O\alpha \text{ or } O\alpha') \text{ and} \\
& \qquad \sigma, \emptyset; \sigma_d(1..) \vDash_f [\overline{\alpha + \alpha'}]C) \\
\Rightarrow\ & \{\text{definiton}\} \\
& \sigma, \sigma_d \vDash_f O_C(\alpha + \alpha')
\end{aligned}
$$

Case $F_C(\alpha; \alpha')$

$$
\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f F_\perp(\alpha) \text{ or } \sigma, \sigma_d \vDash_f [\alpha]F_C(\alpha') \\
\Rightarrow\ & \{\text{definiton}\} \\
& \sigma, \sigma_d \vDash_f F_C(\alpha; \alpha')
\end{aligned}
$$

Case $F_C(\alpha + \alpha')$

$$
\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f F_C(\alpha) \wedge F_C(\alpha') \\
\Rightarrow\ & \{\text{definiton}\} \\
& \sigma, \sigma_d \vDash_f F_C(\alpha + \alpha')
\end{aligned}
$$

Case $P(\alpha; \alpha')$

$$
\begin{aligned}
& len(\sigma) = len(\sigma_d) = 0 \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash_f P(\alpha) \wedge [\alpha]P(\alpha') \\
\Rightarrow\ & \{\text{definiton}\} \\
& \sigma, \sigma_d \vDash_f P(\alpha; \alpha')
\end{aligned}
$$

Case $P(\alpha + \alpha')$

$$\begin{array}{ll} & len(\sigma) = len(\sigma_d) = 0 \\ \Rightarrow & \{\text{inductive hypothesis}\} \\ & \sigma, \sigma_d \vDash_f P(\alpha) \wedge P(\alpha') \\ \Rightarrow & \{\text{definiton}\} \\ & \sigma, \sigma_d \vDash_f P(\alpha + \alpha') \end{array}$$

□

Proposition A.2. *Any finite trace using the semantics of ts3 will satisfy the trivial contract* $\top$

$$\forall \sigma \ \sigma, \sigma_\emptyset(0..len(\sigma)) \vDash_f \top$$

Proof. We shall prove this by induction on the length of σ.

Base Case n=0 from Theorem A.1

Inductive Hypothesis n=k

$$\forall \sigma \ \sigma(0..k), \sigma_\emptyset(0..k) \vDash_f \top$$

Inductive Case n=k+1

$$\begin{array}{ll} & \sigma(0..k+1), \sigma_d(0..k+1) \vDash_f \top \\ \Rightarrow & \{\text{definition}\} \\ & \sigma(1..k+1), \sigma_d(1..k+1) \vDash_f \top \\ & \{\text{the definition does not consider the elements in the trace,} \\ \Rightarrow & \text{only the length. Thus the definition cannot differentiate} \\ & \text{between } \sigma(1..k+1) \text{ and } \sigma(0..k)\} \\ & \sigma(0..k), \sigma_d(0..k) \vDash_f \top \\ \Rightarrow & \{\text{Inductive hypothesis}\} \end{array}$$

□

Proposition A.3. *Any infinite trace using the semantics of ts2 will satisfy the trivial contract* $\top$

$$\forall \sigma \ \sigma, \sigma_\emptyset \vDash^d_\infty \top$$

Proof. The definition of $\sigma, \sigma_d \vDash^d_\infty \top$ is recursive where we chop off the first element from σ and σ_d and then check that $\sigma(1..), \sigma_d(1..) \vDash^d_\infty \top$. The only check on the first element is that $\sigma_d(0) = \emptyset$ thus $\sigma(0)$ is free to be any set of possible actions. Furthermore, we know that $\sigma_d(0) = \emptyset$ since by definition, $\sigma_\emptyset$ is the infinite sequence of $\emptyset$ and thus, this proposition is true. □

Proposition A.4. *For any infinite trace which satisfies the trivial contract* $\top$*, the deontic trace will be empty, as defined before,* $\sigma_d = \sigma_\emptyset$.

$$\sigma, \sigma_d \vDash \top \Rightarrow \sigma_d = \sigma_\emptyset$$

Proof. The definition of $\sigma, \sigma_d \vDash^d_\infty \top$ is recursive where we chop off the first element from σ and σ_d and then check that $\sigma(1..), \sigma_d(1..) \vDash^d_\infty \top$. By definition we require that $\sigma_d(0) = \emptyset$ and then we keep on recursively checking the rest of the trace recursively, removing the first element and checking that $\sigma_d(0) = \emptyset$ and thus for $\sigma, \sigma_d \vDash^d_\infty \top$, $\sigma_d = \sigma_\emptyset$ □

Theorem A.5. *A trace* σ_d *is equal to the combination of* σ'_d *and* σ''_d *(*$\sigma_d = \sigma'_d \cup \sigma''_d$*) iff any subtrace of* σ_d *is equal to the combination of the subtraces of* σ'_d *and* σ''_d *given that they are of the same length.*

$$\sigma_d = \sigma'_d \cup \sigma''_d \Leftrightarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$$

Proof. We shall split this proof in two cases, proving the two directions of the implication separately.

Case $\sigma_d = \sigma'_d \cup \sigma''_d \Rightarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$. The definition of $\sigma_d = \sigma'_d \cup \sigma''_d$ is $\sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n)$. We shall use induction on n in order to prove this case.

Base Case n=0

$$\begin{aligned} & \sigma_d = \sigma'_d \cup \sigma''_d \\ \Rightarrow\ & \{\text{definition}\} \\ & \sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n) \\ \Rightarrow\ & \{\text{n=0, so we chop } \sigma_d \text{ and take only required sequence}\} \\ & \sigma_d(0) = \sigma'_d(0) \cup \sigma''_d(0) \end{aligned}$$

Inductive Hypothesis n=k

$$\sigma_d(k) = \sigma'_d(0..k) \cup \sigma''_d(0..k)$$

Inductive Case n=k+1

$$
\begin{array}{ll}
 & \sigma_d = \sigma'_d \cup \sigma''_d \\
\Rightarrow & \{\text{definition}\} \\
 & \sigma_d = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(n) \cup \sigma''_d(n) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & \sigma_d(k) = \sigma'_d(0..k) \cup \sigma''_d(0..k) \\
\Rightarrow & \{\text{definition}\} \\
 & \sigma_d(k) = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \sigma'_d(k) \cup \sigma''_d(k) \\
\Rightarrow & \{\text{by definition we may add the next element by adding to sequence}\} \\
 & \sigma_d(k+1) = \sigma'_d(0) \cup \sigma''_d(0), \sigma'_d(1) \cup \sigma''_d(1) \ldots \\
 & \qquad \sigma'_d(k) \cup \sigma''_d(k), \sigma'_d(k+1) \cup \sigma''_d(k+1) \\
\Rightarrow & \{\text{definition}\} \\
 & \sigma_d(k+1) = \sigma'_d(k+1) \cup \sigma''_d(k+1)
\end{array}
$$

Case $\sigma_d = \sigma'_d \cup \sigma''_d \Leftarrow \forall n, \sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$

$\sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n)$ holds for all N thus if $n = len(\sigma_d)$ then $\sigma_d(0..n) = \sigma_d, \sigma'_d(0..n) = \sigma'_d$ and $\sigma''_d(0..n) = \sigma''_d$

□

Lemma A.6. *If the infinite traces σ, σ_d satisfy a contract C, then any finite prefix of these traces will not violate the contract.*

$$\sigma, \sigma_d \vDash^d_\infty C \Rightarrow \forall n : N, \sigma(0..n), \sigma_d(0..n) \vDash_f C$$

Proof. $\sigma, \sigma_d \vDash^d_\infty C$ means that the traces σ, σ_d do not violate the contract C. So if the entire infinite trace does not violate the contract then even any finite prefix would not violate the contract, or at least would not have violated the contract yet. This is the defenition of the $\vDash_f$ relationship and so we may conclude that for any finite prefix of the infinite traces that satisfy a contract C will not violate the contract.

