A Dynamic Deontic Logic over Synchronous Actions

Thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
to the Faculty of Mathematics and Natural Sciences,
at the University of Oslo.

Cristian Prisacariu
Precise Modeling and Analysis Group (PMA)
Department of Informatics, University of Oslo

Supervisors:
Assoc. Prof. Gerardo Schneider, IT University of Götenburg
Prof. Olaf Owe, University of Oslo

Thesis committee:
Prof. Marek Sergot, Imperial College London
Assoc. Prof. Thomas Ågotnes, University of Bergen
Prof. Herman Ruge Jervell, University of Oslo

September 13, 2010
Abstract

We present a dynamic deontic logic for specifying and reasoning about complex contracts. The concepts that our contract logic $\mathcal{CL}$ captures are drawn from legal contracts, as we consider that these are more general and expressive that what is usually found in computer science (like in software contracts, web services specifications, or communication protocols). $\mathcal{CL}$ is intended to be used in specifying complex contracts found in computer science. This influences many of the design decisions behind $\mathcal{CL}$. We adopt an ought-to-do approach to deontic logic and apply the deontic modalities exclusively over actions. The actions that we consider are not just basic atomic actions, as in standard multi-modal logics, but have a complex structure that extends the regular structure of the dynamic logic actions. We add to $\mathcal{CL}$ the dynamic logic modality so to be able to reason about what happens after an action is performed. $\mathcal{CL}$ incorporates the notions of contrary-to-duty and contrary-to-prohibition by explicitly attaching to the deontic modalities a reparation which is meant to be enforced in the case of violations. We prove results of decidability and tree model property for the $\mathcal{CL}$ logic, as well as specific properties for the modalities.

$\mathcal{CL}$ can reason about regular synchronous actions done at the same time. We thoroughly investigate and precisely formalize this notion of synchrony as the theory of synchronous Kleene algebra ($\text{SKA}$). The theory of $\text{SKA}$ is used in giving semantics both to the deontic modalities and to the dynamic logic (PDL) modality over the synchronous actions. The extension of PDL with a notion of concurrent actions (specifically, the synchrony notion) is an interesting result in itself. We prove that $\text{SKA}$ (and its extension with Boolean tests) is complete, fact which makes the problem of deciding the equality of two synchronous actions solvable in polynomial time and space.

From a practical point of view we show how to do run-time monitoring of electronic contracts using the $\mathcal{CL}$ logic. Precisely, we give an algorithm for building a monitor (an abstract deterministic machine) which monitors all the actions governed by the contract (defined as a $\mathcal{CL}$ formula). The observable outcome of the monitor is the signaling of the first violation of the contract immediately after this happens. The construction of the monitor involves working with alternating Büchi automata and the use of a recent ingenious method based on a 3-valued semantics approach to run-time monitoring. For doing run-time monitoring of contracts written in $\mathcal{CL}$ we need to develop trace semantics for $\mathcal{CL}$ which is faithful with the full semantics based on normative structures.
Abstract
(for general audience)

Motto: You don’t understand your contract?!! Ask the computer!

How can we use the speed and precision of computers to help with our complicated legal contracts? Can computers understand and reason about legal contracts? Can they answer complicated questions? Can they be trusted to sign electronic contracts for us when we do transactions on the Internet? With proper mathematical theory they can, and can do more!

In this respect, this thesis lays the foundations for how computers should understand and reason about legal contracts. This research develops thoroughly the logic that the computer should have when reading and answering complicated questions about a contract. It is known that “Logic is everywhere inside a computer” even if normal users can’t notice it!

Besides the power to understand and reason, what other practical benefits can a computer get from such a logical theory? Well, now a computer may use techniques to detect errors in contracts automatically, monitor legal contracts, or check predefined general properties that one might expect from a contract before signing it. All these techniques are carefully developed in this thesis. Other practically useful outcomes of this research are: question answering, semi-automatic negotiation, and queries. Think for a second about the usefulness of automatic monitoring of legal contracts; this is easily recognized by any person who had to sign a contract and then had to be careful that her (or his) actions do not violate this contract. Now this stress can be relieved by the computer.

The logical theory developed in this thesis is highly expressive and deep knowledge is needed when implementing software applications based on it. Nevertheless, this powerful theory is invisible to the general user once implemented in a computer software with an intuitive graphical user interface. So now the computer has the theory to understand and the techniques to reason about and investigate legal contracts. Still, many practical applications of this research are yet to be revealed as many implementations are needed for the techniques that have been put forward.
Abstrakt
(for allmennpublikum)

Motto: Forstår du ikke kontrakten din?! Spør datamaskinen!

Hvordan kan vi få en datamaskin med sin hastighet og presisjon til å hjelpe oss med kompliserte juridiske kontrakter? Kan datamaskiner forstå og resonnere om juridiske kontrakter? Kan de svare på kompliserte spørsmål? Kan de bli betrodd å signere kontrakter for oss når vi fører forhandlinger på internett?

For å svare på alle disse spørsmålene, vil vi i avhandlingen undersøke på hvilken måte datamaskiner kan forstå og resonnere om juridiske kontrakter. Denne forskningen utvikler logikken som datamaskinen trenger for å lese og svare på kompliserte spørsmål om en kontrakt. Det er kjent at ”Logikk er overalt innen en datamaskin” selv om alminnelig bruker kan ikke legge merke til det!


Den logiske teorien utviklet i denne avhandlingen er svært uttrykksfull og grundige kunnskaper er forventet av dem som skal implementere programvare basert på den. Imidlertid, blir denne kraftige teorien usynlig for allmennbrukerne når den er implementert i et dataprogram med intuitivt brukergrensensnitt. På denne måten, har nå datamaskinen fått den nødvendige teorien for å forstå, og teknikken for å resonnere om og utforske juridiske kontrakter. Likevel, flere praktiske bruksområder av denne forskningen kommer til å bli avslørt med tiden, siden flere anvendelser er nødvendige for å fullføre teknikkene som har blitt lagt frem.
Acknowledgements!

Most important I would like to warmly thank my sweet wife Ioana Sabina for all her support and advice (and all the energy that she “inflicted” on me) during the time I spent doing the research presented in this thesis.

I would also like to thank my supervisors Gerardo Schneider and Olaf Owe for their time and understanding. Gerardo has collaborated a lot to all sides of the research presented here and has been very helpful with priceless advises about ethics, conduct, state of mind, or research tricks which I wish any PhD student to have. Olaf has been more than a mentor for my development as a person and as a researcher. His stimulating discussions brought important ideas and gave a lot of fuel to my research efforts.

The “Precise Modeling and Analysis” group at the University of Oslo has provided the perfect research environment. I found here a live and open atmosphere with friendly discussions which relax even in the most stressful research moments. I enjoyed many insightful research discussions with Martin Steffen and Einar Broch Johnsen.

Not the least, I would like to thank the administration and support people at the Department of Informatics, (especially Narve, Mozhdeh, and Øystein), which have offered the perfect support for my research needs and initiatives.

What more could one desire!?

The research presented in this thesis was partially funded by the Nordunet3 project “COSoDIS – Contract-Oriented Software Development for Internet Services”.
(http://www.ifi.uio.no/cosodis/)
Environmental Issues!

The research done for this thesis is *carbon neutral* as all carbon produced during the four years of work has been offset by the author. The main body of carbon production was done through traveling and flights which cumulated to approximately 20 metric tons of CO₂. An extra 10 metric tons of CO₂ has been added to account for electricity, printing, books, and other auxiliary research needs. A total of 30 metric tons of CO₂ have been offset partly for planting trees (through the offsetting authority [www.naturefund.de](http://www.naturefund.de)) and the other part for alternative green energy projects (through the offsetting authority [www.carbonfund.org](http://www.carbonfund.org)).
Preface

The present thesis is intended as self contained. No prior expertise is needed, though we expect the reader to have good basic knowledge of mathematics and computer science; in particular of universal algebra, logics (knowledge of modal logic would help), programming theory, and automata theory. For the uninitiated reader we recommend to consult the references given in the introductory chapter and in the preliminaries of each chapter.

How to read this thesis?

The present thesis can be read in different ways depending on the interest of the reader. The thesis covers three main subjects which are interesting in their own:

1. An extension of Kleene algebra with the notion of synchrony;
2. An action-based logic for reasoning about legal (or electronic) contracts;
3. A technique and exemplification of run-time monitoring of electronic contracts.

The reader who is interested only in the synchronous Kleene algebra can safely read only Chapter 2 and skip the rest of the thesis.

The reader who is interested in the CL logic for contracts can safely skip the algebra chapter and read only Chapter 3. The semantics of CL is based on the synchronous Kleene algebra and related results, and whenever these are needed the reader is referred to the appropriate place in Chapter 2. Nevertheless, for a thorough understanding of the full semantics of CL (based on the interpretation of the synchronous actions) the reader is encouraged to go through the concepts and results on the synchronous Kleene algebra of Chapter 2.

The reader who is interested only in the practical part of the CL logic can go directly to Chapter 4 which describes how to do run-time monitoring of electronic/legal contracts. More understanding of the CL logic, which is used in the run-time monitoring process, can be gained by reading Chapter 3 (maybe skipping the more technical parts).

The content of this thesis appeared, usually in more concise and short presentations, as the following international publications: Chapter 2 as the journal paper [Pri10], Chapter 3 as [PS09b] and the recently submitted journal paper [PS10], Chapter 4 as [KPS08]. As time is precious, I strongly encourage the reader to see these papers first and if more details are needed then to proceed to the appropriate chapter in this thesis.

The evolution during the years of the content presented in this thesis can be traced through the following publications: [PS07a, PPS07, KPS08, PS09b] (published in international conferences by Springer’s LNCS series), [PS09a] (published by ACM in the international conference
ICAIL, which is the main forum for artificial intelligence and law), [PS07c, Pri08b, PS07b, PS08, Pri08a] (published as UiO technical reports, most of the times backing some conference paper with proofs, examples, and more explanations), [PS07d, Pri07, Pri08c, Pri09b, Pri09a] (appeared with dissemination purposes in student workshops), and recently [Pri10, PS10] (published as journal papers in Elsevier’s JLAP). For a complete list of the author’s publications relevant to this thesis see the selected bibliography at page 130.
# Contents

1 Introduction .................................................. 1
1.1 Introduction .................................................. 1
1.2 A Contract Example ........................................... 2
1.3 Motivations and design decisions ............................ 3
1.4 Summary of contributions .................................... 6

2 Synchronous Kleene Algebra .................................. 9
2.1 Preliminaries .................................................. 9
2.1.1 Applications .............................................. 11
2.1.2 Kleene algebras ........................................... 11
2.1.3 Synchrony .................................................. 13
2.2 Extending Kleene Algebra with Synchrony ................. 14
2.2.1 Syntax and axiomatization ................................ 14
2.2.2 Deontic actions as $^*$-free actions .................... 18
2.2.3 Standard interpretation over synchronous sets ......... 24
2.2.4 Completeness and decidability ........................... 27
2.2.5 Deontic actions as trees .................................. 35
2.3 Synchronous Kleene Algebra with Boolean Tests ........ 37
2.3.1 Interpretation over sets of guarded synchronous strings 39
2.3.2 Automata on guarded synchronous strings .............. 42
2.3.3 Completeness and decidability ........................... 46
2.3.4 SKAT, Hoare logic, and shared-variables concurrency 47
2.4 Synchronous Kleene Algebra compared to Other Concurrency Models 48
2.4.1 Mazurkiewicz trace theory ................................ 49
2.4.2 Pomsets .................................................... 50
2.4.3 Concurrent Kleene algebra ................................ 52
2.5 Final Remarks on Synchronous Kleene Algebra ............ 53
2.5.1 Related Work .............................................. 53
2.5.2 Open problems ............................................ 54

3 The Contract Logic $\mathcal{CL}$ ............................... 57
3.1 Deontic Modalities over Synchronous Actions ............. 57
3.1.1 Properties of the deontic modalities ...................... 62
3.1.2 Decidability for deontic modalities ...................... 73
3.2 The Full Contract Logic ....................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Properties of the $\mathcal{CL}$ logic</td>
<td>82</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Paradoxes and Puzzles</td>
<td>85</td>
</tr>
<tr>
<td>3.3</td>
<td>Final Remarks</td>
<td>88</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Related Work</td>
<td>88</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Open problems</td>
<td>89</td>
</tr>
<tr>
<td>4</td>
<td>Run-time Monitoring of Electronic Contracts</td>
<td>91</td>
</tr>
<tr>
<td>4.1</td>
<td>Preliminaries</td>
<td>91</td>
</tr>
<tr>
<td>4.2</td>
<td>Semantics on Respecting Traces</td>
<td>92</td>
</tr>
<tr>
<td>4.3</td>
<td>Monitoring $\mathcal{CL}$ Specifications of Contracts</td>
<td>97</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Satisfiability checking for $\mathcal{CL}$ using alternating automata</td>
<td>97</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Constructing the monitor</td>
<td>104</td>
</tr>
<tr>
<td>4.4</td>
<td>Final Remarks and Open Problems</td>
<td>105</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Relating the full and the trace semantics - open problem</td>
<td>106</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Related work</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>Research directions and open problems</td>
<td>113</td>
</tr>
<tr>
<td>Bibliography</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>Author’s Selected Bibliography</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Rewriting theory and convergence for $^*$-free actions</td>
<td>131</td>
</tr>
<tr>
<td>B</td>
<td>Example of a full legal contract</td>
<td>135</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

Internet inter-business collaborations, virtual organizations, and web services, usually communicate through service exchanges which respect an implicit or explicit contract. Such a contract must unambiguously determine correct interactions, and what are the exceptions allowed, or penalties imposed in case of incorrect behavior.

Legal contracts, as found in the usual judicial or commercial arena, may serve as basis for defining such machine-oriented electronic contracts (or e-contracts for short). Ideally, e-contracts should be shown to be contradiction-free both internally, and with respect to the governing policies under which the contract is enacted. Moreover, there must be a run-time system ensuring that the contract is respected. In other words, contracts should be amenable to formal analysis allowing both static and dynamic verification, and therefore written in a formal language.

The present thesis reports on the state-of-the-art of the logic for contracts $\mathcal{CL}$. The goal of $\mathcal{CL}$ is to describe and prescribe, at an abstract level, behaviors of complex systems, as for instance concurrent programs, communicating intelligent agents, web services, or normative systems. From this point of view, $\mathcal{CL}$ needs to be expressive enough to capture behaviors of such systems. The purpose of this logic is not only to formalize such behaviors, but also to reason about them. Therefore, we aim at a decidable logic so to have hopes for automatic verification using formal tools like model-checking and run-time monitoring.

More precisely, $\mathcal{CL}$ has been designed to represent and reason about contracts (which, in particular, can take the form of software contracts, web services, interfaces, communication protocols, etc), and it combines deontic logic (i.e., the logic of legal/normative concepts) [vW51] with propositional dynamic logic (PDL, the logic of complex regular actions) [FL77]. The deontic part of $\mathcal{CL}$ can express obligations, permissions, and prohibitions over structured actions, as well as what happens when obligations or prohibitions are not respected. The dynamic part of $\mathcal{CL}$ expresses what happens after some action (possibly with complex structure) is performed.

A first version of the language $\mathcal{CL}$ has been presented in [PS07a], where explicit temporal operators (always, eventually, and until) were part of the syntax. An encoding into a version of the modal $\mu$-calculus with concurrent actions was used to give semantics. The $\mathcal{CL}$ language

---

1 The author of this thesis is a coauthor of [PS07a].
presented in this thesis is more expressive and has a cleaner syntax (with no syntactic restrictions); it was first introduced in [PS09b]. A variant of $\mathcal{CL}$ (without the propositional constants) was used in [KPS08] for doing run-time monitoring of electronic contracts using a restricted semantics based on traces of actions. This semantics was specially designed for monitoring the actions of the contracting parties at run-time with the purpose of detecting when the contract is violated. The presentation of [KPS08] did not give much explanation nor examples for the intuitions behind the choices that we made in the design of $\mathcal{CL}$. We do this in the present chapter (i.e., in Section 1.3). Our focus here is on the intended properties of the language. We present the full semantics of $\mathcal{CL}$ based on normative structures in Chapter 3.

The semantics of $\mathcal{CL}$ is based on the interpretation of the synchronous actions (both for the deontic and for the dynamic modalities). We study synchronous actions and their interpretation in Chapter 2 by developing a general theory called synchronous Kleene algebra. More extensive motivations about the generality of this theory of synchronous actions and its applicability to $\mathcal{CL}$ are given in the introductory section of Chapter 2.

In a few words, synchronous Kleene algebra ($\text{SKA}$) extends Kleene algebra with the synchrony model of concurrency from R. Milner's SCCS calculus. Models of $\text{SKA}$ are given in terms of sets of synchronous strings and finite automata accepting synchronous strings. We also present an extension of synchronous Kleene algebra with Boolean tests together with models on sets of guarded synchronous strings and the associated automata on guarded synchronous strings. The synchronous Kleene algebra with tests ($\text{SKAT}$) is used in the semantics of the dynamic logic modality of $\mathcal{CL}$. Synchronous Kleene algebra is a true concurrency model and we compare it with Mazurkiewicz traces (which yields their incomparability with synchronous Kleene algebras, i.e., one cannot simulate the other) and with pomsets for which we isolate exactly the class of, what we call, synchronous pomsets.

On the practical side, in Chapter 4, we do run-time monitoring of contracts specified in the language of $\mathcal{CL}$. As far as we know this is the first account of run-time monitoring of electronic/legal contracts that is firmly based on a logical formalism. The run-time monitoring technique requires a trace semantics for $\mathcal{CL}$.

In the next section we focus on the informal explanation of the language and in particular on its design decisions. These design decisions are biased by the fact that $\mathcal{CL}$ is intended to faithfully capture concepts and natural properties from electronic contracts and to avoid the main deontic paradoxes (which are presented and discussed in the context of $\mathcal{CL}$ in Section 3.2.2).

In Section 1.4 we summarize the main contributions of this thesis.

### 1.2 A Contract Example

Throughout the thesis most of our examples are from the legal domain, precisely from the example of a legal contract given in full in Appendix B. Mainly we consider part 7 of that contract between an Internet service provider and a client, where the provider gives access to the Internet to the client. We consider two parameters of the service: high and normal, which denote the client’s Internet traffic (we abstract away from several technical details as how is measured the Internet traffic). Throughout this thesis we will often refer to the following clauses of the contract:
7.1. The Client shall not:
   a) supply false information to the Client Relations Department of the Provider.
7.2. Whenever the Internet Traffic is high then the Client must pay \([price]\) immediately, or the Client must notify the Provider by sending an e-mail specifying that he will pay later.
7.3. If the Client delays the payment as stipulated in 7.2, after notification he must immediately lower the Internet traffic to the normal level, and pay later twice \((2 \times [price])\).
7.4. If the Client does not lower the Internet traffic immediately, then the Client will have to pay \(3 \times [price]\).
7.5. The Client shall, as soon as the Internet Service becomes operative, submit within seven (7) days the Personal Data Form from his account on the Provider’s web page to the Client Relations Department of the Provider.