We shall use structural induction in order to prove this formally. First of all we have the rule that $\sigma, \sigma_d \nvDash_f C$ if $len(\sigma) \neq len(\sigma_d)$. In our case the lengths of σ and σ_d are identical by definition.

Case $\top$

$$
\begin{array}{ll}
 & \sigma, \sigma_d \models^d_\infty \top \\
\Rightarrow & \{\text{Proposition 5.1.4}\} \\
 & \sigma_d = \sigma_\emptyset \\
\Rightarrow & \{\sigma_d = \sigma_\emptyset \text{ and Proposition 5.1.2}\} \\
 & \forall n : N, \sigma(0..n), \sigma_d(0..n) \models_f \top
\end{array}
$$

Case $C_1 \wedge C_2$

$$
\begin{array}{ll}
 & \sigma, \sigma_d \models^d_\infty C_1 \wedge C_2 \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma'_d \models^d_\infty C_1 \text{ and } \sigma, \sigma''_d \models^d_\infty C_2 \text{ and } \sigma_d = \sigma'_d \cup \sigma''_d \\
\Rightarrow & \{\text{inductive hypothesis and Theorem 5.1.5}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n)' \models_f C_1 \text{ and } \forall n : N\ \sigma(0..n), \sigma_d(0..n)'' \models_f C_2 \\
 & \qquad \text{and } \forall n : N\ \sigma_d(0..n) = \sigma_d(0..n)' \cup \sigma_d(0..n)'' \\
\Rightarrow & \{\text{associativity of } \forall \text{ over conjunction}\} \\
 & \forall n : N\ (\sigma(0..n), \sigma_d(0..n)' \models_f C_1 \text{ and } \sigma(0..n), \sigma_d(0..n)'' \models_f C_2 \\
 & \qquad \text{and } \sigma_d(0..n) = \sigma_d(0..n)' \cup \sigma_d(0..n)'') \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f C_1 \wedge C_2
\end{array}
$$

Case $C_1 \oplus C_2$

$$
\begin{array}{ll}
 & \sigma, \sigma_d \models^d_\infty C_1 \oplus C_2 \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & (\sigma, \sigma_d \models^d_\infty C_1 \text{ and } \sigma, \sigma_d \not\models^d_\infty C_2) \text{ or } (\sigma, \sigma_d \models^d_\infty C_2 \text{ and } \sigma, \sigma_d \not\models^d_\infty C_1) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & (\forall n : N\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_1 \text{ and } \forall n : N\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_2) \text{ or} \\
 & \qquad (\forall n : N\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_2 \text{ and } \forall n : N\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_1) \\
\Rightarrow & \{\text{associativity of } \forall \text{ over disjunction}\} \\
 & \forall n : N\ (\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_1 \text{ and } \sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_2) \text{ or} \\
 & \qquad (\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_2 \text{ and } \sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_1) \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f C_1 \oplus C_2
\end{array}
$$

Case $[\alpha_{\&}]C$

$\sigma, \sigma_d \models^d_\infty [\alpha_{\&}]C$

$\Rightarrow$ {definition of ts2}

$\sigma_d(0) = \emptyset$ and $(\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty C,$ or $\alpha_{\&} \nsubseteq \sigma(0)))$

$\Rightarrow$ {inductive hypothesis}

$\sigma_d(0) = \emptyset$ and

$(\alpha_{\&} \subseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f C,$ or $\alpha_{\&} \nsubseteq \sigma(0)))$

$\Rightarrow$ {an empty trace cannot violate a contract(Theorem 5.1.1)}

$len(\sigma) = 0$ or $\sigma_d(0) = \emptyset$ and

$(\alpha_{\&} \subseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f C,$ or $\alpha_{\&} \nsubseteq \sigma(0)))$

$\Rightarrow$ {No free variable n, thus can move $\forall n : N$ outside}

$\forall n : N\ len(\sigma) = 0$ or $\sigma_d(0) = \emptyset$ and

$(\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..n), \sigma_d(1..n) \models_f C,$ or $\alpha_{\&} \nsubseteq \sigma(0)))$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\alpha_{\&}]C$

Case $[\beta; \beta']C$

$\sigma, \sigma_d \models^d_\infty [\beta; \beta']C$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty [\beta][\beta']C$

$\Rightarrow$ {inductive hypothesis}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta][\beta']C$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta; \beta']C$

Case $[\beta + \beta']C$

$\sigma, \sigma_d \models^d_\infty [\beta + \beta']C$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty [\beta]C \wedge [\beta']C$

$\Rightarrow$ {inductive hypothesis}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta]C \wedge [\beta']C$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta + \beta']C$

Case $[\beta^*]C$

$\sigma, \sigma_d \vDash^d_\infty [\beta^*]C$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \vDash^d_\infty C \wedge [\beta; \beta^*]C$

$\Rightarrow$ {inductive hypothesis}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f C \wedge [\beta; \beta^*]C$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f [\beta^*]C$

Case $O_C(\alpha_\&)$

$\sigma, \sigma_d \vDash^d_\infty O_C(\alpha_\&)$

$\Rightarrow$ {definition of ts2}

$\sigma_d(0) = O\alpha$ and

$((\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash^d_\infty \top)$ or $\sigma(1..), \sigma_d(1..) \vDash^d_\infty C)$

$\Rightarrow$ {inductive hypothesis}

$\sigma_d(0) = O\alpha$ and $(\alpha_\& \subseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \vDash_f \top)$ or

$(\forall n : N\ \sigma(1..n), \sigma_d(1..n) \vDash_f C)$

$\Rightarrow$ {No free variable n, thus can move $\forall n : N$ outside}

$\forall n : N\ \sigma_d(0) = O\alpha$ and $(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..n), \sigma_d(1..n) \vDash_f \top)$ or

$(\sigma(1..n), \sigma_d(1..n) \vDash_f C)$

$\Rightarrow$ {an empty trace cannot violate a contract(Theorem A.1)}

$\forall n : N\ len(\sigma) = 0$ or $\sigma_d(0) = O\alpha$ and

$((\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top)$ or $\sigma(1..), \sigma_d(1..) \vDash_f C)$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha_\&)$

Case $O_C(\alpha; \alpha')$

$\sigma, \sigma_d \vDash^d_\infty O_C(\alpha; \alpha')$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \vDash^d_\infty O_C(\alpha) \wedge [\alpha]O_C(\alpha')$

$\Rightarrow$ {inductive hypothesis}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha')$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha; \alpha')$

Case $O_C(\alpha + \alpha')$

$\sigma, \sigma_d \models^d_\infty O_C(\alpha + \alpha')$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty O_\perp(\alpha)$ or $\sigma, \sigma_d \models^d_\infty O_\perp(\alpha')$ or

$(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma, \emptyset; \sigma_d(1..) \models^d_\infty [\overline{\alpha + \alpha'}]C$

$\Rightarrow$ {inductive hypothesis}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha)$ or $\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha')$ or

$(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\forall n : N\ \sigma(0..n), \emptyset; \sigma_d(1..n) \models_f [\overline{\alpha + \alpha'}]C$

$\Rightarrow$ {No free variable n, thus can move $\forall n : N$ outside}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha)$ or $\sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha')$ or

$(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma(0..n), \emptyset; \sigma_d(1..n) \models_f [\overline{\alpha + \alpha'}]C$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f O_C(\alpha + \alpha')$

Case $F_C(\alpha_\&)$

$\sigma, \sigma_d \models^d_\infty F_C(\alpha_\&)$

$\Rightarrow$ {definition of ts2}

$\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty \top$

or $(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty C))$

$\Rightarrow$ {inductive hypothesis}

$\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f \top)$

or $(\alpha_\& \subseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f C))$

$\Rightarrow$ {an empty trace cannot violate a contract(Theorem A.1)}

$len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and

$\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f \top)$

or $(\alpha_\& \subseteq \sigma(0)$ and $\forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f C))$

$\Rightarrow$ {No free variable n, thus can move $\forall n : N$ outside}

$\forall n : N\ len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and

$((\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models_f \top)$ or

$(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models_f C))$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f F_C(\alpha_\&)$

Case $F_C(\alpha;\alpha')$

$$
\begin{aligned}
& \sigma, \sigma_d \vDash^d_\infty F_C(\alpha;\alpha') \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma_d(0) = F\alpha \text{ and } (\sigma, \sigma_d \vDash^d_\infty F_\bot(\alpha) \text{ or } \sigma, \sigma_d \vDash^d_\infty [\alpha]F_C(\alpha')) \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma_d(0) = F\alpha \text{ and } (\forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f F_\bot(\alpha) \text{ or} \\
& \qquad \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f [\alpha]F_C(\alpha')) \\
\Rightarrow\ & \{\text{No free variable n, thus can move } \forall n : N \text{ outside}\} \\
& \forall n : N\ \sigma_d(0) = F\alpha \text{ and } (\sigma(0..n), \sigma_d(0..n) \vDash_f F_\bot(\alpha) \text{ or} \\
& \qquad \sigma(0..n), \sigma_d(0..n) \vDash_f [\alpha]F_C(\alpha')) \\
\Rightarrow\ & \{\text{definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f F_C(\alpha;\alpha')
\end{aligned}
$$

Case $F_C(\alpha+\alpha')$

$$
\begin{aligned}
& \sigma, \sigma_d \vDash^d_\infty F_C(\alpha+\alpha') \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma, \sigma_d \vDash^d_\infty F_C(\alpha) \wedge F_C(\alpha') \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f F_C(\alpha) \wedge F_C(\alpha') \\
\Rightarrow\ & \{\text{definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f F_C(\alpha+\alpha')
\end{aligned}
$$

Case $P(\alpha_\&)$

$$
\begin{aligned}
& \sigma, \sigma_d \vDash^d_\infty P(\alpha_\&) \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma_d(0) = P\alpha \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma_d(0) = P\alpha \text{ and } \forall n : N\ \sigma(1..n), \sigma_d(1..n) \vDash_f \top \\
\Rightarrow\ & \{\text{an empty trace cannot violate a contract(Theorem A.1)}\} \\
& len(\sigma) = 0 \text{ or } \sigma_d(0) = P\alpha \text{ and } \sigma(1..n), \sigma_d(1..n) \vDash_f \top \\
\Rightarrow\ & \{\text{No free variable n, thus can move } \forall n : N \text{ outside}\} \\
& \forall n : N\ len(\sigma) = 0 \text{ or } \sigma_d(0) = P\alpha \text{ and } \sigma(1..n), \sigma_d(1..n) \vDash_f \top \\
\Rightarrow\ & \{\text{definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f P(\alpha_\&)
\end{aligned}
$$

Case $P(\alpha;\alpha')$

$$\begin{aligned}
&\sigma,\sigma_d \vDash^d_\infty P(\alpha;\alpha') \\
\Rightarrow\ &\{\text{definition of ts2}\} \\
&\sigma,\sigma_d \vDash^d_\infty P(\alpha) \wedge [\alpha]P(\alpha') \\
\Rightarrow\ &\{\text{inductive hypothesis}\} \\
&\forall n : N\ \sigma(0..n),\sigma_d(0..n) \vDash_f P(\alpha) \wedge [\alpha]P(\alpha') \\
\Rightarrow\ &\{\text{definition of ts3}\} \\
&\forall n : N\ \sigma(0..n),\sigma_d(0..n) \vDash_f P(\alpha;\alpha')
\end{aligned}$$

Case $P_C(\alpha+\alpha')$

$$\begin{aligned}
&\sigma,\sigma_d \vDash^d_\infty P(\alpha+\alpha') \\
\Rightarrow\ &\{\text{definition of ts2}\} \\
&\sigma,\sigma_d \vDash^d_\infty P(\alpha) \wedge P(\alpha') \\
\Rightarrow\ &\{\text{inductive hypothesis}\} \\
&\forall n : N\ \sigma(0..n),\sigma_d(0..n) \vDash_f P(\alpha) \wedge P(\alpha') \\
\Rightarrow\ &\{\text{definition of ts3}\} \\
&\forall n : N\ \sigma(0..n),\sigma_d(0..n) \vDash_f P(\alpha+\alpha')
\end{aligned}$$

□ □

Lemma A.7. *If all the finite prefixes of an infinite trace do not violate a contract C then the infinite traces satisfy the contract.*

$$\sigma,\sigma_d \vDash^d_\infty C \Leftarrow \forall n : N\sigma(0..n),\sigma_d(0..n) \vDash_f C$$

Proof. We shall use structural induction to show that if for any n, $n : N\sigma(0..n),\sigma_d(0..n) \vDash_f C$ then for the equivalent infinite trace $\sigma,\sigma_d \vDash^d_\infty C$

Case $\top$

$$\begin{aligned}
&\forall n : N,\sigma(0..n),\sigma_d(0..n) \vDash_f \top \\
\Rightarrow\ &\{\text{Proposition A.3}\} \\
&\sigma,\sigma_d \vDash^d_\infty \top
\end{aligned}$$