We also add the clause 11.2 as it is strongly related to clause 7.1 and the two should be taken together:

11.2. Provider may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:
   b) Suspend Internet Services immediately if Client is in breach of Clause 7.1;

### 1.3 Motivations and design decisions

In this section we motivate the particular choices we made in the design of the \(\mathcal{CL}\) contract specification language. We compare to related works and give informal intuitions and examples.

The purpose of \(\mathcal{CL}\) is to specify and reason about contracts, therefore \(\mathcal{CL}\) integrates the normative notions of obligation, permission, and prohibition. These have been extensively investigated in deontic logic \([vW51]\) which is a modal logic where only the \(K\) and \(D\) axioms hold and not the \(T\) axiom (ignore this last axiomatization fact if you are not familiar with modal logic). The deontic notions that we introduce in \(\mathcal{CL}\) are different than the ones in standard deontic logic (SDL) in several respects, as discussed in the rest of this section.

(1) The deontic modalities are applied only over actions instead of over propositions (or state of affairs) as is done in SDL. This is known as the ought-to-do approach to deontic logic as opposed to the more classic ought-to-be approach of SDL. The ought-to-do approach has been advocated by G.H. von Wright \([vW68]\) which argued that deontic logic would benefit from a “foundation of actions”, since many of the philosophical paradoxes of SDL would then not occur; a point which is made clear and precise in \([MDW94]\). Important contributions to this approach were done by K. Segerberg for introducing actions inside the deontic modalities \([Seg82, Seg92]\) and by the seminal work of J.-J.Ch. Meyer on dynamic deontic logic (DDL) \([Mey88]\) (see also \([BWM01, Wyn06]\)).

Compared to \([Mey88, BWM01, VdM90, CM07]\), which also consider deontic modalities (i.e., \(O\), \(P\), and \(F\)) applied over actions, the investigation presented in this thesis at the level of the actions is different in several ways as we elaborate below. The formalization of the actions is thoroughly investigated in Chapter 2 where standard models are defined for actions and completeness results are established. The semantics of the \(\mathcal{CL}\) language is based on this interpretation of the actions as special trees.
The action combinators are the standard + and · (for choice and sequence) but exclude the Kleene star \( \ast \). None of the few papers that consider repetition (e.g. using \( \ast \)) as an action combinator under deontic modalities [VdM90, BWM01] gives a convincing motivation for having such recurring actions inside obligations, permissions, or prohibitions. In fact its use inside the deontic modalities seems counter-intuitive: take the expression \( O(a^\ast) \) - which, using the interpretation of the Kleene \( \ast \), is read “One is obliged to not pay, or pay once, or pay twice in a row, or...” - which puts no actual obligations; or take \( P(a^\ast) \) - “One has the right to do any sequence of action \( a \)” - which is a very shallow permission and is captured by the widespread Closure Principle in jurisprudence where what is not forbidden is permitted [Seg82]. Moreover, as pointed out in [BWM01], expressions like \( F(a^\ast) \) and \( P(a^\ast) \) can be simulated with the propositional dynamic logic (PDL) modalities along with deontic modalities over actions without the Kleene \( \ast \). See subsection 3.2.2 for more discussions and examples related to \( \ast \).

The theory that we develop in this thesis for CL, i.e., the semantics and various proofs, is already quite involved without using the \( \ast \) inside the deontic modalities. If we were to add the Kleene \( \ast \) to capture some esoteric examples that one might find appealing the complexity that this would trigger in terms of theory and proofs does not justify its effort.

1. CL defines an action complement operation which encodes the violation of an obligation. Obligations (and prohibitions) can be violated by not doing the obligatory action (respectively doing the forbidden action). The action complement that we have is different from the various notions of action negation found in the literature on PDL or DDL-like logics [Mey88, HKT00, LW04, Bro03]. In [Mey88], as in [HKT00], action negation is with respect to the universal relation which for PDL gives undecidability. Decidability of PDL with negation of only atomic actions has been achieved in [LW04]. A so called “relativized action complement” is defined in [Bro03] which is the complement of an action (not w.r.t. the universal relation but) w.r.t. a set of atomic actions closed under the application of some action operators. This kind of negation still gives undecidability when several action operators are involved.

In CL the action complement is a derived operator defined as a function which takes a compound action and returns another compound action (i.e., action complement is not a principal combinator like +, ·, or \( \times \)). Intuitively the complement comprises of all the immediate actions that take us outside the tree of the complemented action [BWM01].

4. One difference from the standard PDL is that we consider deterministic actions. This is natural and desired in legal contracts as opposed to the programming languages community where nondeterminism is an important notion. In contracts the outcome of an action like “deposit 100$ in the bank account” is uniquely determined. The deterministic PDL has been investigated in [BAHP81]. On the other hand deterministic PDL is undecidable if action negation (or intersection of actions) is added [HKT00].

5. We add a concurrency operator \( \times \) to model that two actions are done at the same time. The model of concurrency that we adopt is the synchrony model of R. Milner’s SCCS [Mil83]. Synchrony is a natural choice when reasoning about the notion “at the same time” for human-like actions as we have in legal contracts (opposed to the instructions in a programming language). Moreover, from an algebraic point of view, synchrony is easy to integrate with the other regular operations on actions (the choice and the sequence).

The notion of synchrony has different meanings in different areas of computer science. Here
we take the distinction between *synchrony* and *asynchrony* as presented in the SCCS calculus and later implemented in, e.g., the Esterel synchronous programming language [Ber00]. We understand *asynchrony* as when two concurrent systems proceed at indeterminate relative speeds (i.e., their actions may have different non-correlated durations); whereas in the *synchrony* model each of the two concurrent systems instantaneously perform a single action at each time instant. This is an abstract view of the actions found in contracts which is good for reasoning about quite a big range of properties for contracts, like properties that do not take into consideration the structure or types of the actions. Such properties would look only at the interplay of actions, temporal ordering, choice, or existence of actions. If one needs actions which have durations (e.g., “work 3 hours”) or which are parameterized by amounts (e.g., “deposit 100$”) then $\mathcal{CL}$ has to be extended accordingly.

The *synchrony model* of concurrency takes the assumption that time is discrete and that basic actions are instantaneous and represent the time step. Moreover, at each time step all possible actions are performed, i.e., the system is considered *eager* and *active*. For this reason if at a time point there is enabled an obligation to do an action then this action must be immediately executed so that the obligation is not violated. Synchrony assumes a global clock which provides the time for all the actors (participants, parallel components) in the system. Note that for practical implementation purposes this is a rather strong assumption which offends the popular view from process algebras [Mil95, Hoa85]. On the other hand the mathematical framework of the synchrony model is much cleaner and more general than the asynchronous interleaving model (SCCS has the (asynchronous) CCS as a subcalculus [Mil83]). The synchronous composition operator $\times$ is different from the classical $\parallel$ of CCS.

The synchrony model is better suited for reasoning about concurrent actions than for implementing true concurrency. Because of the assumption of an eager behavior for the actions the scope of the obligations (and of the other deontic modalities too) is immediate, making them transient obligations which are enforced only in the current world. One can get persistent obligations by using temporal operators, like the *always* operator. The eagerness assumption facilitates both reasoning about existence of the deontic modalities and about violations of the obligations or prohibitions.

Regarding the dynamic logic part, $\mathcal{CL}$ introduces the synchrony operation $\times$ on the actions inside the dynamic modality. Therefore, $\mathcal{CL}$ can use dynamic logic reasoning about synchronous actions and, from this point of view, it is included in the class of extensions of PDL that can reason about concurrent actions: PDL $\cap$ with intersection of actions [Har83] which is undecidable for deterministic structures or concurrent PDL [Pel85, Pel87] which adopts ideas from alternating automata [CKS81]. Contrasting with the discouraging undecidability results from above, $\mathcal{CL}$ (with action complement and synchronous composition over deterministic actions inside the dynamic modality) is decidable. This makes $\mathcal{CL}$ more attractive for automation of reasoning about contracts.

(6) $\mathcal{CL}$ defines a conflict relation $\not\equiv_{\mathcal{CL}}$ over actions which represents the fact that two actions cannot be done at the same time. This is necessary for detecting (and for ruling out) a first kind of conflicts in contracts: “Obligatory to go west and obligatory to go east” should result in a conflict because the actions “go west” and “go east” cannot be done at the same time (i.e., are conflicting). The second kind of conflicts that $\mathcal{CL}$ rules out are: “Obligatory to go west and
forbidden to go west” which is a standard requirement on a deontic logic.

(7) In CL conditional obligations (or prohibitions) can be of two kinds.

a. The first kind is given with the propositional implication: \( C_1 \rightarrow O_C(\alpha) \) which is read as “if \( C_1 \) is the case then action \( \alpha \) is obligatory” (e.g., “If Internet traffic is high then the Client is obliged to pay”).

b. The second kind is given with the dynamic box modality: \([\beta]O_C(\alpha)\) which is read as “if action \( \beta \) was performed then action \( \alpha \) becomes obligatory” (e.g., “After receiving necessary data . . . the Provider is obliged to offer . . . password”).

(8) Regarding the deontic modalities, CL includes directly in the definition of the obligation and prohibition the reparations in case of violations. The deontic modalities are \( O_C \) and \( F_C \) where \( C \) is a contract clause representing the reparation. This models the notions of contrary-to-duty obligations (CTDs) and contrary-to-prohibitions (CTPs) as found in deontic logic applied over actions like DDL [Mey88, Wyn06]. These notions are in contrast with the classical notion of CTD as found in the SDL literature [PS97, CJ02]. In SDL, what we call reparations are secondary obligations which hold in the same world as the primary obligation. In our setting, where the action changes the context (the world), one can see a violation of an obligation (or prohibition) only after the action is performed and thus the reparations are enforced in the next world (in the changed context).

The approach of CL to contrary-to-duty rules out many of the problems faced by SDL (like the gentle murderer paradox). On the other hand it does not capture the wording of the SDL examples. Section 3.2.2 presents the stand of CL w.r.t. some of the most important paradoxes of SDL.

(9) Standard deontic logic SDL, and other variants of it, consider one of the three deontic modalities as primitive (usually \( O \) or \( P \)) and the other two modalities are defined in terms of this primitive one using the propositional operators. To the contrary, the deontic modalities are not interdefinable in CL. Only some of the implications that SDL makes hold in CL, and we discuss these in Section 3.1.1.

(10) The semantics of CL is given in terms of normative structures and it is specially defined to capture several natural properties which are found in legal contracts. These are motivated (with examples) in Section 3.2.

1.4 Summary of contributions

In this thesis I present the latest version of the CL logic, and related results. I summarize the main research contributions of this thesis in the following.

(1) We have developed an algebraic theory for synchronous regular actions by extending the Kleene algebra (i.e., the algebra of regular actions) with the notion of synchrony from R.Milner’s SCCS.²

²Ironic enough, at the same time when this work was accepted for JLAP, a similar work [HMSW09] was being accepted in CONCUR 2009. Now both these works cite each other but no clear cut relationship between them is known. We have tried in [Pri09b] to draw some relations and distinctions between the two algebraic formalisms.
(2) We have defined a dynamic deontic logic, $CL$, which combines several important notions for contracts. If some of this notions have appeared in different works before, as far as we know, their combination as done in $CL$ is not captured by other formalisms. Among these notions are:

a. An ought-to-do approach to deontic logic where deontic modalities are applied exclusively over actions;
b. Incorporation in the actions of the notion of synchrony to model “actions done at the same time”;
c. The actions are precisely formalized in a novel algebraic formalism which enjoys properties like completeness, decidability, and automata representations;
d. $CL$ incorporates the dynamic logic modality to reason about “what happens after an action is performed”, and to express useful temporal notions like “always” and “eventually”;
e. $CL$ has contrary-to-duties and contrary-to-prohibition syntactic constructs as first class citizens in the logic, allowing to express naturally reparations;

$CL$ respects many logical properties which have been difficult to capture together in the same logical semantics. Moreover, $CL$ avoids most of the standard deontic logic paradoxes.

(3) We present a fully automated method, based on a logical formalism, for monitoring at run-time an electronic/legal contract. The purpose of the monitoring machine is to signal a contract violation as soon as one happens. As far as we know this is the first work of this kind.

Just to give a taste of how the research on $CL$ developed, I will also give a short description, in chronological order, of the (published) work that lead to the present state of $CL$. All this work is only available in the respective publications (and not in this thesis).

[PS07a] is the first paper where we put many of the problems that $CL$ had to face (which are now part of the properties of $CL$) and we gave a first syntax for $CL$ and a semantics in an extension of $\mu$-calculus. There, already, $CL$ was aiming at modeling concurrent actions but the right notion of concurrency was not to come until later. Avoiding deontic paradoxes was a challenge too. This paper had an extended version with proofs and examples as the technical report [PS07c].

[PPS07] still used the variant of $\mu$-calculus from [PS07a] to show how model checking of legal contracts can be done. The merits of the paper were a thorough example of translating a natural language contract example into the formal language of $CL$ (we use the same contract example in this thesis; see Appendix B). Then it used the $\mu$-calculus translation to obtain a Kripke-like model which was fed into the NuSMV model checker. Model checking on this model was done w.r.t. properties specified also in $CL$. The paper shows how the counter examples of NuSMV can be used to repair the original English contract (and its $CL$ formalization) so that it passes all the properties. The work on model checking contracts was already presented in a workshop [PS07d].
[PS07b, Pri08b] are the initial works on the algebra of synchronous actions. These initial results were published in a student workshop affiliated to CALCO (the main conference on algebra in computer science) [Pri07]. Results that are closer to the synchronous Kleene algebra formalism of this thesis appeared towards the end of the year in [Pri08c]. The results as presented here appeared as [Pri09b, Pri09a] before the journal paper [Pri10].

[PS09a] was showing our work to the artificial intelligence and law community. This was based on the long technical reports [Pri08a, PS08] that were investigating, respectively, the deontic modalities in isolation over the synchronous actions, and the full combination with the dynamic logic modality as it is now in $\mathcal{CL}$. These efforts ended up in the international paper [PS09b] and in the journal draft [PS10].

[KPS08] was presenting the run-time monitoring technique at a moment where the full semantics of $\mathcal{CL}$ (based on normative structures as in this thesis) was not available yet. But the more simple trace semantics was possible.
Chapter 2

Synchronous Kleene Algebra

If time is precious for the reader, I strongly encourage to read first the journal paper [Pri10] on which this chapter is based. If more details are needed then these should be found in the following.

The work presented in this chapter investigates the combination of Kleene algebra with the synchrony model of concurrency from R. Milner’s SCCS calculus. The resulting algebraic structure is called synchronous Kleene algebra. Models are given in terms of sets of synchronous strings and finite automata accepting synchronous strings. The extension of synchronous Kleene algebra with Boolean tests is presented together with models on sets of guarded synchronous strings and the associated automata on guarded synchronous strings. Completeness w.r.t. the standard interpretations is given for each of the two new formalisms. Decidability follows from completeness. Kleene algebra with synchrony should be included in the class of true concurrency models. In this direction, a comparison with Mazurkiewicz traces is made which yields their incomparability with synchronous Kleene algebras (one cannot simulate the other). On the other hand, we isolate a class of pomsets which captures exactly synchronous Kleene algebras. We present an application to Hoare-like reasoning about parallel programs in the style of synchrony.

2.1 Preliminaries

Kleene algebra is a formalism used to represent and reason about programs. Kleene algebra with tests combines Kleene algebra with a Boolean algebra; it can express while programs [Koz00] and can encode propositional Hoare logic using a Horn-style inference system. In one form or another, Kleene algebras appear in various formalisms in computer science: relation algebras, logics of programs, in particular, Propositional Dynamic Logic [Pra79, Pra90], and regular expressions and formal language theory [KS86].

The present chapter investigates the extension of Kleene algebra with a particular notion of concurrent actions which adopts the synchrony model. Synchrony is a model of concurrency which was introduced in the process algebra community in R. Milner’s SCCS calculus [Mil83] but which detaches from the general interleaving approach. Synchrony is a concept which belongs to the partial orders model of concurrency [Maz88, NPW79, Pra86]. We see in Section 2.4 how synchrony as defined here compares to Mazurkiewicz traces [Maz88] and pomsets
The synchrony concept proves highly expressive and robust; SCCS can represent CCS (i.e., asynchrony) as a sub-calculus, and a great number of synchronizing operators can be defined in terms of the basic SCCS-Meije operators [dS85]. Meije is the calculus at the basis of the Esterel synchronous programming language [BC85]; Meije and SCCS operators are interdefinable.

The motivation for adding synchrony to Kleene algebra spawns from the need to represent and reason about actions which can be performed “at the same time”. We view this notion as being closer to the models of concurrency based on partial orders than to the ones based on interleaving. For reasoning about actions we choose the established equational formalism of Kleene algebra. For a faithful representation of the notion of “at the same time” in an equational setting the synchrony model is the most appealing. We do not need such powerful concurrency models like the ones based on partial orders; on the other hand, the low level interleaving model is not well suited for (abstract) reasoning about actions done at the same time (and related properties). The synchrony model has an equational representation and thus it is easy to integrate into Kleene algebra. Moreover, the reasoning power and expressiveness that synchrony offers is enough for the applications listed in Section 2.1.1.

This chapter defines two algebraic structures obtained from the combination of Kleene algebra with synchrony, and investigates theoretical aspects of them. Section 2.2 contains the synchronous Kleene algebra (SKA), which is Kleene algebra extended with a synchrony combinator for actions. The main difficulties in the axiomatization of SKA come from the definition of the synchrony operator and its relations with the other operators. We introduce sets of synchronous strings as the standard models for SKA and give the relation between the actions and these models. We prove completeness of the axiomatization w.r.t. the standard models. From completeness we get the decidability of the equality between actions. To prove completeness we need to define a new kind of automata which recognize sets of synchronous strings. For these automata we prove standard results which we need in the completeness proof. The most important is the equivalent of Kleene’s theorem which shows how to build an automaton for an action of SKA that accepts exactly the set of synchronous strings interpreting the action.

Besides the pure theoretical stimulation, the SKA formalism finds applications in deontic logic over synchronous actions and to propositional dynamic logic over synchronous actions. More precisely, the ∗-free actions of SKA (and related results) are used in giving a direct semantics to the action-based contract-specification language CL, as presented in Chapter 3.