Case $C_1 \wedge C_2$

$\forall n : N, \sigma(0..n), \sigma_d(0..n) \models_f C_1 \wedge C_2$

$\Rightarrow$ {definition of ts3}

$\forall n : N, (\sigma(0..n), \sigma'_d(0..n) \models_f C_1$ and $\sigma(0..n), \sigma''_d(0..n) \models_f C_2$ and
$\sigma_d(0..n) = \sigma'_d(0..n) \cup \sigma''_d(0..n))$

$\Rightarrow$ {inductive hypothesis and Theorem A.5}

$\sigma, \sigma'_d \models^d_\infty C_1$ and $\sigma, \sigma''_d \models^d_\infty C_2$ and $\sigma_d = \sigma'_d \cup \sigma''_d$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty C_1 \wedge C_2$

Case $C_1 \oplus C_2$

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f C_1 \oplus C_2$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ (\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_1$ and $\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_2)$ or
$(\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_2$ and $\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_1)$

$\Rightarrow$ {associativity of $\forall$ over disjunction}

$(\forall n : N\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_1$ and $\forall n : N\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_2)$ or
$(\forall n : N\sigma(0..n), \sigma_d(0..n) \models^d_\infty C_2$ and $\forall n : N\sigma(0..n), \sigma_d(0..n) \not\models^d_\infty C_1)$

$\Rightarrow$ {inductive hypothesis}

$(\sigma, \sigma_d \models^d_\infty C_1$ and $\sigma, \sigma_d \not\models^d_\infty C_2)$ or $(\sigma, \sigma_d \models^d_\infty C_2$ and $\sigma, \sigma_d \not\models^d_\infty C_1)$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty C_1 \oplus C_2$

Case $[\alpha_{\&}]C$

$$
\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\alpha_{\&}]C \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & len(\sigma) = 0 \text{ or } (\sigma_d(0) = \emptyset \text{ and } (\alpha_{\&} \subseteq \sigma(0) \text{ and} \\
 & \qquad \forall n : N\ \sigma(1..n), \sigma_d(1..n) \models_f C, \text{ or } \alpha_{\&} \not\subseteq \sigma(0)))) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & len(\sigma) = 0 \text{ or } (\sigma_d(0) = \emptyset \text{ and} \\
 & \qquad (\alpha_{\&} \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \models^d_\infty C, \text{ or } \alpha_{\&} \not\subseteq \sigma(0)))) \\
\Rightarrow & \{\sigma \text{ is infinite so } len(\sigma) \neq 0\} \\
 & \sigma_d(0) = \emptyset \text{ and } (\alpha_{\&} \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \models^d_\infty C, \text{ or } \alpha_{\&} \not\subseteq \sigma(0))) \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \models^d_\infty [\alpha_{\&}]C
\end{array}
$$

Case $[\beta; \beta']C$

$$
\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta; \beta']C \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta][\beta']C \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & \sigma, \sigma_d \models^d_\infty [\beta][\beta']C \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \models^d_\infty [\beta; \beta']C
\end{array}
$$

Case $[\beta + \beta']C$

$$
\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta + \beta']C \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f [\beta]C \wedge [\beta']C \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & \sigma, \sigma_d \models^d_\infty [\beta]C \wedge [\beta']C \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \models^d_\infty [\beta + \beta']C
\end{array}
$$

Case $[\beta^*]C$

$$\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f C \wedge [\beta][\beta^*]C \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & \sigma, \sigma_d \vDash^d_\infty C \wedge [\beta][\beta^*]C \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty
\end{array}$$

Case $O_C(\alpha_\&)$

$$\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha_\&) \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
 & \quad ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..n), \sigma_d(1..n) \vDash_f \top) \text{ or } \sigma(1..n), \sigma_d(1..n) \vDash_f C) \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
 & \quad ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\Rightarrow & \{\sigma \text{ is infinite so } len(\sigma) \neq 0\} \\
 & \sigma_d(0) = O\alpha \text{ and} \\
 & \quad ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash^d_\infty \top) \text{ or } \sigma(1..), \sigma_d(1..) \vDash^d_\infty C) \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty O_C(\alpha_\&)
\end{array}$$

Case $O_C(\alpha; \alpha')$

$$\begin{array}{ll}
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha; \alpha') \\
\Rightarrow & \{\text{definition of ts3}\} \\
 & \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
 & \sigma, \sigma_d \vDash^d_\infty O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\Rightarrow & \{\text{definition of ts2}\} \\
 & \sigma, \sigma_d \vDash^d_\infty O_C(\alpha; \alpha')
\end{array}$$

Case $O_C(\alpha + \alpha')$

$\forall n : N \ \sigma(0..n), \sigma_d(0..n) \models_f O_C(\alpha + \alpha')$

$\Rightarrow$ {definition of ts3}

$\forall n : N \ \sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha)$ or $\sigma(0..n), \sigma_d(0..n) \models_f O_\perp(\alpha')$ or

$(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma(0..n), \emptyset; \sigma_d(1..n) \models_f [\overline{\alpha + \alpha'}]C)$

$\Rightarrow$ {inductive hypothesis}

$\sigma, \sigma_d \models^d_\infty O_\perp(\alpha)$ or $\sigma, \sigma_d \models^d_\infty O_\perp(\alpha')$ or

$(\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma, \emptyset; \sigma_d(1..) \models^d_\infty [\overline{\alpha + \alpha'}]C)$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty O_C(\alpha + \alpha')$

Case $F_C(\alpha_\&)$

$\forall n : N \ \sigma(0..n), \sigma_d(0..n) \models_f F_C(\alpha_\&)$

$\Rightarrow$ {definition of ts3}

$\forall n : N \ len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and

$\sigma(1..n), \sigma_d(1..n) \models_f \top)$ or $(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..n), \sigma_d(1..n) \models_f C))$

$\Rightarrow$ {inductive hypothesis}

$len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty \top)$

or $(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty C))$

$\Rightarrow$ {σ is infinite so $len(\sigma) \neq 0$}

$\sigma_d(0) = F\alpha$ and $((\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty \top)$

or $(\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty C))$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty F_C(\alpha_\&)$

Case $F_C(\alpha; \alpha')$

$\forall n : N \ \sigma(0..n), \sigma_d(0..n) \models_f F_C(\alpha; \alpha')$

$\Rightarrow$ {definition of ts3}

$\forall n : N \ \sigma_d(0) = F\alpha$ and $(\sigma(0..n), \sigma_d(0..n) \models_f F_\perp(\alpha)$ or

$\sigma(0..n), \sigma_d(0..n) \models_f [\alpha]F_C(\alpha'))$

$\Rightarrow$ {inductive hypothesis}

$\sigma_d(0) = F\alpha$ and $(\sigma, \sigma_d \models^d_\infty F_\perp(\alpha)$ or $\sigma, \sigma_d \models^d_\infty [\alpha]F_C(\alpha'))$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty F_C(\alpha; \alpha')$

Case $F_C(\alpha + \alpha')$

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f F_C(\alpha + \alpha')$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f F_C(\alpha) \wedge F_C(\alpha')$

$\Rightarrow$ {inductive hypothesis}

$\sigma, \sigma_d \models^d_\infty F_C(\alpha) \wedge F_C(\alpha')$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty F_C(\alpha + \alpha')$