Other applications of SKA can be found among the various places where Kleene algebra is used (some of which we state in Section 2.1.1) and where is involved a notion of concurrency for which the synchrony model of SKA is expressive enough. To evaluate the expressiveness of SKA we give in Section 2.4 comparisons with related concurrency models like Mazurkiewicz traces, pomsets, and concurrent Kleene algebras. The concluding section 2.5 contains more related work and open problems.

Section 2.3 contains the extension of synchronous Kleene algebra to include Boolean tests (SKAT) which follows the methodology of extending Kleene algebra with tests of [Koz97b]. Mainly, we extend SKA with a Boolean algebra defining the tests (called guards in the models). At the axiomatization level there are no considerable novelties w.r.t. SKA. More difficult is to find standard models; these we define as sets of guarded synchronous strings. In the completeness proof of SKAT we use again an automata theoretic argument and thus we define automata
to accept guarded synchronous strings. Here the operations over the automata are not standard, and care needs to be taken when defining the fusion product and the synchrony product. Using these, the equivalent of Kleene’s theorem leads the way to proving the completeness, and thus the decidability.

One of the standard applications of Kleene algebra with tests (KAT) is to reason about programs in a more general way than with standard Hoare logic. In the same line SKAT can be used to reason about concurrent programs with shared variables in the style of Owicki and Gries [OG76]. We present this application in Section 2.3.4.

The rest of this introductory section gives background material on Kleene algebra and synchrony, and can safely be skipped by an expert reader.

2.1.1 Applications

One application of synchronous Kleene algebras is in the context of the deontic logic of actions [vW68, Seg82] (underlying the semantics of the contract language CL) and propositional dynamic logic with synchronous actions. These applications to deontic and dynamic logics are not part of this chapter. The second application, in Section 2.3.4, presents SKAT as an alternative to Hoare logic for reasoning about parallel programs with shared variables in the synchrony style.

More generally, wherever one uses Hoare logic to reason about programs one can use the more powerful Kleene algebra with tests (the KAT-ML prover [AHK06a, Koz00] may be used to reason about programs in the style of Kleene algebra with tests). Similarly, one may safely choose SKAT when reasoning about concurrent executions is needed (similar to some extent with the current work of C.A.R. Hoare [Hoa07]). In these contexts (synchronous) Kleene algebra proves more powerful and more general than classical logical formalisms.

As other applications, we envisage the use of synchronous Kleene algebra to give semantics for Java threads and to give semantics to an extension of propositional dynamic logic (PDL) with synchrony. (In the same way as KAT is the underlying formalism for the programs (actions) of PDL, SKAT would be underlying the synchronous programs of this extension.) This would be an alternative to the PDL∩ [HKT00] or concurrent PDL [Pel85].

2.1.2 Kleene algebras

Kleene algebra (KA) was named after S.C. Kleene who in the fifties studied regular expressions and finite automata [Kle56]. Kleene algebra formalizes axiomatically these structures. Further developments on the algebraic theory of KA were done by J.H. Conway [Con71]. For references and an introduction to Kleene algebra see the extensive work of D. Kozen [Koz79, Koz90, Koz97b]. Completeness of the axiomatization of KA was studied in [Sal66, Koz94], complexity in [CKS96], and applications to concurrency control, static analysis and compiler optimization, or pointer arithmetics in [Coh94, KP00, Koz03b, Möl97]. Some variants of KA include the notion of tests [Koz97b], and others add some form of types [Koz98].

**Definition 2.1.1** An idempotent semiring is an algebraic structure \((A, +, \cdot, 0, 1)\) that respects axioms (1)-(9) of Table 2.1. A Kleene algebra \((A, +, \cdot, *, 0, 1)\) is an idempotent semiring with one extra unary (postfix) operation \(*\), which respects the extra axioms (10)-(13) of Table 2.1. We
(1) \( \alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma \)
(2) \( \alpha + \beta = \beta + \alpha \)
(3) \( \alpha + 0 = 0 + \alpha = \alpha \)
(4) \( \alpha + \alpha = \alpha \)
(5) \( \alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma \)
(6) \( \alpha \cdot 1 = 1 \cdot \alpha = \alpha \)
(7) \( \alpha \cdot 0 = 0 \cdot \alpha = 0 \)
(8) \( \alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma \)
(9) \( (\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma \)

Table 2.1: Axioms of Kleene algebra

understand the operations as representing respectively nondeterministic choice, sequence, and iteration. Henceforth we denote elements of \( A \) by \( \alpha, \beta, \gamma \), and call them (compound) actions. The constants 0 and 1 are sometimes called the fail action respectively the skip action. For an idempotent semiring the natural order \( \leq \) is defined as:

\[
\alpha \leq \beta \triangleq \alpha + \beta = \beta;
\]

and in this chapter we usually say that “\( \beta \) is preferable to \( \alpha \)”.

An intuitive understanding of the natural order of a semiring is that \( \leq \) states that the left operand has less behavior than the right operand, or in other words, the right operand specifies behavior which includes all the behavior specified by the left operand (and possibly more).

**Remarks:** It is easy to check that \( \leq \) is a partial order and that it forms a semilattice with least element 0 and with \( \alpha + \beta \) the least upper bound of \( \alpha \) and \( \beta \). Moreover, the three operators are monotone w.r.t. \( \leq \).

**Notation:** Fix a set of basic (or atomic) actions \( A_B \) (henceforth denoted by \( a, b, c \in A_B \)). Consider the corresponding term algebra \( T_{KA}(A_B) \) that is finitely generated from the generator set \( A_B \cup \{0, 1\} \). We use \( T_{KA} \) whenever \( A_B \) is understood from context. The syntactic terms of \( T_{KA} \) (the actions) can be seen as generated by the grammar:

\[
\alpha ::= a \mid 0 \mid 1 \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^*\]

where \( a \in A_B \). An acquainted reader may remark that \( T_{KA}(A_B) \) corresponds to the set of regular expressions over the alphabet \( A_B \).

The axioms (1)-(4) define the choice operator \( + \) to be associative, commutative, with neutral element 0, and idempotent. Axioms (5)-(7) define the sequence operator \( \cdot \) to be associative, with neutral element 1, and with annihilator 0 both on the left and right side. Axioms (8) and (9) give the distributivity of \( \cdot \) over \( + \).

The equations (10) and (11) and equational implications (12) and (13) are the standard axiomatization of * [Sal66, Koz94] which say that \( \alpha^* \cdot \beta \) is the least solution w.r.t. the preference relation \( \leq \) for the equation \( \beta + \alpha \cdot \mathcal{X} \leq \mathcal{X} \) (and dually \( \beta \cdot \alpha^* \) is the least solution to the equation \( \beta + \mathcal{X} \cdot \alpha \leq \mathcal{X} \)).

**Examples of Kleene algebras:** Consider, in language theory, \( \Sigma^* \) the set of all finite words over the alphabet \( \Sigma \) [HMU00]. The powerset \( \mathcal{P}(\Sigma^*) \) (i.e., the set of all languages) with the standard operations of union, concatenation, and Kleene star over languages forms a Kleene algebra.
For a second example consider the set of all binary relations over a set $X$. The powerset of $X \times X$ with the standard empty relation (for 0), identity relation (for 1), union of relations, relational composition, and the transitive and reflexive closure of a relation (for $\ast$) forms a Kleene algebra. This algebra is used in the semantics of logics of programs, like Propositional Dynamic Logic [FL77, HKT00].

As a last example consider the less known min,+ algebra (also called the tropical algebra) which is useful in shortest path algorithms on graphs [Koz97a]. The operations are defined over the domain $\mathbb{R}_+ \cup \{\infty\}$. The $+$ operation from Kleene algebra is defined as the $\min$ operation on reals giving the minimum of two elements under the natural order on $\mathbb{R}_+ \cup \{\infty\}$ where $\infty$ is always the greatest element. The operation $\cdot$ is interpreted as the standard arithmetic $+$ on $\mathbb{R}_+ \cup \{\infty\}$. The two constants 0 and 1 are interpreted respectively as $\infty$ and 0. The $\ast$ operation is surprisingly defined as $x^\ast = 0$.

Note that for the first two examples above the preference relation $\leq$ is defined to be set inclusion $\subseteq$ whereas for the last example it is the reverse of the natural order on reals.

### 2.1.3 Synchrony

The notion of synchrony has different meanings in different areas of computer science. Here we take the distinction between synchrony and asynchrony as presented in the SCCS calculus of [Mil83] and later implemented in, e.g., the Esterel synchronous programming language [BC85, BG92]. We understand asynchrony as the execution of two concurrent systems at independent relative speeds (i.e., their actions may have different non-correlated durations), whereas in the synchrony model each of the two concurrent systems execute instantaneously a single action at each time instant.

The synchrony model takes the assumption that time is discrete and that basic actions are instantaneous. Moreover, at each time step, all possible actions are performed, i.e., the system is considered eager and active (idling is not possible). Synchrony assumes a global clock which provides the time unit for all the actors in the system. For practical purposes this is a rather strong assumption which is in contrast with the popular view from process algebras [Mil95, Hoa85]. On the other hand, the equational framework of the synchrony model is much cleaner and more general than the asynchronous interleaving model; the well known CCS calculus [Mil95] is just a sub-calculus of the asynchronous version of SCCS (named ASCCS) [Mil83]. Moreover, the experience of the Esterel implementation and use in industry contradict the general belief.

Regarding expressiveness, the work of [dS85] establishes the relative completeness of expressiveness of the SCCS and Meije languages (i.e., they are equally expressive). Meije was the core language of the Esterel synchronous programming language, which is now widely used in industry.

SCCS introduces a synchronous composition operator $\times$ over processes which is different from the well known $\parallel$ of CCS (actually $\parallel$ can be defined in terms of $\times$). SCCS keeps the process algebra style of giving meaning to processes using structural operational semantics.
The operational semantics of $\times$ is:

$$
\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{a \times b} P' \times Q'}
$$

In this chapter we do not use structural operational semantics, but take an algebraic equa-
tional view in the style of Kleene algebras.

2.2 Extending Kleene Algebra with Synchrony

We add to the standard Kleene algebra of Section 2.1.2 an operator to model concurrency similar
to the synchronous composition of SCCS presented in Section 2.1.3. We call the resulting
algebra synchronous Kleene algebra and abbreviate it $SKA$. The $SKA$ algebra has the following
particularities:

a. It formalizes a notion of concurrent actions based on the synchrony model.

b. It has a standard interpretation of the actions as sets of synchronous strings. The actions
can also be represented as special finite automata which accept the same sets of synchronous
strings that form the models of the actions.

c. It incorporates a notion of conflicting actions.

This section (as well as the next) is concerned with the theoretical investigations of the new
algebra. The general motivations are given in the introduction and conclusion as well as through
the examples of applications from the end of each section, and through the comparisons that we
do in Section 2.4. Occasionally we give short intuitions for the particular notions presented and
suggest applications.

The investigation of synchronous actions in an algebraic setting implies that one should
consider them in the most general (and abstract) manner. Particular views of the actions can
be as human-like actions from legal contracts, as instructions in a programming language, or as
parallel executing processes.

2.2.1 Syntax and axiomatization

**Definition 2.2.1 (Synchronous Kleene Algebra)** A synchronous Kleene algebra ($SKA$) is a
structure $(A, +, \cdot, \times, *, 0, 1, A_B)$ obtained from a Kleene algebra by adding a “$\times$” operation for
synchronous composition of two actions. The new operation $\times$ respects the axioms (14)-(21) of
Table 2.2.

**Notation:** Consider the set $A_B^\times \subset A$ to be the set $A_B$ closed under application of $\times$. We
call the elements of $A_B^\times$-actions and denote them generically by $a\times$ (e.g., $a, a \times b \in A_B^\times$ but
$a + b, ab + c, a \cdot b \not\in A_B^\times$ and $0, 1 \not\in A_B^\times$). Note that $A_B^\times$ is finite because there is a finite number
of basic actions in $A_B$ which may be combined with the synchrony operator $\times$ in a finite number
of ways (due to the weak idempotence of $\times$ over basic actions; see axiom (18) of Table 2.2).
Note the inclusion of sorts $A_B \subseteq A_B^\times \subseteq A$. For brevity we often drop the sequence operator and
All axioms of Kleene algebra from Table 2.1

(14) \( \alpha \times (\beta \times \gamma) = (\alpha \times \beta) \times \gamma \)
(15) \( \alpha \times \beta = \beta \times \alpha \)
(16) \( \alpha \times 1 = 1 \times \alpha = \alpha \)
(17) \( \alpha \times 0 = 0 \times \alpha = 0 \)
(18) \( \alpha \times a = a \quad \forall a \in A_B \)
(19) \( \alpha \times (\beta + \gamma) = \alpha \times \beta + \alpha \times \gamma \)
(20) \( (\alpha + \beta) \times \gamma = \alpha \times \gamma + \beta \times \gamma \)
(21) \( (\alpha_x \cdot \alpha) \times (\beta_x \cdot \beta) = (\alpha_x \times \beta_x) \cdot (\alpha \times \beta), \forall \alpha, \beta \in A_B^+ \)

Table 2.2: Axioms of SKA

Instead of \( \alpha \cdot \beta \) we write \( \alpha \beta \). To avoid unnecessary parentheses we use the following precedence over the constructors: \(+ < \cdot < \times < \ast\).

Axioms (14)-(17) give the properties of \( \times \) to be associative, commutative, with identity element \( 1 \), and annihilator element \( 0 \) (i.e., \( \langle A, \times, 1, 0 \rangle \) is a commutative monoid with an annihilator element). Axioms (14) and (15) basically say that the syntactic ordering of actions in a \( \times \)-action does not matter. Axiom (18) defines \( \times \) to be weakly idempotent over the basic actions \( a \in A_B \).

Note that this does not imply that we have an idempotent monoid. Axioms (19) and (20) define the distributivity of \( \times \) over \(+\). From axioms (14)-(20) together with (1)-(4) we conclude that \( \langle A, +, \times, 0, 1 \rangle \) is a commutative and idempotent semiring (NB: idempotence comes from axiom (4), and the axiom (18) is just an extra property of the semiring).

At this point we give an informal intuition for the actions (elements) of \( A \): we consider that the actions are “done” by somebody (be that a person, a program, or an agent). One should not think exclusively of processes “executing” instructions as this is only one way of viewing the actions. Moreover, we do not discuss in this chapter operational semantics nor bisimulation equivalences (like is done in SCCS).

With this non-algebraic intuition of actions we can elaborate on the purpose of \( \times \), which models the fact that two actions are done at the same time. Doing actions at the same time should not depend on the syntactic ordering of the concurrent actions; thus the associativity and commutativity axioms (14) and (15) of \( \times \). Intuitively, if a component does a skip action \( 1 \) then this should not be visible in the synchronous action of the whole system (thus the axiom (16)); whereas, if a component fails (i.e., does action \( 0 \)) then the whole system fails (thus the axiom (17)).

Particular to \( \times \) is the axiom (18) which defines a weak form of idempotence for the synchrony operator. The idempotence is natural for basic actions but it is not desirable for complex actions. Take as example a choice action performed synchronously with itself, \( (a + b) \times (a + b) \). The first entity may choose \( a \) and the second entity may choose \( b \) thus performing the synchronous action \( a \times b \). Therefore, the complex action is the same as \( a + a \times b + b \) (by the distributivity axiom (19), the commutativity of \( \times \) and \(+\), idempotence of \( \times \) over basic actions (18), and idempotence of \(+\)).

Particular to our concurrency model is axiom (21) which synchronizes sequences of actions by working in steps given by the \( \cdot \) constructor. This encodes the synchrony model.

Note that there is no axiom relating the \( \times \) with the Kleene star. There is no need as the relation is done by the synchrony axiom (21) and the fact that \( \alpha^* = 1 + \alpha \cdot \alpha^* \) from the axioms.
of *. Moreover, when combining two repetitive actions synchronously $\alpha^* \times \beta^*$ the synchrony operator will go inside the * operator. This results in an action inside * which has a maximum of $|\alpha| \times |\beta|$ steps (i.e., number of - applications). This is strongly related to the dimension of the automaton constructed later in Theorem 2.2.26 to handle the $\times$ (the size of the synchronous automaton is the size of the cartesian product of the two smaller automata; i.e., it is the product of the number of states).

**Definition 2.2.2** Consider $SKA \vdash \alpha = \beta$ to mean that the SKA equation can be deduced from the axioms of SKA using the standard rules of equational reasoning (reflexivity, symmetry, transitivity, and substitution), instantiation, and introduction and elimination of implication. Consider henceforth the relation $\equiv \subseteq T_{SKA} \times T_{SKA}$ defined as: $\alpha \equiv \beta \iff SKA \vdash \alpha = \beta$.

**Remark:** The proof that $\equiv$ is a congruence is straightforward, based on the deduction rules, and we leave it to the reader.

**Definition 2.2.3** (demanding relation) We call $\prec \times$ the demanding relation and define it as:

$$\alpha \prec \times \beta \triangleq \alpha \times \beta = \beta.$$ 

(2.1)

We say that $\beta$ is more demanding than $\alpha$ iff $\alpha \prec \times \beta$. We denote by $\leq \times$ the relation $\prec \times \cup = (i.e., \alpha \leq \times \beta$ iff either $\alpha \prec \times \beta$ or $\alpha = \beta$).

The intuition is that an action $\beta$ is considered more demanding than another action $\alpha$ if we can see $\beta$ as doing all the actions in $\alpha$ and something more. Consider the following examples: $1 \prec \times a$, $a \prec \times a \times b$, $a \prec \times a$, $a \not\prec \times b$, $a \not\prec \times a + b$, $a + b \leq \times a + b$, and $a \not\prec \times b \times c$. Note that the least demanding action is $1$ (skipping means not doing any action). On the other hand, if we do not consider $1$ then we have the basic actions of $\mathcal{A}$ as the minimal demanding actions; the basic actions are not related to each other by $\prec \times$. We use $\prec \times$ mainly to compare $\times$-actions, where in this case, an $\alpha_{\times} \in \mathcal{A}_{\times}$ can be seen as the set of basic actions that compose it and, hence, $\prec \times$ becomes just $\subseteq$.