Case $P(\alpha_\&)$

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f P(\alpha_\&)$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ len(\sigma) = 0$ or $\sigma_d(0) = P\alpha$ and $\sigma(1..n), \sigma_d(1..n) \models_f \top$

$\Rightarrow$ {inductive hypothesis}

$len(\sigma) = 0$ or $\sigma_d(0) = P\alpha$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty \top$

$\Rightarrow$ {σ is infinite so $len(\sigma) \neq 0$}

$\sigma_d(0) = P\alpha$ and $\sigma(1..), \sigma_d(1..) \models^d_\infty \top$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty P(\alpha_\&)$

Case $P(\alpha; \alpha')$

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f P(\alpha; \alpha')$

$\Rightarrow$ {definition of ts3}

$\forall n : N\ \sigma(0..n), \sigma_d(0..n) \models_f P(\alpha) \wedge [\alpha]P(\alpha')$

$\Rightarrow$ {inductive hypothesis}

$\sigma, \sigma_d \models^d_\infty P(\alpha) \wedge [\alpha]P(\alpha')$

$\Rightarrow$ {definition of ts2}

$\sigma, \sigma_d \models^d_\infty P(\alpha; \alpha')$

Case $P(\alpha + \alpha')$

$$
\begin{aligned}
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f P(\alpha + \alpha') \\
\Rightarrow\ & \{\text{definition of ts3}\} \\
& \forall n : N\ \sigma(0..n), \sigma_d(0..n) \vDash_f P(\alpha) \wedge P(\alpha') \\
\Rightarrow\ & \{\text{inductive hypothesis}\} \\
& \sigma, \sigma_d \vDash^d_\infty P(\alpha) \wedge P(\alpha') \\
\Rightarrow\ & \{\text{definition of ts2}\} \\
& \sigma, \sigma_d \vDash^d_\infty P(\alpha + \alpha')
\end{aligned}
$$

□ □

Lemma A.8. *ts3 is correct with respect to ts2*

$$\sigma, \sigma_d \vDash^d_\infty C \Leftrightarrow \forall n : N\sigma(0..n), \sigma_d(0..n) \vDash_f C$$

Proof. Directly from Lemma A.6 and Lemma A.7. □

Appendix B

Correctness of Algorithm

Lemma B.1. *Given a $\mathcal{CL}$ expression C, we may build an automaton $A(C)$ that will accept all and only the traces σ that satisfies the contract, that is $\sigma \vDash C$. This can be achieved by proving*

$$\sigma \vDash C \Leftrightarrow \sigma(1..) \vDash f(C, \sigma(0))$$

We shall prove this by applying induction on the structure of the formula.

Proof. **Case** $C_1 \wedge C_2$

$$\begin{aligned}
& \sigma \vDash C_1 \wedge C_2 \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \sigma \vDash C_1 \text{ and } \sigma \vDash C_2 \\
\Leftrightarrow\ & \{\text{Inductive hypothesis}\} \\
& \sigma(1..) \vDash f(C_1, \sigma(0)) \text{ and } \sigma(1..) \vDash f(C_2, \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of trace semantics } (\wedge)\} \\
& \sigma(1..) \vDash f(C_1, \sigma(0)) \wedge f(C_2, \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(C_1 \wedge C_2, \sigma(0))
\end{aligned}$$

Case $C_1 \oplus C_2$

$$\begin{array}{ll}
& \sigma \models C_1 \oplus C_2 \\
\Leftrightarrow & \{\text{Definition of Trace Semantics}\} \\
& (\sigma \models C_1 \text{ and } \sigma \not\models C_2) \text{ or } (\sigma \not\models C_1 \text{ and } \sigma \models C_2) \\
\Leftrightarrow & \{\text{Inductive Hypothesis}\} \\
& (\sigma(1..) \models f(C_1, \sigma(0)) \text{ and } \sigma(1..) \not\models f(C_2, \sigma(0))) \text{ or} \\
& \quad (\sigma(1..) \not\models f(C_1, \sigma(0)) \text{ and } \sigma(1..) \models f(C_2, \sigma(0))) \\
\Leftrightarrow & \{\text{Definition of Trace Semantics } (\oplus)\} \\
& \sigma(1..) \models f(C_1, \sigma(0)) \oplus f(C_2, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of } \oplus\} \\
& \sigma(1..) \models ((f(C_1, \sigma(0)) = 1 \wedge f(C_2, \sigma(0)) = 0) \text{ or} \\
& \quad (f(C_1, \sigma(0)) = 0 \wedge f(C_2, \sigma(0)) = 1) \text{ or} \\
& \quad f(C_1, \sigma(0)) \oplus f(C_2, \sigma(0))) \\
\Leftrightarrow & \{\text{Definition of f}\} \\
& \sigma(1..) \models f(C_1 \oplus C_2, \sigma(0))
\end{array}$$

Case $[\alpha_{\&}]C$

$$\begin{array}{ll}
& \sigma \models [\alpha_{\&}]C \\
\Leftrightarrow & \{\text{Definition of Trace Semantics}\} \\
& \alpha_{\&} \not\subseteq \sigma(0) \text{ or } \sigma(1..) \models C \\
\Leftrightarrow & \{\text{Inductive Hypothesis}\} \\
& \alpha_{\&} \not\subseteq \sigma(0) \text{ or } \sigma(2..) \models f(C, \sigma(1)) \\
\Leftrightarrow & \{\text{Definition of trace semantics } ([\,])\} \\
& \sigma(1..) \models [\alpha_{\&}]f(C, \sigma(1)) \\
\Leftrightarrow & \{\text{Definition of } [\,]\} \\
& \sigma(1..) \models (\alpha_{\&} \not\subseteq \sigma(0) \text{ or } f(C, \sigma(1)) \\
\Leftrightarrow & \{\text{Definition of f}\} \\
& \sigma(1..) \models f([\alpha_{\&}]C, \sigma(0))
\end{array}$$

Case $[\beta \cdot \beta']C$

$$\begin{array}{ll}
 & \sigma \models [\beta \cdot \beta']C \\
\Leftrightarrow & \{\text{Definition of Trace Semantics}\} \\
 & \sigma \models [\beta][\beta']C \\
\Leftrightarrow & \{\text{Inductive hypothesis}\} \\
 & \sigma(1..) \models f([\beta][\beta']C, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of f}\} \\
 & \sigma(1..) \models f([\beta \cdot \beta']C, \sigma(0))
\end{array}$$

Case $[\beta + \beta']C$

$$\begin{array}{ll}
 & \sigma \models [\beta + \beta']C \\
\Leftrightarrow & \{\text{Definition of Trace Semantics}\} \\
 & \sigma \models [\beta]C \text{ and } \sigma \models [\beta']C \\
\Rightarrow & \{\text{Inductive hypothesis}\} \\
 & \sigma(1..) \models f([\beta]C, \sigma(0)) \text{ and } \sigma(1..) \models f([\beta']C, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of trace semantics } (\wedge)\} \\
 & \sigma(1..) \models f([\beta]C, \sigma(0)) \wedge f([\beta']C, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of f } (\wedge)\} \\
 & \sigma(1..) \models f([\beta]C \wedge [\beta']C, \sigma(0)) \\
\Leftrightarrow & \{\text{Definition of f}\} \\
 & \sigma(1..) \models f([\beta + \beta']C, \sigma(0))
\end{array}$$

Case $O_C(\alpha_{\&})$

$$\begin{aligned}
& \sigma \vDash O_C(\alpha_{\&}) \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \alpha_{\&} \subseteq \sigma(0) \text{ or } \sigma(1..) \vDash C \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \alpha_{\&} \subseteq \sigma(0) \text{ or } \sigma(2..) \vDash f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of Obligation}\} \\
& \sigma(1..) \vDash (O_{f(C,\sigma(1))}(\alpha_{\&})) \\
\Leftrightarrow\ & \{\text{Opening of Obligation}\} \\
& \sigma(1..) \vDash (\alpha_{\&} \subseteq \sigma(0) \text{ or } f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(O_C(\alpha_{\&}), \sigma(0))
\end{aligned}$$

Case $O_C(\alpha \cdot \alpha')$

$$\begin{aligned}
& \sigma \vDash O_C(\alpha \cdot \alpha') \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \sigma \vDash O_C(\alpha) \text{ and } [\alpha]O_C(\alpha') \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \sigma(1..) \vDash f(O_C(\alpha)) \text{ and } \sigma(1..) \vDash ([\alpha]O_C(\alpha')) \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics } (\wedge)\} \\
& \sigma(1..) \vDash f(O_C(\alpha) \wedge [\alpha]O_C(\alpha')) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(O_C(\alpha \cdot \alpha'), \sigma(0))
\end{aligned}$$

Case $O_C(\alpha + \alpha')$

$\sigma \vDash O_C(\alpha + \alpha')$

$\Leftrightarrow$ {Definition of Trace Semantics}

$\sigma \vDash O_\perp(\alpha)$ or $\sigma \vDash O_\perp(\alpha')$ or $\sigma \vDash [\overline{\alpha + \alpha'}]C$

$\Leftrightarrow$ {Inductive Hypothesis}

$\sigma(1..) \vDash f(O_\perp(\alpha), \sigma(0))$ or $\sigma(1..) \vDash f(O_\perp(\alpha'), \sigma(0))$
or $\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

Case : $\sigma(1..) \vDash f(O_\perp(\alpha), \sigma(0))$

$\Rightarrow$ {From definition of obligation of choice, $\sigma \vDash O(\alpha) \rightarrow \sigma \vDash O(\alpha + \alpha')$}

$\sigma(1..) \vDash f(O_\perp(\alpha + \alpha'), \sigma(0))$

$\Rightarrow$ {If trace satisfies clause with no reparation, then it will also satisfy one with a reparation($O_\perp(\alpha) \rightarrow O_C(\alpha)$)}

$\sigma(1..) \vDash f(O_C(\alpha + \alpha'), \sigma(0))$

Case : $\sigma(1..) \vDash f(O_\perp(\alpha'), \sigma(0))$

$\Rightarrow$ {From definition of obligation of choice, $\sigma \vDash O(\alpha) \rightarrow \sigma \vDash O(\alpha + \alpha')$}

$\sigma(1..) \vDash f(O_\perp(\alpha + \alpha'), \sigma(0))$

$\Rightarrow$ {If trace satisfies clause with no reparation, then it will also satisfy one with a reparation($O_\perp(\alpha) \rightarrow O_C(\alpha)$)}

$\sigma(1..) \vDash f(O_C(\alpha + \alpha'), \sigma(0))$

Case : $\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$\Rightarrow$ {Definition of f}

$\sigma(1..) \vDash (1 \text{ if } (f([\overline{\alpha}]C, \sigma(0)) = 1 \text{ or } f([\overline{\alpha'}]C, \sigma(0)) = 1))$ or
$(C \text{ if } (f([\overline{\alpha}]C, \sigma(0)) = f([\overline{\alpha'}]C, \sigma(0))) = C)$ or $[\overline{\alpha + \alpha'/\sigma(0)}]C$

$\Rightarrow$ {Definition of f ($f([\overline{\alpha}]C, \sigma(0)) = 1 \rightarrow f(O_\perp(\alpha), \sigma(0)) = 1$)}

$\sigma(1..) \vDash (1 \text{ if } (f(O_\perp(\alpha), \sigma(0)) = 1 \text{ or } f(O_\perp(\alpha'), \sigma(0)) = 1))$ or
$(C \text{ if } (f([\overline{\alpha}]C, \sigma(0)) = f([\overline{\alpha'}]C, \sigma(0))) = C)$ or
$[\overline{\alpha + \alpha'/\sigma(0)}]C$

$\Rightarrow$ {Definition of f ($f([\overline{\alpha}]C, \sigma(0)) = C \rightarrow f(O_C(\alpha), \sigma(0)) = C$)}

$\sigma(1..) \vDash (1 \text{ if } (f(O_\perp(\alpha), \sigma(0)) = 1 \text{ or } f(O_\perp(\alpha'), \sigma(0)) = 1))$ or
$(C \text{ if } (f(O_C(\alpha), \sigma(0)) = f(O_C(\alpha'), \sigma(0))) = C)$ or
$[\overline{\alpha + \alpha'/\sigma(0)}]C$

$\Rightarrow$ {Definition of trace semantics ($\sigma \vDash [\overline{\alpha}]C \leftrightarrow \sigma \vDash O_C(\alpha)$)}

$\sigma(1..) \vDash (1 \text{ if } (f(O_\perp(\alpha), \sigma(0)) = 1 \text{ or } f(O_\perp(\alpha'), \sigma(0)) = 1))$ or
$(C \text{ if } (f(O_C(\alpha), \sigma(0)) = f(O_C(\alpha'), \sigma(0))) = C)$ or
$O_C(\alpha + \alpha'/\sigma(0))$

$\Rightarrow$ {Definition of f}

$\sigma(1..) \vDash f(O_C(\alpha + \alpha'), \sigma(0))$

$Case : \sigma(1..) \vDash (1 \text{ if } (f(O_\bot(\alpha), \sigma(0)) = 1 \text{ or } f(O_\bot(\alpha'), \sigma(0)) = 1)))$

$\Rightarrow$ {Definition of trace semantics and f}
$\sigma(1..) \vDash f(O\bot(\alpha), \sigma(0)) \text{ or } \sigma(1..) \vDash f(O\bot(\alpha'), \sigma(0))$

$\Rightarrow$ {Disjunction introduction}
$\sigma(1..) \vDash f(O_\bot(\alpha), \sigma(0)) \text{ or } \sigma(1..) \vDash f(O_\bot(\alpha'), \sigma(0)) \text{ or }$
$\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$Case : C \text{ if } (f(O_\bot(\alpha), \sigma(0)) = f(O_\bot(\alpha'), \sigma(0)) = 0)$