**Proposition 2.2.4** The relation $\prec \times \big|_{A_{\times}}$ is a partial order.

**Proof:** For the relation $\prec \times$ restricted to $\times$-actions the reflexivity is assured by the weak idempotence axiom (18) together with (14) and (15). The transitivity and antisymmetry are immediate and moreover, they hold for the whole set $\mathcal{A}$ of actions; e.g., for transitivity take any $\alpha, \beta, \gamma \in \mathcal{A}$ s.t. $\alpha \prec \times \beta$ and $\beta \prec \times \gamma$. Then it is the case that from $\alpha \times \beta = \beta$ and $\beta \times \gamma = \gamma$ we get $\alpha \times \gamma = \alpha \times \beta \times \gamma = \beta \times \gamma = \gamma$ which is the desired conclusion $\alpha \prec \times \gamma$ (we used associativity of $\times$ and transitivity of the equality of actions).

**Corollary 2.2.5** The relation $\leq \times$ is a partial order for $\mathcal{A}$.

**Proof:** Transitivity and antisymmetry were proven in Proposition 2.2.4 and reflexivity follows from the definition of $\leq \times$. $\square$
The conclusion of the two results above is that $<_x$ is not a partial order for the whole set of actions (as opposed to the natural order $\leq$ of Kleene algebra). It is a partial order only when is restricted to $\times$-actions (the weak idempotence axiom (18) is used). On the other hand, when $<_x$ is explicitly extended with equality we get a partial order $\leq_x$ for the general actions.

Because $\times$ lacks idempotence for general actions we loose some monotonicity properties. In contrast to the natural order $\leq$, the operators $+$ and $\cdot$ are not monotone w.r.t. $<_x$. The next result proves some weak monotonicity properties.

**Proposition 2.2.6**

a. If $\alpha_x^i <_x \beta_x^j$ for all $1 \leq i \leq n$, then $\alpha_x^1 \cdots \alpha_x^n <_x \beta_x^1 \cdots \beta_x^n \cdot \gamma$

where $\alpha_x^i, \beta_x^j \in \mathcal{A}_B$ and $\gamma \in \mathcal{A}$.

b. If $\alpha_x^i <_x \beta_x^j$ for all $i \leq n$ and $j \leq m$, then $(\alpha_x^1 + \cdots + \alpha_x^n) <_x (\beta_x^1 + \cdots + \beta_x^m)$.

**Proof:** For the first part of the proposition, the hypothesis is translated to $\alpha_x^i \times \beta_x^j = \beta_x^j$ for all $1 \leq i \leq n$. We need to prove that $(\alpha_x^1 \cdots \alpha_x^n) \times (\beta_x^1 \cdots \beta_x^n \cdot \gamma) = \beta_x^1 \cdots \beta_x^n \cdot \gamma$.

The synchrony axiom (21) can be applied on the left part of the equality to combine the $\alpha_x^i$ and $\beta_x^j$ synchronously two by two which, by the hypothesis, become $\beta_x^j$. We obtained exactly $\beta_x^1 \cdots \beta_x^n \cdot \gamma$.

The proof of the second part of the proposition is similar. It makes use of the distributivity axiom (19) first, then uses the hypothesis in the same manner as before, and finally it contracts the same actions $\beta_x^j$ with the axiom (4) of idempotence of $+$. \hfill \Box

Note that reflexivity for $<_x$ is not a property of the general actions, but only of the $\times$-actions. Therefore, irreflexivity of $<_x$ is not a property of the general actions either. On the other hand, we give some weaker results related to $<_x$ applied to, somehow, more complex actions. Corollary 2.2.14 shows that for any $\ast$-free action $\alpha$ there exists a fixed point for the application of the $\times$ to the action itself. More precisely, define $\beta_0 = \alpha$ and $\beta_i = \beta_{i-1} \times \alpha$, then $\exists n \in \mathbb{N}$ and $\exists j < n$ s.t. $\beta_j = \beta_n$. This means that $\alpha <_x \beta_n$ for any $n \geq j$. For example $(a + b) \times (a + b) = a + b + a \times b$ but $(a + b + a \times b) \times (a + b) = a + b + a \times b$ (mainly due to the weak idempotence of the $\times$ over $\mathcal{A}_B$). The proof uses the canonical representation of the $\ast$-free action $\alpha$.

**Definition 2.2.7 (conflict and compatibility)** Consider a symmetric and irreflexive relation over the set of basic actions $\mathcal{A}_B$, which we call the conflict relation and denote by $\#_C \subseteq \mathcal{A}_B \times \mathcal{A}_B$. The complement relation of $\#_C$ is the symmetric and reflexive compatibility relation which we denote by $\sim_C$ and define as:

$$
\sim_C \triangleq \mathcal{U}_B \setminus \#_C
$$

where $\mathcal{U}_B = \mathcal{A}_B \times \mathcal{A}_B$ is the universal relation over basic actions.

When talking about actions done at the same time (i.e., synchronously) then it is natural to think about actions which cannot be done at the same time. For the basic actions this information is considered to be given a priori (by an oracle) and formalized as the conflict relation on

---

1The $\ast$-free actions are defined in the next Section 2.2.2.
The notion of conflicting actions (and hence conflict relation) is natural and often found in legal contracts. The intuition of the conflict relation is that if two actions are in conflict then the actions cannot be done synchronously. This intuition explains the need for the following equational implication:

\[(22) \quad a \#_c b \rightarrow a \times b = 0 \quad \forall a, b \in \mathcal{A}_B.\]

The intuition of the compatibility relation is that if two actions are compatible then the actions can always be performed synchronously. Compatibility is obtained from the assumption that what is not in conflict is compatible. There is no transitivity of \(\#_c\) or \(\sim_c\); in general an action \(b\) may be in conflict with both \(a\) and \(c\) but still \(a \sim_c c\) (i.e., not necessarily \(a \#_c c\)). Consider the following natural counterexample from the legal domain for transitivity: action “drive” may be in conflict with both “drink” and “talk at the phone” but still one can “drink and talk at the phone” at the same time.

### 2.2.2 Deontic actions as \(\ast\)-free actions

The canonical form (defined below) gives us a more structured way of viewing the \(\ast\)-free actions and makes easier the formulation and proof of the two important results of Corollary 2.2.14 and Corollary 2.2.16 and the results leading to it. Motivations and intuitive meaning for these results are given at the appropriate place where they appear.

**Definition 2.2.8 (\(\ast\)-free actions)** We call \(\ast\)-free actions the terms of \(T_{SKA}\) constructed with the grammar below:

\[
\alpha \ ::= \ a \mid 0 \mid 1 \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha
\]

where \(a \in \mathcal{A}_B\) is a basic action. Denote the set of all \(\ast\)-free actions by \(\mathcal{A}_D\). In later sections we call \(\ast\)-free actions deontic actions because they will be used only inside the deontic operators of the \(CL\) logic.

**Definition 2.2.9 (canonical form)** We say that an action \(\alpha\) is in canonical form, denoted by \(\hat{\alpha}\), iff it is 0, 1, or has the following form:

\[
\hat{\alpha} \triangleq \sum_{i \in I} \alpha^i \cdot \alpha^i
\]

where

- a. \(\alpha^i \in \mathcal{A}_B^\ast\);
- b. \(\alpha^i \neq \alpha^j\) for all \(i, j \in I\);
- c. for all \(i \in I\), either
  - c.1. \(\alpha^i \in \mathcal{A}^\ast \setminus \{0, 1\}\) is an action in canonical form or
  - c.2. \(\alpha^i\) does not exist and in this case \(\alpha^i \in \mathcal{A}_B^\ast \cup \{1\}\) is allowed to be also 1;
Note that $\alpha^i$ does not contain twice the same action $a$, for any $a \in A_B$ and $i \in I$ as it is part of $A_B^i$, thus, $a \times a$ is not in canonical form, but $a \times b$ is. The indexing set $I$ is finite as $A_B^i$ is finite and $\alpha^i$ are different. Consider the example $\alpha = (a + b) \cdot (c + d)$ which is not in canonical form but is equivalent to the canonical form $\alpha = a \cdot (c + d) + b \cdot (c + d)$ (apply axiom (9)). On the other hand $\alpha$ is equivalent also to $a \cdot c + a \cdot d + b \cdot c + b \cdot d$ (apply (9) then (8)) which is not in canonical form because the constraint $b$, in Definition 2.2.9 is not met as $a$ appears twice as first element in the summation.

**Theorem 2.2.10** For every $*$-free action $\alpha \in A^D$ there is corresponding $\alpha$ in canonical form and equivalent to $\alpha$ (i.e., $SKA \vdash \alpha = \alpha'$).

**Proof:** We use structural induction on the structure of the actions of $A^D$ given by the constructors of the algebra. In the inductive proof we take one case for each action construct. The proof also makes use of the axioms of SKA. For convenience in the presentation of the proof, we define, for an action in canonical form $\alpha$, the set $R = \{\alpha^i | i \in I\}$ to contain all the $*$-actions on the first “level” of $\alpha$. Thus, we often use in the proof the alternative notation for the canonical form $\alpha = \overset{R}{\land} \alpha^i \cdot \alpha^i$ which emphasizes the exact set of $*$-actions on the first level of the action $\alpha$. Often the $R$ in the notation is omitted for brevity.

**Base case:**

a) Basic actions $a$ of $A_B$ are, by definition, in canonical form; i.e., action $a$ is in canonical form with the set $R = \{a\}$ and the $\cdot$-constructor is applied to $a$ and to skip action $1 (\overset{\{a\}}{+} a \cdot 1 \equiv a)$.

b) The special actions $1$ and $0$ are, by definition, in canonical form.

In the inductive step we consider only one step of the application of the constructors; the general compound actions follow from the associativity of the constructors.

**Inductive step:**

a) Consider $\alpha = \beta + \beta'$ is a compound action obtained by applying once the $+$ constructor. By the induction hypothesis $\beta$ and $\beta'$ are equivalent to canonical forms: $\beta \equiv \overset{b_i \in B}{\lor} b_i \cdot \beta_i$ and $\beta' \equiv \overset{b_j' \in B'}{\lor} b_j' \cdot \beta_j'$. Because of the associativity and commutativity of $+$, $\beta + \beta'$ is also in canonical form:

$$\beta + \beta' \equiv \overset{b_i \in B}{\lor} b_i \cdot \beta_i + \overset{b_j' \in B'}{\lor} b_j' \cdot \beta_j' = \overset{a \in B \cup B'}{\lor} a \cdot \beta_a$$

where $a$ and $\beta_a$ are related as follows: whenever $b_i = b_j' = a$ then $\beta_a = \beta_i + \beta_j'$ (i.e., this is the case for common elements of $B$ and $B'$); otherwise if $a = b_i$ then $\beta_a = \beta_i$, or if $a = b_j'$ then $\beta_a = \beta_j'$. Because the inductive hypothesis states that all $\beta_i$ and $\beta_j'$ are in canonical form it follows that also $\beta_a$ (which is either just a change of notation or a choice of two smaller canonical forms $\beta_i + \beta_j'$) are in canonical form.

b) Consider $\alpha = \beta \cdot \beta'$ with $\beta \equiv \overset{b_i \in B}{\lhd} b_i \cdot \beta_i$ and $\beta' \equiv \overset{b_j' \in B'}{\lhd} b_j' \cdot \beta_j'$ in canonical form. We now make use of the distributivity of $\cdot$ over $+$, and of the associativity of the $\cdot$ and $+$ constructors. In several steps $\alpha$ is transformed into a canonical form. In the first step $\alpha$ becomes

$$\alpha = \beta \cdot \beta' \equiv (\overset{b_i \in B}{\lor} b_i \cdot \beta_i) \cdot (\overset{b_j' \in B'}{\lor} b_j' \cdot \beta_j')$$
and, considering $|B| = m$, by the distributivity axiom (9), $\alpha$ becomes:

$$\alpha \equiv b_1 \cdot \beta_1 \cdot (\sum_{b'_j \in B'} ( + b'_j \cdot \beta'_j)) + \ldots + b_m \cdot \beta_m \cdot (\sum_{b'_j \in B'} ( + b'_j \cdot \beta'_j)).$$

Subsequently $\cdot$ is distributed over all the members of the choice actions using axiom (8). In the end $\alpha$ becomes a choice of sequences, when we consider $|B| = m$ and $|B'| = k$:

$$\alpha \equiv b_1 \cdot \beta_1 \cdot b'_1 \cdot \beta'_1 + \ldots + b_m \cdot \beta_m \cdot b'_k \cdot \beta'_k.$$ 

This is clearly a canonical form because all actions $\beta_i \cdot b'_j \cdot \beta'_j$ are equivalent to canonical forms due to the inductive hypothesis. For the special case when $\beta_i = 1$ axiom (6) is applied to contract it to $b'_1 \cdot \beta'_j$.

c) Consider $\alpha = \beta \times \beta'$ with $\beta \equiv +_{b_i \in B} b_i \cdot \beta_i$ and $\beta' \equiv +_{b'_j \in B'} b'_j \cdot \beta'_j$ in canonical form. First we use the distributivity axioms of $\times$ over $+$ and assume $|B| = m$ and $|B'| = k$, and from

$$\alpha \equiv ( +_{b_i \in B} b_i \cdot \beta_i) \times ( +_{b'_j \in B'} b'_j \cdot \beta'_j)$$

we get

$$\alpha \equiv (b_1 \cdot \beta_1) \times ( +_{b'_j \in B'} b'_j \cdot \beta'_j) + \ldots + (b_m \cdot \beta_m) \times ( +_{b'_j \in B'} b'_j \cdot \beta'_j),$$

which distributes more to

$$\alpha \equiv (b_1 \cdot \beta_1) \times (b'_1 \cdot \beta'_1) + \ldots + (b_m \cdot \beta_m) \times (b'_k \cdot \beta'_k).$$

By applying the synchrony axiom (21) each summand is equivalent to

$$(b_i \cdot \beta_i) \times (b'_j \cdot \beta'_j) \equiv b_i \times b'_j \cdot \beta_i \cdot \beta'_j.$$ 

Clearly $b_i \times b'_j$ is a $\times$-action and the inductive hypothesis applied to the smaller $\beta_i$ and $\beta'_j$ yields that $\beta_i \times \beta'_j$ is equivalent to a canonical form. This means that each summand is equivalent to a canonical form, making $\alpha$ equivalent to a canonical form too.

\[\square\]

Corollary 2.2.11 says that $\ast$-free actions can be seen as choices of sequences of synchronous actions of $A_D^j$; or, in other words, as sets of words over $A_D^j$ (or as sums of products over $A_D^j$). This is the syntactic formulation of what we see later as being the standard models of SKA.

**Corollary 2.2.11 (normal form for $\ast$-free actions)** For an arbitrary $\ast$-free action $\alpha$ there exists an equivalent action $\beta \in A^D$ (i.e., $\beta \equiv \alpha$) which is of the following form:

$$( +_{i \in I} \cdot \alpha^i_j).$$

**Proof:** The corollary basically says that any $\ast$-free action $\alpha$ is equivalent to a canonical form $\alpha$ which is equivalent to an action that is a choice of sequences of synchronous actions $\alpha_i^j$. 

This is a simple consequence of the Theorem 2.2.10. Take the canonical form $\alpha = \bigoplus_{i \in I} \alpha^i$. A simple inductive argument suffices: the base case is trivial. For the inductive case consider

$$\alpha^i = \bigoplus_{j \in J, k \in K} \alpha^j_x,$$

and just distribute the $\alpha^i_x$ to obtain

$$+ \bigoplus_{j \in J, k \in K} \alpha^j_x \cdot \alpha^k_x.$$

The associativity of $+$ finishes the proof. \(\square\)

**Lemma 2.2.12** For any action that has the form of a sequence of synchronous actions, $\alpha = \alpha_1 \cdot \ldots \cdot \alpha^m$, then $\alpha \times \alpha = \alpha$. In other words, $\times$ is idempotent and $\preceq_\times$ is reflexive for actions of this form.

**Proof:** The proof is trivial by using the synchrony axiom and the weak idempotence for $\times$. \(\square\)

**Theorem 2.2.13** For any *-free action $\alpha$ there exists $n \in \mathbb{N}$, which depends on $\alpha$, for which $\alpha \preceq_\times \alpha^n$, where $\alpha^n$ denotes the action obtained by putting $n$ copies of $\alpha$ in synchronous combination.

**Proof:** By Corollary 2.2.11 we consider $\alpha = \bigoplus_{i \in I} \bigoplus_{j \in J} \alpha^i_j$. Claim: $n = |I|$. We prove the claim using induction on $|I|$. 

_Base case:_ $|I| = 1$, means there is only one summand. Lemma 2.2.12 proves the case; i.e., because $\alpha = \bigoplus_{j \in J} \alpha^i_j$ then $\alpha \times \alpha = \alpha$ which is $\alpha \preceq_\times \alpha$.

_Inductive step:_ consider $\alpha = \alpha_k + \beta$ where $\alpha_k = \bigoplus_{k \in K} \alpha^k_x$ is just a sequence of synchronous actions and $\beta = \bigoplus_{i \in I} \bigoplus_{j \in J} \alpha^i_j$ with $|I| = n$ for which the induction hypothesis applies, and says that $\beta \preceq_\times \beta^n$. The induction hypothesis translates to $\beta \times \beta^n = \beta^n = \beta^{n+1} = \beta^{n+2} = \ldots$. We have to prove that $\alpha \preceq_\times \alpha^{n+1}$. In other words, we need to prove that $\alpha^{n+2}$ is idempotent. But $\alpha^{n+2} = \alpha \times \alpha^n = \alpha \times \alpha \times \alpha^{n-1} = (\alpha_k + \beta) \times (\alpha_k + \beta) \times \alpha^{n-1}$. Using distributivity and commutativity of $\times$, idempotence of $+$, and Lemma 2.2.12 for $\alpha_k$ we obtain $(\alpha_k + \alpha_k \times \beta + \beta^2) \times (\alpha_k + \beta) \times \alpha^{n-2} = (\alpha_k + \alpha_k \times \beta + \alpha_k \times \beta^2 + \beta^3) \times (\alpha_k + \beta) \times \alpha^{n-3}$. In the end we obtain $\alpha^{n+1} = \alpha_k + \alpha_k \times \beta + \alpha_k \times \beta^2 + \ldots + \beta^{n+1}$. Therefore, we have to prove that $\alpha_k + \alpha_k \times \beta + \alpha_k \times \beta^2 + \ldots + \alpha_k \times \beta^{n+1} = \alpha_k + \alpha_k \times \beta + \alpha_k \times \beta^2 + \ldots + \alpha_k \times \beta^n + \beta^{n+1}$.

But this is easy using the induction hypothesis and the idempotence of $+$ to contract summands which are the same. \(\square\)

To motivate the next corollary, recall that SKA does not have general idempotence for $\times$, but only idempotence over basic actions (this is opposed to the general idempotence that $\oplus$ has). The next corollary is a fix-point result, which is the closest we can get to a property like idempotence. The corollary says that for a general *-free action if we combine it with itself using the $\times$ operator several times we are bound to reach a fix-point where we will obtain the same action no mater home many more times we do the combination. The idempotence property can be understood like a fix-point property where the combination can be stopped after only one application.
Corollary 2.2.14 (fix-point for \( \times \) application) For any \( \ast \)-free action \( \alpha \in A^D \) there exists \( n \in \mathbb{N} \) for which for all \( m \geq n \) it holds that \( \alpha \times n = \alpha \times m \).

In rewriting theory, to apply an axiom means to apply the rule obtained from directing the axiom, in our case we direct the axioms from left to right; see [BN98] or Appendix A for details on rewriting theory.

Theorem 2.2.15 For a canonical form \( \underline{\alpha} = +_{i \in I} \alpha^i \cdot \alpha^i \) only axiom (8) can be applied (and none other of the axioms of Table 2.2), modulo associativity and commutativity.

Proof: Note first that we work modulo associativity and commutativity of \( + \) and \( \times \), and modulo associativity for \( \cdot \) (thus we do not consider axioms (1), (2), (5), (14), (15)). The remaining axioms of Table 2.2 are considered directed from left to right.

Axiom (3) is not applicable because \( \alpha^i \) cannot be 0 and neither can \( \underline{\alpha}^i \) because of Definition 2.2.9-c.(a). Axiom (4) is dealt with by the condition in Definition 2.2.9-b. The left part of axiom (6) cannot be applied because of the constraint in Definition 2.2.9-c.(a) which makes \( \underline{\alpha}^i \neq 1 \). The right part of (6) is not applicable because \( \alpha^i \neq 1 \) when \( \underline{\alpha}^i \) exists. Similar arguments using the c.(a) constraint again show that axiom (7) is not applicable. Clearly, axiom (9) is not applicable to a canonical form. Axioms (16) and (17) are dealt with by the fact that \( \alpha^i \) from \( A^D \) (contain only basic actions). Axiom (18) is taken care of by the same argument. The main purpose of the canonical form is to make sure that the axioms for \( \times \) (like (17), (19), (20), (21)) are applied exhaustively to the original action, and cannot be applied any more to the canonical form. In other words the axioms (19), (20), (21) push the \( \times \) inside the action until it reaches the basic actions.

The following corollary is immediate from the proof of Theorem 2.2.15 (applying axiom (8) to a canonical form breaks the canonicity). The importance of this corollary is multiple; uniqueness of the canonical form is a nice property in itself, but more, any functional operation that is applied to a canonical form is bound to return a unique result. This is the case later with the action complement. Therefore, we are assured that by complementing an action (i.e., its equivalent canonical form) we obtain a unique action in return, which captures exactly and uniquely what it means to “not do an action”.

Corollary 2.2.16 (uniqueness of the canonical forms for \( \ast \)-free actions) The canonical form of an action \( \alpha \) is unique (modulo associativity and commutativity).

For the deontic modalities, one important notion is action complement. In our view action complement encodes the violation of an obligation (as we see later in Section 3.1). Intuitively, we say that the complement \( \pi \) of action \( \alpha \) is the action given by all the immediate actions which take us outside the tree of \( \alpha \). This view was aired in [BWM01] but no formal definition was given. In our case we need to deal with synchronous actions too. The notion of action complement that we give shares ideas with [Mey88] but it is not restricted to respect all the axioms of [Mey88]; we want action complement to capture naturally what it means to violate an obligation in an eager system (where no idling is possible). With \( \alpha \) it is easy to formally define \( \pi \).
Definition 2.2.17 (action complement) The action complement is denoted by $\overline{\alpha}$ and is defined as a function $\overline{\cdot} : \mathcal{A}^D \to \mathcal{A}^D$ (i.e., action complement is not a principal combinator for the actions) and works on the equivalent canonical form $\alpha$ as:

$$\overline{\alpha} = + \sum_{i \in I} \alpha_i^i \cdot \overline{\beta}_i \triangleq + \beta \sum_{j \in J} \gamma_j^j \cdot + \sum_{i \in I'} \gamma_i^i \cdot$$

Consider $R \triangleq \{ \alpha_i^i \mid i \in I \}$. The set $\overline{R}$ contains all the $\times$-actions $\beta_i$ with the property that none of the actions $\alpha_i^i$ are included in $\beta_i$:

$$\overline{R} \triangleq \{ \beta_i \mid \beta_i \in \mathcal{A}_B^\alpha \text{ and } \forall i \in I, \alpha_i^i \not\subseteq \beta_i \};$$

and $\gamma_j^j \in \mathcal{A}_B^\alpha$ and $\exists \alpha_i^i \in R \text{ s.t. } \alpha_i^i \subseteq \gamma_j^j$. The indexing set $I_j' \subseteq I$ is defined for each $j \in J$ as:

$$I_j' \triangleq \{ i \in I \mid \alpha_i^i \subseteq \gamma_j^j \}.$$  

Complement of 1 is $0 = \overline{1}$ and complement of 0 is $1 = \overline{0}$. 

The complement operation formalizes the fact that an action is not performed. In an eager system this boils down to either not performing any of its immediate actions $\alpha_i^i$, or by performing one of the immediate actions and then not performing the remaining action. Note that to perform an action $\alpha_i^i$ means to perform any action that includes $\alpha_i^i$. Therefore in the complement we may have actions $\gamma_j^j$ which include more immediate actions, e.g. $\alpha = a \cdot b + c \cdot d$ and may perform $\gamma_j^j = a \times c$. At this point we need to look at both actions $b$ and $d$ in order to derive the complement, e.g. performing now $d$ means that $\alpha$ was done, whereas performing $c$ means that $\alpha$ was not done (and $a \times c \cdot c$ must be part of complement).

The following result states that our notion of action complement always produces an action in canonical form.

Proposition 2.2.18 The complement operation returns a (finitely described) deontic action which is in canonical form.

Proof: For the first part of the proposition we prove that $\overline{\alpha}$ has no infinite application of the $+$ constructor (we say “no infinite branching”) and also no infinite application of the $\cdot$ constructor (we say “no infinite depth”). In both cases we use induction on the structure of the action complement. The basis of the induction is clearly satisfied as 0, 1, and all $a \in \mathcal{A}_B$ have both finite branching and finite depth.

$\overline{R}$ is finite because $\overline{R} \subseteq \mathcal{A}^*_B$, where $\mathcal{A}^*_B$ is finite, and thus $+_{\beta_i \in \overline{R}} \beta_i$ is finitely branching. The indexing set $J$ is finite (having maximum size $|\mathcal{A}^*_B|$) thus $+_{j \in J} \gamma_j^j$ is finitely branching. Lastly, the indexing sets $I_j'$ are finite subsets of the $I$, hence $+_{i \in I'} \alpha_i^i$ is a subaction of $\alpha$. Thus we apply the induction hypothesis to it and deduce that its complement $+_{i \in I'} \overline{\alpha_i^i}$ is finitely branching, for any $I_j' \subseteq I$. We conclude that $\overline{\alpha} = +_{\beta \in \overline{R}} \beta \cdot +_{j \in J} \gamma_j^j \cdot +_{i \in I'} \overline{\alpha_i^i}$ is finitely branching.

It remains to prove that $\overline{\alpha}$ has no infinite depth. The first part of the action complement (i.e., $+_{\beta \in \overline{R}} \beta$) introduces only branches of finite depth. For the second part (i.e., $+_{j \in J} \gamma_j^j \cdot +_{i \in I'} \overline{\alpha_i^i}$) we can apply the induction hypothesis to $+_{i \in I'} \overline{\alpha_i^i}$ because, as we discussed before, this is a
subaction of \( \alpha \). Thus, we have that each \( \sum_{i \in I} \alpha_i \) has finite depth. These are concatenated to the \( \gamma'_j \) actions which have depth 1, thus, making all the second choice of finite depth, and hence the \( \tau \) has finite depth.

For the second part of the proposition it is easy to see that the action complement respects the canonical form. Action complement is a choice of sequences, each sequence being either a single \( \times \)-action (i.e., from \( \sum_{i \in I} \beta_i \)) or an \( \times \)-action \( \gamma'_j \) followed by another action \( \sum_{i \in I} \alpha_i \) which we know by induction that is in canonical form.

\[ \square \]

### 2.2.3 Standard interpretation over synchronous sets

We give the standard interpretation of the actions of \( A \) by defining a homomorphism \( \hat{I}_{SKA} \) which takes any action of the \( SKA \) algebra into a corresponding synchronous set and preserves the structure of the actions given by the constructors.

**Definition 2.2.19 (synchronous sets)** We consider a finite set denoted \( A_B \) (which for our discussion can be thought of as the basic actions). Consider a finite alphabet \( \Sigma = \mathcal{P}(A_B) \setminus \{\emptyset\} \) consisting of all nonempty subsets of \( A_B \) (denote them \( x, y \in \Sigma \)). Synchronous strings over \( \Sigma \) are elements of \( \Sigma^* \) including the empty string \( \epsilon \) (denote them \( u, v, w \in \Sigma^* \)). A synchronous set is a subset of synchronous strings from \( \Sigma^* \) (denoted by \( A, B, C \)). Consider the following definitions and operations on synchronous sets:

\[
\begin{align*}
0 & \triangleq \emptyset \\
1 & \triangleq \{\epsilon\} \\
A + B & \triangleq A \cup B \\
A \cdot B & \triangleq \{uv \mid u \in A, v \in B\} \\
A \times B & \triangleq \{u \times v \mid u \in A, v \in B\} \\
A^* & \triangleq \bigcup_{n \geq 0} A^n
\end{align*}
\]

where \( uv \) is the concatenation of the two synchronous strings \( u \) and \( v \), and \( u \times v \) is defined as:

\[
\begin{align*}
u \times \epsilon & \triangleq u \triangleq \epsilon \times u \\
u \times v & \triangleq (x \cup y)(u' \times v') \text{ where } u = xu' \text{ and } v = yv',
\end{align*}
\]

and where \( x, y \in \Sigma \) are sets of elements of \( A_B \). The powers \( A^n \) are defined as:

\[
\begin{align*}
A^0 & \triangleq \{\epsilon\} \\
A^n & \triangleq A \cdot A^{n-1}.
\end{align*}
\]

By convention when \( A = \emptyset \) then \( A^* = \{\epsilon\} \); thus \( A^* \) always contains the empty string \( \epsilon \). This operation is called the Kleene star.

**Notation:** Recall from formal languages the convention \( u \epsilon = u = \epsilon u \). We abuse the notation and write \( a \) instead of the singleton set \( \{a\} \) (also write \( a \in \Sigma \)). Moreover, we consider any subset of \( A_B \) as a synchronous string of length 1 and sometimes write \( a \) or \( x \) instead of \( u \) when the intention is clear from the context.
Theorem 2.2.20 Any set of synchronous sets containing 0 and 1 and closed under the operations $+, \cdot, \times, ^*$ of Definition 2.2.19 is a synchronous Kleene algebra and forms a subalgebra of the powerset synchronous Kleene algebra of $\Sigma^*$.

Proof: Routine check that the operations of Definition 2.2.19 obey the axioms of SKA from Table 2.2. Particular care needs to be taken for axioms (18) and (21) as they are defined on particular elements. Axiom (18) is defined only on the singleton sets $\{a\}$ with $a \in A_B$ whereas axiom (21) is defined only on singleton synchronous sets $\{x\}$ with $x \in \Sigma$.

First we check that the full powerset of $\Sigma^*$ is a SKA. It is easy to see that it is closed under $+, \cdot$, and $^*$. It is also closed under $\times$ because if we take any two synchronous sets $A$ and $B$ then, by Definition 2.2.19, $A \times B$ is a set of strings where each element of a string is of the form $x \cup y \in \Sigma$ for $x, y \in \Sigma$; i.e., each element of $A \times B$ is a synchronous string.

We now prove that the powerset algebra the operations of Definition 2.2.19 satisfy the SKA axioms of Table 2.2. The proofs for $+, \cdot$, and $^*$ are standard. In short, the $+$ operation over synchronous sets respects axioms (1)-(4) because it is defined in terms of the union operation $\cup$ over sets and 0 is defined as the empty set. The $\cdot$ operation is defined as in formal language theory and the proofs that it respects (5)-(9) are standard as we also have the convention $u \epsilon = u$ for the 1 case. For these cases and for the $^*$, the fact that we work with synchronous strings makes no difference.

For the associativity axiom (14) of $\times$ we prove $u \in A \times (B \times C) \iff u \in (A \times B) \times C$, for any $A, B, C \subseteq \Sigma^*$. We prove the forward implication (the backward implication follows a similar reasoning). From the definition we have that $u = u_A \times (u_B \times u_C)$ with $u_A \in A, u_B \in B, u_C \in C$. We consider the general case in the definition of $\times$ over synchronous strings (where the particular case for $\epsilon$ follows from the proof of axiom (16)), and thus, $u_A = x_A u'_A, u_B = x_B u'_B$, and $u_C = x_C u'_C$. We have $x_A u'_A \times ((x_B \cup x_C)(u'_B \times u'_C)) \overset{\text{Def.}}{=} (x_A \cup (x_B \cup x_C))(u'_A \times (u'_B \times u'_C))$. Because $\cup$ for sets is associative and for the $u'$ strings we follow an inductive argument we have that $u = ((x_A \cup x_B) \cup x_C)((u'_A \times u'_B) \times u'_C)$ which is the same as $(u_A \times u_B) \times u_C$; i.e., $u \in (A \times B) \times C$.

For the commutativity axiom (15) the proof is similar as above and it rests on the observation that the $\times$ operation of synchronous strings is commutative because it uses the set union $\cup$ at each element of the string.

For axioms (16) and (17) the proof comes from the definitions of 1 and 0 respectively. Consider $u \in A \times \{\epsilon\}$; then, by definition, $u = u_A \times \epsilon$ which, by the special case in the definition of $\times$ on synchronous strings, is equal to $u_A \in A$. For 0 it is clear from the definition of $\times$ on synchronous sets that $A \times \emptyset = \emptyset$.

The special axiom (18) applies only to singleton synchronous sets $\{a\}$ with $a \in A_B$. It is easy to see that $\{a\} \times \{a\} = \{a\}$ because $a \cup a = a$ (i.e., $\cup$ over sets is idempotent).

The argument for the distributivity axioms (19) and (20) is similar to that for the $\cdot$ and $+$ operations. Consider $u \in A \times (B + C); \text{ then, by definition, } u = u_A \times u'$ with $u' \in B + C$. This means that either $u = u_A \times u_B$ or $u = u_A \times u_C$. This is the same as $u \in (A \times B) + (A \times C)$.

The synchrony axiom (21) the proof makes use of the fact that we need to consider only the special singleton synchronous sets of the form $\{x\}$ with $x \in \Sigma$. We need to prove $u \in \{x\} \times (\{y\} \times B)$ iff $u \in \{(x \times \{y\}) \times \{(x \times B)\}$ for any two arbitrary singleton synchronous sets $\{x\}$ and $\{y\}$. By definition $u = xu_A \times yu_B = (x \cup y)(u_A \times u_B)$ for arbitrary $u_A \in A$ and
Most interesting is the smallest such algebra which contains $u_2$ and all $\{a\}$ for $a \in A_B$, where $A_B$ is some set which is finite and fixed beforehand. Denote this algebra by $\text{ASS}$.

**Definition 2.2.21 (standard interpretation)** An interpretation of $\text{SKA}$ is a homomorphism with domain the term algebra $T_{\text{SKA}}$. We call standard interpretation the homomorphism $\hat{I}_{\text{SKA}} : T_{\text{SKA}} \rightarrow \text{ASS}$. $\hat{I}_{\text{SKA}}$ is defined as the homomorphic extension of the map $I_{\text{SKA}} : A_B \cup \{0, 1\} \rightarrow \text{ASS}$ to the whole set of actions $T_{\text{SKA}}$. $I_{\text{SKA}}$ maps the generators of $T_{\text{SKA}}$ into synchronous sets as follows:

$$
I_{\text{SKA}}(a) = \{ \{a\} \}, \forall a \in A_B
$$

$$
I_{\text{SKA}}(0) = \emptyset
$$

$$
I_{\text{SKA}}(1) = \{ \}^*
$$

The homomorphic extension is standard:

$$
\hat{I}_{\text{SKA}}(\alpha) = I_{\text{SKA}}(\alpha), \forall \alpha \in A_B \cup \{0, 1\}
$$

$$
\hat{I}_{\text{SKA}}(\alpha + \beta) = \hat{I}_{\text{SKA}}(\alpha) + \hat{I}_{\text{SKA}}(\beta)
$$

$$
\hat{I}_{\text{SKA}}(\alpha \cdot \beta) = \hat{I}_{\text{SKA}}(\alpha) \cdot \hat{I}_{\text{SKA}}(\beta)
$$

$$
\hat{I}_{\text{SKA}}(\alpha^\cdot) = \hat{I}_{\text{SKA}}(\alpha)^*
$$

The standard interpretation offers a method (i.e., a deterministic algorithm) for obtaining a model (as a set of synchronous strings) for an action of $\text{SKA}$. Just implement $\hat{I}_{\text{SKA}}$ as a recursive function on the structure of the actions stopping at the generators of $T_{\text{SKA}}$. From here the operations of $\text{ASS}$ are applied upwards to generate the synchronous set corresponding to the initial action. Consider the example$^2$: for $a, b, c \in A_B$, $\hat{I}_{\text{SKA}}(a \times b \cdot c) = \hat{I}_{\text{SKA}}(a \times b) \cdot \hat{I}_{\text{SKA}}(c) = (\hat{I}_{\text{SKA}}(a) \times \hat{I}_{\text{SKA}}(b)) \cdot \hat{I}_{\text{SKA}}(c) = (\{a\} \times \{b\}) \cdot \{c\} = \{a, b\} \{c\}$.

The standard models are the linking factor (semantically) between the syntactic elements of the algebra (i.e., between the equivalent actions). Moreover, the standard models are objects which are closer to our intuition, as sets of synchronous strings, and thus it is easier to work with (and compare) them.

Intuitively the skip action 1 means not performing any action and its interpretation as the set with only the empty string, which contains no basic action, goes well with the intuition. The fail action 0 is interpreted as the empty set following the intuition that there is no way of respecting a fail action.