$\Rightarrow$ {Definition of trace semantics and f}
$\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$\Rightarrow$ {Disjunction introduction}
$\sigma(1..) \vDash f(O_\bot(\alpha), \sigma(0)) \text{ or } \sigma(1..) \vDash f(O_\bot(\alpha'), \sigma(0)) \text{ or }$
$\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$Case : \sigma(1..) \vDash O_C(\alpha + \alpha'/\sigma(0)$

$\Rightarrow$ {Definition of trace semantics and f}
$\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$\Rightarrow$ {Disjunction introduction}
$\sigma(1..) \vDash f(O_\bot(\alpha), \sigma(0)) \text{ or } \sigma(1..) \vDash f(O_\bot(\alpha'), \sigma(0)) \text{ or }$
$\sigma(1..) \vDash f([\overline{\alpha + \alpha'}]C, \sigma(0))$

$\Rightarrow$ {From sub-cases}
$\sigma(1..) \vDash (1 \text{ if } (f(O_\bot(\alpha), \sigma(0)) = 1 \text{ or } f(O_\bot(\alpha'), \sigma(0)) = 1))) \text{ or }$
$(C \text{ if } (f(O_\bot(\alpha), \sigma(0)) = f(O_\bot(\alpha'), \sigma(0)) = 0))) \text{ or } O_C(\alpha + \alpha'/\sigma(0))$

$\Leftrightarrow$ {Definition of f}
$\sigma(1..) \vDash f(O_C(\alpha + \alpha'), \sigma(0))$

Case $F_C(\alpha_{\&})$

$$\begin{aligned}
& \sigma \vDash F_C(\alpha_{\&}) \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \alpha_{\&} \nsubseteq \sigma(0) \text{ or } \sigma(1..) \vDash C \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \alpha_{\&} \nsubseteq \sigma(0) \text{ or } \sigma(2..) \vDash f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of Prohibition}\} \\
& \sigma(1..) \vDash F_{f(C,\sigma(1))}(\alpha_{\&}) \\
\Leftrightarrow\ & \{\text{Opening of Prohibition}\} \\
& \sigma(1..) \vDash \alpha_{\&} \nsubseteq \sigma(0) \text{ or } f(C, \sigma(1))) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(F_C(\alpha_{\&}), \sigma(0))
\end{aligned}$$

Case $F_C(\alpha \cdot \alpha')$

$$\begin{aligned}
& \sigma \vDash F_C(\alpha \cdot \alpha') \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \sigma \vDash [\alpha]F_C(\alpha') \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \sigma(1..) \vDash f([\alpha]F_C(\alpha'), \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \vDash f(F_C(\alpha \cdot \alpha'), \sigma(0))
\end{aligned}$$

Case $F_C(\alpha + \alpha')$

$$
\begin{aligned}
& \sigma \models F_C(\alpha + \alpha') \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \sigma \models F_C(\alpha) \text{ and } \sigma \models F_C(\alpha') \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \sigma(1..) \models f(F_C(\alpha), \sigma(0)) \text{ and } \sigma(1..) \models f(F_C(\alpha'), \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \sigma(1..) \models f(F_C(\alpha), \sigma(0)) \wedge f(F_C(\alpha'), \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of f } (\wedge)\} \\
& \sigma(1..) \models f(F_C(\alpha) \wedge F_C(\alpha'), \sigma(0)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \models f(F_C(\alpha + \alpha'), \sigma(0))
\end{aligned}
$$

Case $[\overline{\alpha_\&}]C$

$$
\begin{aligned}
& \sigma \models [\overline{\alpha_\&}]C \\
\Leftrightarrow\ & \{\text{Definition of Trace Semantics}\} \\
& \alpha_\& \subseteq \sigma(0) \text{ or } \sigma(1..) \models C \\
\Leftrightarrow\ & \{\text{Inductive Hypothesis}\} \\
& \alpha_\& \subseteq \sigma(0) \text{ or } \sigma(2..) \models f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of trace semantics } ([\,])\} \\
& \sigma(1..) \models [\overline{\alpha_\&}]f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of } [\,]\} \\
& \sigma(1..) \models (\alpha_\& \subseteq \sigma(0) \text{ or } f(C, \sigma(1)) \\
\Leftrightarrow\ & \{\text{Definition of f}\} \\
& \sigma(1..) \models f([\overline{\alpha_\&}]C, \sigma(0))
\end{aligned}
$$

Case $[\beta \cdot \beta']C$

$\sigma \models [\overline{\beta \cdot \beta'}]C$

$\Leftrightarrow$ {Definition of Trace Semantics}

$\sigma \models [\overline{\beta}][\overline{\beta'}]C$

$\Leftrightarrow$ {Inductive hypothesis}

$\sigma(1..) \models f([\overline{\beta}][\overline{\beta'}]C, \sigma(0))$

$\Leftrightarrow$ {Definition of f}

$\sigma(1..) \models f([\overline{\beta \cdot \beta'}]C, \sigma(0))$

Case $[\overline{\beta + \beta'}]C$

$\sigma \models [\overline{\beta + \beta'}]C$

$\Leftrightarrow$ {Definition of Trace Semantics}

$\sigma \models [\overline{\beta}]C$ or $\sigma \models [\overline{\beta'}]C$

$\Rightarrow$ {Inductive hypothesis}

$\sigma(1..) \models f([\overline{\beta}]C, \sigma(0))$ or $\sigma(1..) \models f([\overline{\beta'}]C, \sigma(0))$

$\Leftrightarrow$ {Definition of f}

$\sigma(1..) \models C$ if $(f([\overline{\alpha}]C, \delta) = C$ or $f([\overline{\alpha'}]C, \delta) = C)$
or $\sigma(1..) \models 1$ if $(f([\overline{\alpha}]C, \delta) = f([\overline{\alpha'}]C, \delta) = 1)$ or $\sigma(1..) \models [\overline{\alpha + \alpha'/\delta}]C$

$\Leftrightarrow$ {Definition of f}

$\sigma(1..) \models f([\overline{\beta + \beta'}]C, \sigma(0))$

□

Bibliography

[1] A. R. Anderson. Some nasty problems in the formalization of ethics. *Noûs*, 1:345–360, 1967.

[2] G. Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.

[5] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[6] P. Blackburn, J. F. A. K. van Benthem, and F. Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning).* Elsevier Science Inc., New York, NY, USA, 2006.

[7] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *In Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156. MIT Press, 1996.

[8] J. Broersen. Action negation and alternative reductions for dynamic deontic logics. *J. Applied Logic*, 2(1):153–168, 2004.

[9] J. Broersen, R. Wieringa, and J.-J. C. Meyer. A fixed-point characterization of a deontic logic of regular action. *Fundam. Inf.*, 48(2-3):107–128, 2001.

[10] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[11] E. S. J. P. W. Carlos Molina-Jiménez, Santosh K. Shrivastava. Runtime Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications*, 3(2), 2004.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176–195, 2001.

[13] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking.* The MIT Press, 1999.

[15] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008).* To be published by Springer Verlag in Lecture Notes in Computer Science, 2008.

[16] A. Daskalopulu. Model Checking Contractual Protocols. In L. Breuker and Winkels, editors, *Legal Knowledge and Information Systems, JURIX 2000: The 13th Annual Conference*, Frontiers in Artificial Intelligence and Applications Series, pages 35–47. IOS Press, 2000.