Consider a $\times$-action $\alpha_x = a_1 \times \ldots \times a_n$ with $a_i \in A_B$ for all $1 \leq i \leq n$. The standard interpretation interprets $\alpha_x$ as a singleton set $\hat{I}_{\text{SKA}}(\alpha_x) = \{\{a_1, \ldots, a_n\}\}$ where the only string $w = \{a_1, \ldots, a_n\}$ has just one element of the alphabet $\Sigma$ (i.e., one set of basic actions which form the $\times$-action $\alpha_x$). Henceforth we denote sets like $\{a_1, \ldots, a_n\}$, coming from a $\times$-action $\alpha_x$, by $\{\alpha_x\}$. We use this notation, instead of just a general $x \in \Sigma$, when we want to make more explicit the set $x$. Moreover, we may apply set union to mean $\{\alpha_x\} \cup \{\beta_x\} = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$.

$^2$Recall the precedence of the operators $+, <, \cdot, \times, \prec$. 

$u_B \in B$. On the right hand side of the implication we have that $u = (x \cup y)(u_A \times u_B)$ for arbitrary $u_A \in A$ and $u_B \in B$. This completes the proof. 

From the proof, it is clear that any subalgebra of the powerset algebra of $\Sigma^*$ is an $\text{SKA}$. We use this notation, instead of just a general $\times$-action $\alpha_x$, by $\{\alpha_x\}$. We use this notation, instead of just a general $x \in \Sigma$, when we want to make more explicit the set $x$. Moreover, we may apply set union to mean $\{\alpha_x\} \cup \{\beta_x\} = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$.
2.2.4 Completeness and decidability

Regular expressions and finite automata (FA) are equivalent syntactic representations of regular languages (or regular sets as we call them) [HMU00]. In this section we define finite automata that accept synchronous sets and prove an equivalent of Kleene’s theorem. That is, for each action of $T_{SKA}$ we can build a corresponding automaton which accepts the same synchronous set as the interpretation of the action. Using the translation as automata we give a combinatorial proof of completeness of the $SKA$ w.r.t. the standard interpretation. Decidability follows from completeness and from the decidability of the inclusion problem for regular languages.

Definition 2.2.22 (automata on synchronous strings) Nondeterministic finite automata on synchronous strings (NFA) are tuples $A = (S, \Sigma, S_0, \rho, F)$ consisting of a finite set of states $S$, the finite alphabet of synchronous actions $\Sigma = P(A_B \setminus \{\emptyset\})$ (i.e., the powerset of the set of basic actions $A_B$ minus the empty set), a set of initial designated states $S_0 \subseteq S$, a transition function $\rho : \Sigma \rightarrow P(S \times S)$, and a set of final states $F$. An NFA is called deterministic (DFA) iff $|S_0| = 1$ and the transition function returns, for each label, not a relation over $S$ but a partial function, i.e., $\rho : \Sigma \rightarrow (S \rightarrow S)$.

Notation: We denote the states of an automaton by either $s_i \in S$ or $i \in S$ with $i \in \mathbb{N}$. We often use $s_0$ to stand for the initial state and $s_f$ to stand for a final state. Instead of a transition $(s_1, s_2) \in \rho(\{\alpha\})$ we often use the graphical notation $s_1 \xrightarrow{\alpha} s_2$ or the tuple notation $(s_1, \{\alpha\}, s_2)$. A transition $s \xrightarrow{\epsilon} s'$ is called an $\epsilon$-transition. The $\epsilon$ is not part of the alphabet $\Sigma$ and thus it does not contribute to the accepted strings of an automaton. The label $\epsilon$ allows an automaton to take a transition without accepting any new symbol. This is useful later to give nice definitions of operations on automata. Further on we use set union over labels of the transitions; the special case for the $\epsilon$ should be understood as $\{\alpha\} \cup \epsilon = \{\alpha\}$. By the definition of $\rho$, NFA and DFA may not have $\epsilon$-transitions (also called $\epsilon$-free automata). In the sequel we need to talk about automata that do have $\epsilon$-transitions and call them $\epsilon$-NFA.

Definition 2.2.23 (acceptance) A run of an automaton $A$ is a finite sequence of transitions starting in the initial state, i.e., $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_n} s_n$. A run is called accepting if it ends in a final state, i.e., $s_n \in F$. An automaton accepts a string $w$ iff there exists an accepting run s.t. $\{\alpha_1\}\{\alpha_2\} \ldots \{\alpha_n\} = w$. The set of strings accepted by an automaton forms the language accepted by the automaton, denoted $L(A)$ or just $A$.

Our definition of NFA (and DFA) differs from the standard definition [HMU00] in the choice of alphabet (we have sets of basic actions as labels) and the new definition of a synchrony product for these automata, which we see later. Because of this, many of the standard results for NFA and DFA which do not depend on the choice of alphabet can be adapted to our automata on synchronous strings.

Proposition 2.2.24 ([HMU00])

a. DFAs and NFAs recognize the same class of languages; i.e., for any NFA there is a determinization procedure to generate a DFA which accepts the same language (and any DFA is also an NFA).
For any DFA one may use a Myhill-Nerode minimization procedure to obtain a minimal and unique DFA which accepts the same language as the initial automaton.

c. For any NFA we can construct an equivalent NFA which has only one final state and no transitions starting from the final state.

d. For any ε-NFA we can construct (using the standard ε-closure construction) an ε-free NFA which will accept the same language.

e. For any NFA with unique final state and no transitions starting from the final state we can remove all the ε-transitions which do not end in the final state (using a variation of ε-closure which does not consider the final state as part of the closures), and the resulting automaton will accept the same regular set.

Because of the results of Proposition 2.2.24 we are free to work with NFA with one initial state, possibly one final state with no outgoing transitions, and which may have ε-transitions only ending in the final state; we denote the class of these automata by $A_S$. Representatives of this class are denoted $A_S^S(\alpha)$ when they are related to $\alpha$ (because, generally, these are the automata generated in Theorem 2.2.26 for some particular action $\alpha$). This kind of automata facilitate the definition of the synchrony product operation on automata, corresponding to $\times$.

Corollary 2.2.25 (unique minimal automaton) For any automaton on synchronous sets there exists a unique minimal deterministic automaton accepting the same synchronous set.

Theorem 2.2.26 (actions to automata) For any action $\alpha \in SKA$ we can construct an automaton $A^S(\alpha)$ which accepts precisely $\hat{I}_{SKA}(\alpha)$.

Proof: The proof is adapted from [HMU00] for the regular expressions operators (+, ·, and *) and we add a new construction for the synchrony operator $\times$. We use induction on the structure of actions.
Extending Kleene Algebra with Synchrony

Figure 2.2: Automata corresponding to $\alpha + \beta$, $\alpha \cdot \beta$, and $\alpha^*$. 

**Base case:** For each action $a \in \mathcal{A}_B$, 0, and 1 we build the automata from Fig. 2.1 respectively (i), (ii), and (iii). It is easy to check that these automata are of $A^S$ type and that they accept the corresponding synchronous sets.

**Inductive step:** For actions of the form $\alpha + \beta$, $\alpha \cdot \beta$, and $\alpha^*$ we use the standard constructions [HMU00] pictured in Fig. 2.2 respectively (i), (ii), and (iii). These automata are constructed from the smaller automata corresponding to the smaller actions $\alpha$ and $\beta$ and it is easy to check that they accept precisely the corresponding synchronous sets. Note that the automata of Fig. 2.2 are not of $A^S$ type, therefore, after each operation we need to apply Proposition 2.2.24(e) to remove the unwanted $\epsilon$-transitions (e.g., in Fig. 2.2(i) states 0, 1, 3 collapse into one, call it 013, and all transitions of the form $1 \xrightarrow{\alpha} i$ or $3 \xrightarrow{\alpha} i$ are replaced by a transition $013 \xrightarrow{\alpha} i$, and the two $\epsilon$-transitions are removed).

The new construction is the one for actions of the form $\alpha \times \beta$ which is schematically pictured in Fig. 2.3. The automaton $A^S(\alpha \times \beta)$ is constructed from the two smaller automata $A^S(\alpha) = (S_\alpha, \mathcal{P}(\mathcal{A}_B^\alpha) \setminus \{\emptyset\}, s_1, \rho_\alpha, s_2)$ which accepts $I_{SKA}(\alpha)$ and $A^S(\beta) = (S_\beta, \mathcal{P}(\mathcal{A}_B^\beta) \setminus \{\emptyset\}, s_3, \rho_\beta, s_4)$ which accepts $I_{SKA}(\beta)$, as follows:

$$A^S(\alpha \times \beta) = (S_\alpha \times S_\beta, \mathcal{P}(\mathcal{A}_B^\alpha \cup \mathcal{A}_B^\beta) \setminus \{\emptyset\}, (s_1, s_3), \rho_{\alpha\beta}, (s_2, s_4)).$$

Note that states of $A^S(\alpha \times \beta)$ are pairs of states of the old automata. Therefore, the initial state is the pair of the two initial states $(s_1, s_3)$ and the final state is the pair of the old final states $(s_2, s_4)$. The new transition relation is:

$$((s_i^\alpha, s_j^\beta), \gamma, (s_k^\alpha, s_l^\beta)) \in \rho_{\alpha\beta} \text{ iff either:}$$

1. $\exists \gamma_1, \gamma_2, \text{s.t. } (s_i^\alpha, \gamma_1, s_k^\alpha) \in \rho_\alpha \text{ and } (s_j^\beta, \gamma_2, s_l^\beta) \in \rho_\beta \text{ and } \gamma_1 \cup \gamma_2 = \gamma$, or
2. $(s_i^\alpha, \gamma, s_k^\alpha) \in \rho_\alpha \text{ and } s_j^\beta = s_l^\beta = s_4$, or
3. $(s_j^\beta, \gamma, s_l^\beta) \in \rho_\beta \text{ and } s_i^\alpha = s_k^\alpha = s_2$.

\[\text{In Fig. 2.3 we picture only an example where there are no loop transitions for the initial states.}\]
The intuition behind this construction is that whenever both smaller automata can make a move then the new automaton can also make a move but labeled with the union of the two labels of the smaller automata. The last two cases are for when one of the states in the pair is a final state of one of the original automata. This is because the new automaton should be able to make a move whenever one of the smaller automata has stopped in a final state and the other automaton can still make a move. This behavior captures the application of synchrony to two synchronous strings of different lengths (the shorter one being accepted by the automaton that stops first). The new automaton $A^{S}(\alpha \times \beta)$ has size $|S_{\alpha}| \times |S_{\beta}|$.

We need to prove now that the automaton $A^{S}(\alpha \times \beta)$ accepts exactly the synchronous strings of the synchronous set $I_{SKA}(\alpha \times \beta)$. We know that $I_{SKA}$ is a homomorphism, thus $I_{SKA}(\alpha \times \beta) = I_{SKA}(\alpha) \times I_{SKA}(\beta)$, and from the inductive hypothesis we know that $A^{S}(\alpha)$ accepts exactly $I_{SKA}(\alpha)$ and that $A^{S}(\beta)$ accepts exactly $I_{SKA}(\beta)$. Therefore we need to prove the following double implication:

\[ w \in A^{S}(\alpha \times \beta) \iff \exists u \in A^{S}(\alpha) \text{ and } \exists v \in A^{S}(\beta) \text{ s.t. } u \times v = w. \]

For the $\iff$ implication we assume that:

a. there exists an accepting run of $A^{S}(\alpha)$ for $u$, i.e., $s_{0}^{\alpha} \xrightarrow{\alpha_{1}} s_{1}^{\alpha} \xrightarrow{\alpha_{2}} s_{2}^{\alpha} \ldots \xrightarrow{\alpha_{n}} s_{n}^{\alpha}$ with $u = \{\alpha_{1}\} \{\alpha_{2}\} \ldots \{\alpha_{n}\}$;

b. there exists an accepting run of $A^{S}(\beta)$ for $v$, i.e., $s_{0}^{\beta} \xrightarrow{\beta_{1}} s_{1}^{\beta} \xrightarrow{\beta_{2}} s_{2}^{\beta} \ldots \xrightarrow{\beta_{m}} s_{m}^{\beta}$ with $v = \{\beta_{1}\} \{\beta_{2}\} \ldots \{\beta_{m}\}$;

c. $m \geq n$, and

d. $w = (\{\alpha_{1}\} \cup \{\beta_{1}\}) (\{\alpha_{2}\} \cup \{\beta_{2}\}) \ldots (\{\alpha_{n}\} \cup \{\beta_{n}\}) \{\beta_{n+1}\} \ldots \{\beta_{m}\}$.

From the construction of $A^{S}(\alpha \times \beta)$ we need to find an accepting run (i.e., starting in $(s_{0}^{\alpha}, s_{0}^{\beta})$ and ending in $(s_{n}^{\alpha}, s_{m}^{\beta})$) for the string $w$. It is easy to see that there are the following transitions in $\rho_{\alpha \beta}$: $(s_{i-1}^{\alpha}, s_{i-1}^{\beta}) \xrightarrow{\alpha_{i} \cup \beta_{i}} (s_{i}^{\alpha}, s_{i}^{\beta})$, for $0 < i \leq n$, forming a run which starts in the initial state of $A^{S}(\alpha \times \beta)$ and ends in $(s_{n}^{\alpha}, s_{m}^{\beta})$ which accepts the first part of $w$. Because $s_{n}^{\alpha}$ is the final state of $A^{S}(\alpha)$ then, from the construction of $\rho_{\alpha \beta}$, for each transition $s_{j}^{\beta} \xrightarrow{\beta_{j+1}} s_{j+1}^{\beta}$, with $n \leq j < m$, in $A^{S}(\beta)$ there is in $A^{S}(\alpha \times \beta)$ the following transition $(s_{n}^{\alpha}, s_{j}^{\beta}) \xrightarrow{\beta_{j+1}} (s_{n}^{\alpha}, s_{j+1}^{\beta})$, with $n \leq j < m$. These transitions form a run in $A^{S}(\alpha \times \beta)$ continuing the previous run and stopping when reaching a final state because final states have no outgoing transitions.\footnote{$A^{S}$ automata stop when reaching a final state because final states have no outgoing transitions.}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{automaton_construction.png}
\caption{Example of automaton construction corresponding to $\alpha \times \beta$.}
\end{figure}
-ending in the state \((s^α_n, s^β_m)\) which is the final state of \(A^S(α × β)\), and it accepts the second part of \(w\). Thus, we have found an accepting run of \(A^S(α × β)\) over the whole string \(w\). Note that the assumption that \(m ≥ n\) is without loss of generality; if we were to take the opposite assumption then we would have worked in the automaton \(A^S(α)\) for the second part of \(w\) and not in \(A^S(β)\) as we did by now.

For \(⇒\) we assume that there exists an accepting run of \(A^S(α × β)\) for \(w = \{w^1_x\} \ldots \{w^m_x\}\); let that be \((s^α_0, s^β_0) \xrightarrow{\{w^1_x\}} \ldots \xrightarrow{\{w^m_x\}} (s^α_n, s^β_m)\). By the construction of \(A^S(α × β)\) from the smaller automata \(A^S(α)\) and \(A^S(β)\) we know that \(s^α_n\) is final state for \(A^S(α)\) and \(s^β_m\) is final state for \(A^S(β)\). The choice of indices is wlog. We need to show that there are two accepting runs for \(u ∈ A^S(α)\) and \(v ∈ A^S(β)\) such that \(u × v = w\).

Consider a first case when \(s^α_0 = s^α_n\) (wlog we work in \(A^S(α)\)). Because \(A^S\) automata have no outgoing transitions from the final state then it implies that, in our run, \(s^α_i = s^α_0\) for all \(0 ≤ i ≤ n\) and \(A^S(α)\) accepts only \(ε\); thus \(u = ε\). Moreover, from the construction of \(ρ_{αβ}\) we conclude that there exist the transitions \(s^γ_i \xrightarrow{\{w^i_x\}} s^γ_{i+1} ∈ ρ_{αβ}\) for all \(0 ≤ i < m\). Therefore we find in \(ρ_{β}\) an accepting run (ending in \(s^β_m\)) over \(v = \{w^1_x\} \ldots \{w^m_x\}\). It is clear that \(u × v = ε × \{w^1_x\} \ldots \{w^m_x\} = w\).

Consider now the remaining case when \(s^α_0 ≠ s^α_n\) and \(s^β_0 ≠ s^β_m\) (i.e., neither \(s^α_0\) nor \(s^β_0\) are final states). Therefore, for the first transition \((s^α_0, s^β_0) \xrightarrow{\{w^1_x\}} (s^α_1, s^β_1)\) we are in the first case of the construction of \(ρ_{αβ}\) and thus, there exist the transitions \(s^α_0 \xrightarrow{\{w^1_x\}} s^α_1 ∈ ρ_{α}\) and \(s^β_0 \xrightarrow{\{v^1_x\}} s^β_1 ∈ ρ_{β}\) s.t. \(\{w^1_x\} = \{w^α_x\} \cup \{v^1_x\}\). We continue with the subsequent transitions for \(w\) and find similar subsequent transitions for \(v\) until we reach a case like: \((s^α_{n-1}, s^β_{n-1}) \xrightarrow{\{w^α_n\}} (s^α_n, s^β_n)\) where \(s^α_n\) is the final state of \(A^S(α)\). (Note that we make, wlog, the assumption that we reach first the final state of \(A^S(α)\); an analogous reasoning as the one below would work if we take the opposite assumption that we reach the final state of \(A^S(β)\) first.) In this case we have found the accepting run for \(u = \{w^1_x\} \ldots \{w^n_x\}\). Moreover, if \(s^β_n = s^β_m\) (i.e., we reach at the same time the final state of \(A^S(β)\)) then we also found an accepting run for \(v = \{v^1_x\} \ldots \{v^n_x\}\) and the proof is finished.

Consider now that \(s^β_n ≠ s^β_m\). Because \(s^α_n\) is final state there is no outgoing transition from it, and thus the only way to continue with the run for \(w\) is to have transitions of the form \((s^α_n, s^β_j) \xrightarrow{\{w^j_{x+1}\}} (s^α_n, s^β_{j+1})\) with \(n ≤ j < m\). Each of these transitions yields a transition for \(v\) in \(A^S(β)\): \(s^β_j \xrightarrow{\{w^j_{x+1}\}} s^β_{j+1}\). This process stops in the state \((s^α_n, s^β_m)\) and therefore the run for \(v\) stops in the final state \(s^β_m\) of \(A^S(β)\). We have found the accepting run for \(v = \{v^1_x\} \ldots \{v^n_x\}\). From the reasoning it is clear that \(w = u × v\).

In [Koz94] the completeness of Kleene algebra is proven by appealing to the representation of finite automata with matrices over arbitrary Kleene algebras. The most important construction is the * operation over matrices which basically gives the regular expressions encoding the regular languages accepted when going from each state of the automaton to every other state; a construction which comes from J.H. Conway [Con71]. In essence this construction is the algebraic equivalent of the combinatorial procedure of transforming an NFA into a regular expression [HMU00]. In the algebraic approach to automata the regular language accepted by the automaton is obtained as (the interpretation of) a single regular expression.
The proof of completeness that we give follows similar ideas except that it uses a combinatorial argument. The motivation is that when giving semantics to deontic logic over synchronous actions or to the extension of PDL with synchrony we use the automata associated to actions, thus a combinatorial argument is more clarifying in this direction. The algebraic approach with matrices over synchronous Kleene algebras is based on definitions of operations on matrices corresponding to the operations on automata that we gave in Theorem 2.2.26.

We make use of the procedure of eliminating states, which generates a regular expression from an NFA [HMU00]. Adapting the method of eliminating states to our automata on synchronous strings is trivial. Consider this method (which we denote $\mathcal{E}$) as a function which takes an automaton on synchronous strings $A^S_\alpha$ and returns an action $\alpha$ of SKA s.t. $\hat{I}^{SKA}(\alpha) = \mathcal{L}(A^S_\alpha)$. Moreover, $\mathcal{E}$ considers the automata to have as labels actions from SKA instead of elements of the alphabet [HMU00]. We consider the reader familiar with this standard technique for finite automata.

**Lemma 2.2.27** For all $\alpha \in T_{SKA}$ we have $\alpha \equiv \mathcal{E}(A^S(\alpha))$.

**Proof:** The proof of the lemma uses induction on the structure of the action. Moreover, we consider that $\mathcal{E}$ returns, instead of an action $\gamma$, an automaton with one initial and one final state, and one transition from the initial to the final state labeled by $\gamma$. (It is easy to see how $\mathcal{E}$ returns the corresponding $\gamma$ from such an automaton.) This helps in the inductive reasoning below.

**Base case:** take the actions 0, 1, and $a \in A_B$ and thus consider the $A^S$ automata of Fig. 2.1. It is easy to see that the $\mathcal{E}$ procedure returns the regular expressions 0, 1, and $a \in A_B$ respectively.

**Inductive step:**

**Case 1:** for $\alpha = \alpha_1 + \alpha_2$. The automaton $A^S(\alpha_1 + \alpha_2)$ is obtained with the construction for $+$ from Fig. 2.2(i) of Theorem 2.2.26 from the two automata $A^S(\alpha_1)$ and $A^S(\alpha_2)$. The inductive hypothesis says that $\alpha_1 \equiv \mathcal{E}(A^S(\alpha_1))$; and we consider that $\mathcal{E}$ returns the automaton pictured in Fig. 2.4(i), and similar for $\alpha_2$. It is easy to see how the automaton $A^S(\alpha_1 + \alpha_2)$ is...
transformed into the one of Fig. 2.4(ii) by application of $\mathcal{E}$ to the smaller automata $A^S(\alpha_1)$ and $A^S(\alpha_2)$. From here $\mathcal{E}$ first eliminates states 1 and 2 to obtain a transition labeled with the action $1 \cdot \alpha_1 \cdot 1 \equiv \alpha_1$ and then states 3 and 4 to obtain a transition labeled with $1 \cdot \alpha_2 \cdot 1 \equiv \alpha_2$. Then it contracts the two resulting transitions into one labeled with $\alpha_1 + \alpha_2 \equiv \mathcal{E}(A^S(\alpha_1 + \alpha_2))$ (as in Fig. 2.4(iii)).

**Case 2:** for $\alpha = \alpha_1 \cdot \alpha_2$. The automaton $A^S(\alpha_1 \cdot \alpha_2)$ is obtained from the two automata $A^S(\alpha_1)$ and $A^S(\alpha_2)$ as in Theorem 2.2.26. Using the same inductive hypothesis as in Case 1 the procedure $\mathcal{E}$ is first applied to the two smaller automata and ends up with the automaton on the left of Fig. 2.5(i). By further eliminating the middle states we obtain $\mathcal{E}(A^S(\alpha_1 \cdot \alpha_2)) = \alpha_1 \cdot 1 \cdot \alpha_2 \equiv \alpha_1 \cdot \alpha_2$.

**Case 3:** for $\alpha = \alpha_1^*$. Take the automaton of Fig. 2.2(iii) which is constructed from $A^S(\alpha_1)$ using the $^*$ operation from Theorem 2.2.26. By the same inductive hypothesis as in Case 1 we consider that $\mathcal{E}$ is first applied on $A^S(\alpha_1)$ obtaining an automaton as in Fig. 2.5(ii), left. After this, the $\epsilon$-elimination transforms it into a proper $A^S$ from which $\mathcal{E}$ returns $\alpha_1^*$. Note that $\mathcal{E}$ could have worked directly on the automaton with no $\epsilon$-elimination and the result would have been the same (i.e., $\mathcal{E}(A^S(\alpha_1^*)) = 1 \cdot \alpha_1 \cdot (1 \cdot \alpha_1)^* \cdot 1 + 1 \equiv \alpha_1 \cdot \alpha_1^* + 1 \equiv \alpha_1^*$).

**Case 4:** for $\alpha = \alpha_1 \times \alpha_2$. The proof for this case is more involved due to the local nature of the $\times$ operation. This local behavior can be seen both in the synchrony axiom (21) and in the definition of $\times$ over automata in Theorem 2.2.26 where the new transition relation is obtained by looking at each individual transition of the two smaller automata.

In short, the proof of this case uses again an inductive argument on the structure of the two actions $\alpha_1$ and $\alpha_2$. Essentially, we have to reason step by step on actions $\alpha_x \in \mathcal{A}_B^*$ because of the local behavior of $\times$, but we reason inductively.

**Base case:** $\alpha_1 = a$ and $\alpha_2 = b$. We again consider the automata $A^S$ to have actions as labels, and instead of doing union of sets of basic actions we apply $\times$ (i.e., instead of $\{a\} \cup \{b\}$ we now have $a \times b$). The automaton $A^S(a \times b)$ is obtained from the two smaller automata $A^S(a)$ and $A^S(b)$ as in Fig. 2.3 on page 30. It is easy to see that $\mathcal{E}(A^S(a \times b))$ returns the action $a \times b$.

For the rest of the base case consider $\alpha_1$ or $\alpha_2$ to be one of 0 or 1. The conclusion follows similarly as above.
From the base case is easy to see that we can reason with $\alpha_\times \in A^*_B$ actions as our basis, instead of just basic actions $a, b \in A_B$. This is because the base case generates a $\times$-action, and if we apply it several times we get exactly all the $\times$-actions. In other words we can use an induction argument only using basic actions and the base case, and we prove the conclusion for $\times$-actions. Therefore, we may take the $\times$-actions as our basis. We do so in the rest of the proof.

**Inductive step:** We fix $\alpha_2 = \beta_\times$ with $\beta_\times \in A^*_B$, and take cases after $\alpha_1$. This is the local behavior of the proof; it works one step at a time (i.e., one transition at a time, labeled with $\times$-actions). When considering more complex actions for $\alpha_2$ we still work with this assumption and treat the rest of the action inductively. Each time, the same cases as below need to be treated.

**Case 4.1:** for $\alpha_1 = \alpha_\times \cdot \gamma$. The automaton $A^S(\alpha_\times \times \alpha_2)$ is obtained from the two smaller automata $A^S(\alpha_\times \cdot \gamma)$ and $A^S(\beta_\times)$ as in Fig. 2.6. The inductive hypothesis tells that $E$ applied to the smaller automaton $A^S(\gamma)$ returns one transition labeled with $\gamma$. Further applying $E$ on the node $(2, 5)$ gives $\alpha_\times \times \beta_\times \cdot \gamma$. Note that the rest of the nodes do not play a role for $E$ in generating the final action.

**Case 4.2:** for $\alpha_1 = \gamma_1 + \gamma_2$. For the $+$ operator we can reason about general actions because of the distributivity properties of the $\times$ on automata over $+$, in which case $\times$ simply considers the two sets of transitions from the two smaller automata separately. The argument is based on the case 4.1 before.

The automaton $A^S(\alpha_\times \times \alpha_2)$ is built from the two smaller automata $A^S(\gamma_1 + \gamma_2)$ and $A^S(\beta_\times)$. The automaton $A^S(\gamma_1 + \gamma_2)$ is constructed as a disjoint union of the two automata $A^S(\gamma_1)$ and $A^S(\gamma_2)$. Therefore, when combined synchronously with $A^S(\beta_\times)$ the transition relation of $A^S((\gamma_1 + \gamma_2) \times \beta_\times)$ is constructed independently for the nodes of $A^S(\gamma_1)$ and for the nodes of $A^S(\gamma_2)$. Moreover, the nodes are separated into two disjoint parts. It is simple to see that this automaton comes from the disjoint union of $A^S(\gamma_1 \times \beta_\times)$ and $A^S(\gamma_2 \times \beta_\times)$. We now use Case 4.1 to apply $E$ on each of these two disjoint automata to obtain transitions labeled respectively with $\gamma_1 \times \beta_\times$ and $\gamma_2 \times \beta_\times$. Then we finish by applying Case 1 and obtain $E(A^S((\gamma_1 + \gamma_2) \times \beta_\times)) = \gamma_1 \times \beta_\times + \gamma_2 \times \beta_\times \equiv (\gamma_1 + \gamma_2) \times \beta_\times$ by the distributivity axiom (20) of SKA.

**Case 4.3:** for $\alpha_1 = \alpha_\times^*$. For this case it is easy to look at the transition relation of the automaton $A^S(\alpha_\times^* \times \beta_\times)$ and the inductive reasoning applies $E$ on the smaller automata for $A^S(\alpha_\times \times \beta_\times)$ and $A^S(\alpha_\times)$ (as in Fig. 2.7).
**Theorem 2.2.28 (completeness of axiomatization)** For any two actions $\alpha$ and $\beta$ it is the case that $SKA \vdash \alpha = \beta$ iff the corresponding synchronous sets $\hat{I}_{SKA}(\alpha)$ and $\hat{I}_{SKA}(\beta)$ are the same.

**Proof:** The proof of the forward implication follows from the fact that $ASS$ is a synchronous Kleene algebra (see Theorem 2.2.20). We use induction on the derivation and have as base case that the implication holds for the axioms of $SKA$, which was proven in Theorem 2.2.20. For the inductive step we consider the rules of equational reasoning, which are the same for both $SKA$ and $ASS$ (the details are omitted).

The proof of the converse implication is based on Lemma 2.2.27. Take two arbitrary actions $\alpha, \beta \in SKA$ s.t. $\hat{I}_{SKA}(\alpha)$ and $\hat{I}_{SKA}(\beta)$ denote the same synchronous set (i.e., $\hat{I}_{SKA}(\alpha) = \hat{I}_{SKA}(\beta)$). Construct, cf. Theorem 2.2.26, $A^S(\alpha)$ and $A^S(\beta)$ corresponding to the actions and accepting respectively $\hat{I}_{SKA}(\alpha)$ and $\hat{I}_{SKA}(\beta)$. Then transform them into unique deterministic automata, cf. Corollary 2.2.25. Because $\hat{I}_{SKA}(\alpha) = \hat{I}_{SKA}(\beta)$ we have that $A^S(\alpha)$ and $A^S(\beta)$ denote the same automaton (up to isomorphism of states). Now we apply $E$ to obtain an action $\gamma$ which is both $\gamma \equiv \alpha$ and $\gamma \equiv \beta$ (cf. Lemma 2.2.27). Therefore we have the conclusion $\alpha \equiv \beta$ (i.e., $SKA \vdash \alpha = \beta$).

**Theorem 2.2.29 (decidability)** The problem of deciding whether $\alpha = \beta$ in $SKA$ is solved in quadratic time and is PSPACE-complete.

**Proof:** The proof is a consequence of the completeness theorem. In order to test the equality of two actions we test the equality of the corresponding synchronous sets $\hat{I}_{SKA}(\alpha)$ and $\hat{I}_{SKA}(\beta)$. This is done with the help of the translation of the actions into automata on synchronous sets. Then we use the method of [SM73] to get the PSPACE-completeness and a table-filling method to get a quadratic running time [HMU00].

### 2.2.5 Deontic actions as trees

A simple but important consequence of Theorem 2.2.26 that we need later in the semantics of the $CL$ logic in Chapter 3 is the interpretation of the deontic actions as rooted trees.\(^5\) Trees are structures simpler than automata and more intuitive to work with.

**Definition 2.2.30 (rooted tree)** A rooted tree with labeled edges is an acyclic connected graph $(\mathcal{N}, \mathcal{E}, \mathcal{A})$ with a designated node $r$ called root node. $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set

\(^5\)Recall that we call the $\ast$-free actions from Definition 2.2.8 deontic actions.
of labeled edges between nodes (in graphical notation \( n \xrightarrow{\alpha} m \)). The labels \( \alpha \in 2^{A_B} \) are sets of basic labels. Labels are compared for set equality (or set inclusion). Note the special empty set label. We consider a special label \( \Lambda \) to stand for an impossible label. We restrict our presentation to finite rooted trees (i.e., there is no infinite path in the graph starting from the root node). The set of all such defined trees is denoted \( T \).

Notation: When the label of an edge is not important (i.e., it can be any label) we may use the notation \( n \rightarrow m \) instead of \( n \xrightarrow{\alpha} m \, \forall \alpha \in 2^{A_B} \). Each node in \( \{ m \mid (n,m) \} \) is called a child node of \( n \). The siblings of a node \( m \) are all the nodes which have common parent with \( m \); i.e. \( \text{sibl}(m) = \{ m' \mid (n,m), (n,m') \in E \} \). Note that the root node has no siblings. We use the notation \( T_n \subseteq T \) to denote the subtree of \( T \) with root in the node \( n \) of \( T \). We denote by \(|n|\) the depth of the node \( n \) in the tree; which is the number of edges needed to reach \( n \) from the root. A path of a tree is denoted \( \sigma \in T \). A path which cannot be extended with a new edge is called final. The final nodes on each final path are called leaf nodes; denoted by \( \text{leafs}(T) = \{ n \mid n \text{ is a leaf node} \} \). The height of a tree, denoted \( h(T) \), is the maximum of \(|n|\) for all the leaf nodes \( n \). We write \( T_1 = T_2 \) when two trees are equal modulo renaming of the nodes (i.e. isomorphic).

**Theorem 2.2.31 (trees for deontic actions)** For any action \( \alpha \) there exists a tree representation corresponding to the canonical form \( \alpha \).

**Proof:** First remark that a rooted tree with labeled edges is just an automaton on synchronous strings with the following restrictions on the transition function:

a. no run goes through the same state twice (i.e., no loops),

b. each state different than the initial state has exactly one parent
   (i.e., \( \forall s_i \in A^S \) if \( s_i \neq s_n \) then \( \exists s_j \in A^S, \exists w \in \Sigma \) s.t. \( s_j \xrightarrow{w} s_i \in \rho \)),

c. between two states there is at most one transition
   (i.e., if \( s_j \xrightarrow{w} s_i \in \rho \) then \( s_j \xrightarrow{w} s_i \notin \rho \) for any \( w \neq w' \)).

The initial state is the root node and the final states are among the leaf nodes.

The construction of automata \( A^S \) from Theorem 2.2.26 can be viewed as a function \( A^S : A \rightarrow A^S \) that for each general action of \( SKA \) returns an automaton on synchronous strings as in Definition 2.2.22. Denote the restriction \( A^S|_{A^D} \) by \( A^D \) and call this the representation...
Figure 2.9: Tree interpretation for $p \cdot b + (d \times n) \cdot p \cdot p$.

Tests add expressive power to Kleene algebra. Kleene algebra with tests is known to be more expressive than propositional Hoare logic [Koz00] and it is the underlying algebraic formalism of the regular programs of PDL. On the other hand $\mathcal{KAT}$ is less expressive than PDL and different in time complexity too; $\mathcal{KAT}$ is PSPACE-complete whereas PDL is EXPTIME-complete.

---

6 These are the same as the automata from Fig. 2.1 on page 28 except for the tree corresponding to 0 where for technical reasons we used the special label $\Lambda$. For the time being we could do without this special label and use exactly the basic automata from Theorem 2.2.15(Fig. 2.8) and have leaf nodes that are not final states; when later the technicality using $\Lambda$ appears we can say that all these nonfinal leaf nodes are extended with one transition labeled with $\Lambda$. 

---

2.3 Synchronous Kleene Algebra with Boolean Tests

Synchronous Kleene Algebra with Boolean Tests
As algebras with the expressive power of PDL which are also extensions of Kleene algebras we mention dynamic algebras [Koz79, Pra90] and modal Kleene algebras [DBS04].

For this paper we are interested only in KAT. We follow the work of D. Kozen [Koz97b] and extend synchronous Kleene algebras with Boolean tests (denoted SKAT). This adds the expressive power of the Boolean tests to the synchronous actions of SKA.

SKAT inherits the expressivity of Kleene algebra with tests and it has the extra synchrony constructor. In the end of this section we show how we can reason about parallel programs with SKAT in the style of Owicki and Gries. Letting aside this particular application, SKAT is a general formalism which adds to the synchronous actions the power to make tests (of only Boolean expressivity). With SKAT one can express that at any point in a sequence of actions the system can stop and make a test (as a Boolean formula); if the test is successful then the execution can continue, otherwise it stops.

**Definition 2.3.1** Synchronous Kleene algebra with tests SKAT = (A, A^?, +, ·, ×, *, −, 0, 1) is an order-sorted algebraic structure with A^? ⊆ A which combines the previously defined SKA with a Boolean algebra. The structures (A^?, +, ·, ×, *, −, 0, 1) and (A^?, +, ×, −, 0, 1) are Boolean algebras and the Boolean negation operator − is defined only on Boolean elements of A^?.

**Notation:** The elements of the set A^? are called tests (or guards) and are included in the set of actions (i.e., tests are special actions). As in the case of actions, the Boolean algebra is generated by a finite set A^?_B of basic tests. We denote tests by ϕ, ψ, . . . and basic tests by p, q, . . . Note the overloading of the functional symbols +, ·, ×, and functional constants 0, 1: over arbitrary actions they have the meaning as in the previous section, whereas, over tests they take the meaning of the well known disjunction (for +), conjunction (for · and ×), falsity and truth (for 0 and 1). In this richer context the elements of A (i.e., the actions and tests) are the syntactic terms constructed with the grammar below:

\[\begin{align*}
\alpha & ::= a | \phi | \alpha + \alpha | \alpha \cdot \alpha | \alpha \times \alpha | \alpha^* \\
\phi & ::= p | 0 | 1 | \phi + \phi | \phi \cdot \phi | \phi \times \phi | \neg \phi
\end{align*}\]

We do not go into details about the properties of a Boolean algebra as these are standard results. For a thorough understanding see [Koz97b] and references therein.