[17] A. Daskalopulu and T. S. E. Maibaum. Towards Electronic Contract Performance. In *Legal Information Systems Applications, 12th International Conference and Workshop on Database and Expert Systems Applications*, pages 771–777. IEEE C.S. Press, 2001.

[18] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proceedings of WWW2004*, pages 144–153, May 2004.

[19] D. A. P. Edmund M. Clarke Jr., Orna Grumberg. *Model Checking.* 1999.

[20] E. A. Emerson. Temporal and modal logic. pages 995–1072, 1990.

[21] E. A. Emerson and J. Y. Halpern. ’̈sometimes’̈ and ’̈not never’̈ revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[22] J. L. Fiadeiro and T. S. E. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Asp. Comput.*, 4(3):239–272, 1992.

[23] M. Fitting and R. L. Mendelsohn. *First-order modal logic.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[24] G. Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216, 2005.

[25] G. Governatori and A. Rotolo. Logic of violations: A gentzen system for reasoning with contrary-to-duty obligations. *Australatian Journal of Logic*, 4:193–215, 2006.

[26] J. Hintikka. Some main problems of deontic logic. In R. Hilpinen, editor, *Deontic Logic: Introductory and Systematic Readings*, pages 59–104. D. Reidel Publ. Co., Dordrecht, 1971.

[27] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[28] H. E. Jensen. *Abstraction-based verification of distributed systems /- by Henrik Ejersbo Jensen.* PhD thesis, Aalborg, Denmark :Aalborg University, Dept. of Computer Science,, 1999.

[29] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[31] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[32] D. Kozen and R. Parikh. A decision procedure for the propositional μ-calculus. In E. M. Clarke and D. Kozen, editors, *4th Workshop on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 313–325. Springer, 1983.

[33] O. Kupferman and M. Y. Vardi. From linear time to branching time. *ACM Trans. Comput. Logic*, 6(2):273–294, 2005.

[34] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA'08*, LNCS. Springer-Verlag, October 2008. To appear.

[35] L. Lamport. "sometime" is sometimes "not never": on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM.

[36] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[37] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM.

[38] M. Maidi. The common fragment of ctl and ltl. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 643, Washington, DC, USA, 2000. IEEE Computer Society.

[39] E. Mally. *Grundgesetze des Sollens. Elemente fer Logik des Willens.* Graz: Leuschner & Lubensky, 1926.

[40] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.*, 46(3):255–281, 2003.

[41] K. L. McMillan. *Symbolic Model Checking:An approach to the state explosion problem.* PhD thesis, Carnegie Mellon university, May 1992.

[42] K. L. McMillan. *The SMV language.* Cadence Berkeley Labs, 2001 Addison St. Berkeley, CA 94704 USA, 1st edition, March 1999.

[43] J.-J. C. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29:109–136, 1988.

[44] J.-J. C. Meyer, F. Dignum, and R. Wieringa. The paradoxes of deontic logic revisited: A computer science perspective (or: Should computer scientists be bothered by the concerns of philosophers?). Technical Report UU-CS-1994-38, Department of Information and Computing Sciences, Utrecht University, 1994.

[45] J.-J. C. Meyer and R. J. Wieringa, editors. *Deontic logic in computer science: normative system specification.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[46] H. C. neda. The paradoxes of deontic logic: The simplest solution to all of them in one fell swoop. In R. Hilpinen, editor, *New Studies in Deontic Logic*, pages 37–85. D. Reidel Publishing Company, Dordrecht, 1981.

[47] M. J. Osborne and A. Rubinstein. *A Course in Game Theory.* MIT Press, 1999.

[48] G. Pace, C. Prisacariu, and G. Schneider. Model Checking Contracts –a case study. In *5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 82–97, Tokyo, Japan, October 2007. Springer.

[49] Y. V. Patrick Blackburn, Maarten de Rijke. *Modal Logic*. Cambridge University Press, 2001.

[50] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[51] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, London, UK, 1979. Springer-Verlag.

[52] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[53] V. R. Pratt. Dynamic algebras and the nature of induction. In *12th ACM Symposium on Theory of Computing (STOC'80)*, pages 22–28. ACM, 1980.

[54] C. Prisacariu and G. Schneider. An Algebraic Structure for the Action-Based Contract Language CL - theoretical results. Technical Report 361, Department of Informatics, University of Oslo, Oslo, Norway, July 2007.

[55] C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In M. Bonsangue and E. B. Johnsen, editors, *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189, Paphos, Cyprus, June 2007. Springer.

[56] C. Prisacariu and G. Schneider. Towards a Formal Definition of Electronic Contracts. Technical Report 348, Department of Informatics, University of Oslo, Oslo, Norway, January 2007.

[57] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[58] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(2):81–132, 1980.

[59] R. Reussner et al. CoCoME - the common component modelling example. To appear in Lecture Notes in Computer Science, 2008.

[60] I. Song and G. Governatori. Nested rules in defeasible logic. In *RuleML*, volume 3791 of *Lecture Notes in Computer Science*, pages 204–208, 2005.

[61] R. H. Thomason. Deontic logic as founded on tense logic. In R. Hilpinen, editor, *New studies in deontic logic: norms, actions, and the foundations of ethics*, pages 165–176. D. Reidel Publishing Company, Dordrecht, Holland, 1981.

[62] R. van der Meyden. The dynamic logic of permission. In *LICS*, pages 72–78, 1990.

[63] J. van Eck. A system of temporally relative modal and deontic predicate logic and its philosophical applications. *Logique et Analyse*, 99:249–290, 1982.

[64] I. Walukiewicz. *A Complete Deductive System for the μ-Calculus.* PhD thesis, 1993.

[65] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. In *10th IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 14–24. IEEE Computer Society, 1995.

[66] R. Wieringa, J.-J. C. Meyer, and H. Weigand. Specifying dynamic and deontic integrity constraints. *Data Knowl. Eng.*, 4:157–189, 1989.

[67] R. J. Wieringa and J. j. Ch. Meyer. Applications of deontic logic in computer science: A concise overview. In *Deontic Logic in Computer Science: Normative System Specification*, pages 17–40. John Wiley & Sons, 1993.

[68] R. J. Wieringa, H. Weigand, J. j. Ch. Meyer, and F. P. M. Dignum. The inheritance of dynamic and deontic integrity constraints. *Annals of Mathematics and Artificial Intelligence*, 1991:393–428, 1991.

[69] G. H. V. Wright. Deontic logic. *Mind*, (60):1–15, 1951.

[70] G. H. V. Wright. A new system of deontic logic. *Danish Yearbook of Philosophy 1*, pages 173–182, 1964.

[71] G. H. V. Wright. A correction to a new system of deontic logic. *Danish Yearbook of Philosophy 2*, pages 103–107, 1965.

[72] G. H. V. Wright. Deontic logic: A personal view. *Ratio Juris*, 12(1):26–38, 1999.

[73] E. N. Zalta. Basic concepts in modal logic. Technical report, Center for the Study of Language and Information, 1995.