Note that the preference relation ≤ is defined over tests also and 1 is the most preferable test; i.e., ∀ϕ ∈ A^?, ϕ ≤ 1. It is natural to think of 1 as T because testing a tautology always succeeds; i.e., 1 · α = α, which says that the action α can always be performed after a 1 test. The 0 is seen as the dual ⊥ meaning that testing a falsity never succeeds, and thus, any following action α is never performed; i.e., 0 · α = 0 (the action sequence stops when it reaches the 0 test).

**Definition 2.3.2 (extra axiom)** We give the equivalent of axiom (21) for tests:

\[(21') \quad (\phi \cdot \alpha) \times (\varphi \cdot \beta) = (\phi \times \varphi) \cdot (\alpha \times \beta) \quad \forall \phi, \varphi \in A^?\].

Note that 1 ∈ A^? and therefore this axiom allows sequences of actions with 1, which was not the case in axiom (21). On the other hand 1 is dealt with only in conjunction with another test, and not with another action. In this way the extra axiom (21') still avoids interleaving; synchronous actions cannot be reduced to interleavings. Particular instances of this axiom are \(\alpha \times \phi = \phi \cdot \alpha\) and \(\phi \times \alpha = \phi \cdot \alpha\).
2.3.1 Interpretation over sets of guarded synchronous strings

Guarded strings have been introduced in [Kap69] and have been used to give interpretation to Kleene algebra with tests [Koz97b]. Here we need an extension to guarded synchronous strings similar to the extension we gave in Section 2.2.3 from strings to synchronous strings. We intentionally overload several symbols as they have the same intuitive meaning but the particular definitions (adapted to guarded synchronous strings) are different.

Definition 2.3.3 (guarded synchronous strings) Over the set of basic tests $A_B^2$ we define atoms as functions $\nu : A_B^2 \rightarrow \{0, 1\}$ assigning a Boolean value to each basic test. Consider the same finite alphabet $\Sigma$ of all nonempty subsets of basic actions (denoted $x, y$ as before). A guarded synchronous string (denoted by $u, v, w$) is a sequence

$$w = \nu_0 x_1 \nu_1 \ldots x_n \nu_n, \quad n \geq 0,$$

where $\nu_i$ are atoms. We define first$(w) = \nu_0$ and last$(w) = \nu_n$.

Notation: Denote by $Atoms = \{0, 1\}^{A_B^2}$ the set of all atoms $\nu$. We say that an atom satisfies a test $\phi$ (denoted $\nu \models \phi$) iff the truth assignment of the atom $\nu$ to the basic tests makes $\phi$ true. Note that for basic tests $\nu \models p$ iff $\nu(p) = 1$. We define two mappings over guarded synchronous strings: $\tau$ which returns the associated (unguarded) synchronous string; i.e., $\tau(w) = x_1 \ldots x_n$ and $\pi$ which returns the sequence of guards; i.e., $\pi(w) = \nu_0 \nu_1 \ldots \nu_n$. Consider $Pref(\pi(w))$ to be the set of all prefixes of $\pi(w)$. Recall that when the $\times$-action $\alpha_x$ is known or important then we use the notation $\{\alpha_x\} \in P(A_B)$ instead of $x$.

Definition 2.3.4 Consider sets of guarded synchronous strings denoted $A, B, C$. On these we define the following operations:

$$0 \triangleq \emptyset$$

$$1 \triangleq Atoms$$

$$A + B \triangleq A \cup B$$

$$A \cdot B \triangleq \{uv \mid u \in A, v \in B\}$$

$$A \times B \triangleq \{uv \mid u \in A, v \in B\}$$

$$A^* \triangleq \bigcup_{n \geq 0} A^n$$

$$\neg A \triangleq Atoms \setminus A, \quad \forall A \subseteq Atoms$$

where $u = \nu_0^n x_1 \nu_1^n \ldots x_m \nu_m^n$ and $v = \nu_0^n y_1 \nu_1^n \ldots y_n \nu_n^n$ are guarded synchronous strings. The operation $\cup$ is just union over sets. The fusion product $uv$ of two guarded synchronous strings is defined iff $\text{last}(u) = \text{first}(v)$ and is $uv = \nu_0^m x_1 \ldots x_m \nu_0^m y_1 \nu_1^n \ldots y_n \nu_n^n$ with $\nu_0^m = \nu_0^n$. The synchrony operation on guarded synchronous strings $u \times v$ is defined iff $\pi(u) \in Pref(\pi(v))$ when $n \geq m$, and is:

$$u \times v \triangleq \nu_0^n (x_1 \cup y_1) \nu_1^n (x_2 \cup y_2) \nu_2^n \ldots (x_m \cup y_m) \nu_m^n y_{m+1} \nu_{m+1}^n \ldots y_n \nu_n^n;$$

whereas when $m \geq n$, we require $\pi(v) \in Pref(\pi(u))$, and in the definition above literally and everywhere, interchange $u$ and $v$, $m$ and $n$, and $x$ and $y$. 
Note that guarded synchronous strings can also be a single atom in which case \( \pi \) returns the atom itself. In such a case when both \( u \) and \( v \) are atoms then \( u \times v \) is defined only if \( u \) and \( v \) are the same atom. The same holds for \( \cdot \) over two atoms. Thus, when we consider two sets \( A \) and \( B \) of only atoms, then \( A \cdot B \) and \( A \times B \) become just intersection of sets.

**Definition 2.3.5 (interpretation [Koz03a])** The interpretation of the guarded actions is defined as the homomorphism \( \hat{I}_{SKAT} \) from \( T_{SKAT} \) into \( AGSS \). \( \hat{I}_{SKAT} \) is the homomorphic extension of the map \( I_{SKAT} : A_B \cup A_B^2 \cup \{0, 1\} \rightarrow AGSS \) which maps the generators of \( T_{SKAT} \) as follows:

\[
I_{SKAT}(a) = \{ \nu \{a\} \nu' | \nu, \nu' \in Atoms \}, \forall a \in A_B
\]

\[
I_{SKAT}(p) = \{ \nu \in Atoms | \nu(p) = 1 \}, \forall p \in A_B^2
\]

\[
I_{SKAT}(0) = \emptyset
\]

\[
I_{SKAT}(1) = Atoms
\]

The homomorphic extension is standard:

\[
\hat{I}_{SKAT}(\alpha) = I_{SKAT}(\alpha), \forall \alpha \in A_B \cup A_B^2 \cup \{0, 1\}
\]

\[
\hat{I}_{SKAT}(\alpha + \beta) = \hat{I}_{SKAT}(\alpha) + \hat{I}_{SKAT}(\beta)
\]

\[
\hat{I}_{SKAT}(\alpha \cdot \beta) = \hat{I}_{SKAT}(\alpha) \cdot \hat{I}_{SKAT}(\beta)
\]

\[
\hat{I}_{SKAT}(\alpha \times \beta) = \hat{I}_{SKAT}(\alpha) \times \hat{I}_{SKAT}(\beta)
\]

\[
\hat{I}_{SKAT}(\alpha^*) = \hat{I}_{SKAT}(\alpha)^*
\]

\[
\hat{I}_{SKAT}(\neg \phi) = \neg \hat{I}_{SKAT}(\phi)
\]

**Theorem 2.3.6** The smallest set containing \( \emptyset, Atoms, \) and the sets corresponding to \( A_B \) and \( A_B^2 \) (i.e., those sets returned by the \( I_{SKAT} \) of Definition 2.3.5), and closed under the operations \( +, \cdot, \times, ^*, \neg \) of Definition 2.3.4 is a synchronous Kleene algebra with tests. (Denote it \( AGSS \).)

**Proof:** By routine check of the axioms of Table 2.2 together with the extra axiom for tests (21′) and the axioms for the two Boolean algebras of Definition 2.3.1. For all the axioms a thorough proof would consider a double implication: \( \forall w : w \in A \iff w \in B \), where \( A = B \) is an axiom. Here we only discuss the proof and do not go into details.

Because \( 0 = \emptyset \) and \( + \) is defined with \( \cup \) the axioms (1)-(4) are as in Theorem 2.2.20. For axiom (5) we know from Definition 2.3.4 that any \( w \in A \cdot (B \cdot C) \) is \( w = w_{AWBw_C} \) with \( last(w_B) = first(w_C) \) and \( last(w_A) = first(w_Bw_C) \) which is the same as \( last(w_A) = first(w_B) \). The same conditions hold for the right part of the axiom. For (6) recall that \( 1 = Atoms \) and thus, each \( w \in A \) is also part of \( A \cdot 1 \) (and also \( 1 \cdot A \)) because \( last(w) \in Atoms \) (respectively \( first(w) \in Atoms \)). Moreover, all other combinations of \( w \in A \) with some \( \nu \in Atoms \) with \( last(w) \neq \nu \) are not included in \( A \cdot 1 \). Thus, the two sets \( A \cdot 1 \) and \( A \) have exactly the same guarded synchronous strings. For axiom (7) the cartesian product has no element because \( 0 = \emptyset \). For the distributivity axioms (8) and (9) a guarded synchronous string \( w \in A \cdot (B + C) \) is of the form \( w = w_{AWBC} \) where \( w_{BC} \) is either in \( B \) or in \( C \). If \( w_{BC} \in B \) then \( w \in A \cdot B \) and thus \( w \in (A \cdot B) + (A \cdot C) \). For axioms (10)-(13) of the Kleene *, see the related proof of [Koz97b] as the definition of * is essentially the same.
We need to check the axioms (14)-(21) for $\times$. We first check commutativity (15). We assume that $w \in A \times B$ and thus $w = w_A \times w_B$ and assume wlog that $|w_A| = n \leq m = |w_B|$ and thus $\pi(w_A) \in \text{Pref}(\pi(w_B))$. Then $w$ looks like:

$$w = \nu_0^B (x_1^B \cup y_1^B)\nu_1^B (x_2^B \cup y_2^B)\nu_2^B \ldots (x_n^B \cup y_n^B)\nu_n^B y_{n+1}^B \nu_{n+1}^B \ldots y_m^B \nu_m^B.$$

On the other hand, under the same assumption $n \leq m$ we can combine $w_B \times w_A$ to obtain:

$$w_B \times w_A = \nu_0^B (y_1^B \cup x_1^A)\nu_1^B (y_2^B \cup x_2^A)\nu_2^B \ldots (y_n^B \cup x_n^A)\nu_n^B y_{n+1}^B \nu_{n+1}^B \ldots y_m^B \nu_m^B.$$

Clearly, by the commutativity of $\cup$ for sets we have $w = w_B \times w_A \in B \times A$.

To check for associativity (14) take $w \in A \times (B \times C)$ to be $w = w_A \times w_{BC}$ where $w_{BC} = w_B \times w_C$. Because we proved commutativity we can now assume wlog that $|w_B| = m \leq |w_C| = n$ (we can reorder the terms using commutativity s.t. our assumption holds). Therefore, from the definition we have $\pi(w_B) \in \text{Pref}(\pi(w_C))$ and it remains to check three cases depending on the dimension of $w_A$:

a. assume $|w_A| = k \leq |w_B| = m$. Using the definition of $\times$ and the associativity of $\cup$ then $w_{BC}$ looks like:

$$w_{BC} = \nu_0^C (y_1^B \cup z_1^C)\nu_1^C \ldots \nu_k^C z_{k+1}^C \ldots z_n^C \nu_n^C,$$

and because $k \leq m$ then $\pi(w_A) \in \text{Pref}(\pi(w_{BC}))$ and $w$ becomes:

$$w = \nu_0^C (x_1^A \cup y_1^B \cup z_1^C)\nu_1^C \ldots \nu_k^C (y_{k+1}^B \cup z_{k+1}^C)\nu_{k+1}^C \ldots \nu_m^C z_{m+1}^C \ldots z_n^C \nu_n^C.$$

Under the same assumption $k \leq m \leq n$ we can combine first $w_A \times w_B = w_{AB} = \nu_0^B (x_1^A \cup y_1^B)\nu_1^B \ldots \nu_k^B z_{k+1}^B \ldots z_m^B \nu_m^B$ and because $\pi(w_{AB}) \in \text{Pref}(\pi(w_B)) \subseteq \text{Pref}(\pi(w_C))$ we can combine $w_{AB} \times w_C$ to obtain the same guarded synchronous string $w$.

b. assume $|w_B| = m \leq |w_A| = k \leq |w_C| = n$. We can combine $w_B \times w_C$ to obtain the $w_{BC}$ from case 1. Because $|w_A| \leq |w_C|$ then $\pi(w_A) \in \text{Pref}(\pi(w_C)) = \text{Pref}(\pi(w_{BC}))$. Therefore we may combine $w_A \times w_{BC}$ and obtain:

$$w = \nu_0^C (x_1^A \cup y_1^B \cup z_1^C)\nu_1^C \ldots \nu_m^C (x_{m+1}^A \cup z_{m+1}^C)\nu_{m+1}^C \ldots \nu_k^C z_{k+1}^C \ldots z_n^C \nu_n^C.$$

On the other hand, when combining first $w_A \times w_B$ we obtain:

$$w_{AB} = \nu_0^A (x_1^A \cup y_1^B)\nu_1^A \ldots \nu_k^A z_{k+1}^A \ldots x_k^A \nu_k^A.$$

Because $\pi(w_{AB}) = \pi(w_A) \in \text{Pref}(\pi(w_C))$ we may combine $w_{AB} \times w_C$ to obtain $w$ as before.

c. assume $|w_B| = m \leq |w_C| = n \leq |w_A| = k$. We combine $w_B \times w_C$ to obtain the $w_{BC}$ from case 1. Because $|w_C| \leq |w_A|$ then $\pi(w_{BC}) = \pi(w_C) \in \text{Pref}(\pi(w_A))$. Therefore we may combine $w_A \times w_{BC}$ and obtain:

$$w = \nu_0^A (x_1^A \cup y_1^B \cup z_1^C)\nu_1^A \ldots \nu_m^A (x_{m+1}^A \cup z_{m+1}^C)\nu_{m+1}^A \ldots \nu_k^A z_{k+1}^A \ldots x_k^A \nu_k^A.$$

As in the case 2. we may combine $w_A \times w_B$ to obtain $w_{AB}$ as before. Because $\pi(w_C) \in \text{Pref}(\pi(w_A)) = \text{Pref}(\pi(w_{AB}))$ we may combine $w_{AB} \times w_C$ to obtain the same $w$ as before.

Checking (16) and (17) is less laborious. For (17) use the same argument as for (7). For (16) is easy to see that for any atom $\nu \in \text{Atoms}$, $|\nu| \leq |w|$ for any guarded synchronous string $w$. Thus, a check $\pi(\nu) \in \text{Pref}(\pi(w))$ becomes just the check $\nu = \text{first}(w)$. In $1 \times A$, because $1 = \text{Atoms}$, for each $w \in A$ we always find a $\nu \in \text{Atoms}$ to match $\text{first}(w)$. Therefore, we always find the $w$ in $1 \times A$ (and in $A \times 1$ when we check for $\text{last}(w)$).

For the weak idempotence axiom (18) we work only with words of the form $\nu\{a\}\nu'$. It is easy to see that this axiom is respected. The proof for the distributivity axiom (19) follows an analogous argument as for (8). Axiom (20) is just a consequence of (19) and (15).

We now prove that the synchrony axiom (21) is respected. We consider $w \in (A_x \times A) \times (B_x \times B)$ where the sets $A_x$ and $B_x$ are obtained only from sets that interpret $\alpha \in A_B$ using only the $\times$ operation. For example, take two sets $\{\nu\{a\} \nu' \mid \nu, \nu' \in \text{Atoms}\}$ and $\{\nu\{b\} \nu' \mid \nu, \nu' \in \text{Atoms}\}$, then their synchronous combination is $\{\nu\{a, b\} \nu' \mid \nu, \nu' \in \text{Atoms}\}$. Note that only the action changes, whereas the atoms remain all the possible ones from $\text{Atoms}$. Therefore, $A_x = \{\nu\{\alpha_x\} \nu' \mid \nu, \nu' \in \text{Atoms}\}$, for some set of basic actions $\{\alpha_x\} \subseteq A_B$.

We have that $w = w_A, w_A \times w_{B_x} w_B$ where $w_{A_x} = \nu_A\{\alpha_x\} \nu'_A$ and $w_{B_x} = \nu_B\{\beta_x\} \nu'_B$ s.t. $\nu'_A = \text{first}(w_A)$ and $\nu'_B = \text{first}(w_B)$. Moreover, $\nu_A = \nu_B$ and $\nu'_A = \nu'_B$, and wlog we assume $\pi(w_A, w_B) \in \text{Pref}(\pi(w_{B_x}, w_B))$ which entails $\pi(w_A) \in \text{Pref}(\pi(w_B))$. Thus, the combination $w_A \times w_{B_x} \in A \times B$ and has $\nu'_A = \nu'_B$ as the first atom. We can also make the synchronous composition $w_{A_x} \times w_{B_x} \in A_x \times B_x$ which is $\nu_A\{\alpha_x, \beta_x\} \nu'_A$. Because $\nu'_A$ is the last atom of $w_{A_x} \times w_{B_x}$ and the first atom of $w_A \times w_B$ the concatenation $(w_{A_x} \times w_{B_x})(w_A \times w_B) \in (A_x \times B_x) \cdot (A \times B)$ and is the same as $w$.

The proof of the extra axiom (21') follows a similar argument as for (21).

Checking that $+, \cdot$, and $\neg$ over tests satisfy the laws of Boolean algebra is standard [Koz03a]. Moreover, from the results above it is clear that $\times$ over tests behaves like $\cdot$ (i.e., like Boolean conjunction).

\[ \square \]

### 2.3.2 Automata on guarded synchronous strings

Automata on guarded strings have been introduced in [Koz03a] as an extension of finite automata with transitions labeled with a test or with a basic action. These automata accept regular languages of guarded strings. We define here automata on guarded synchronous strings with the help of the definition of automata on synchronous strings from the previous section. The presentation that we give in this section is an alternative to the presentation of automata on guarded strings from [Koz03a]. The main motivation is that it makes simpler some definitions for our automata on guarded synchronous strings, and the proofs that lead to completeness become easier to present.

First we identify a class of automata which accept sets of atoms. We then define what we call two level automata, which accept guarded synchronous strings. Then we need to define the particular operations of fusion product and synchrony product (corresponding to respectively $\cdot$ and $\times$ over sets of guarded strings) in order to prove the equivalent of Kleene’s
Theorem. Using this, the proof of completeness requires a similar argument as in Theorem 2.2.28.

The next result is folklore and we omit its proof, which can be found in [Pri08b].

Proposition 2.3.7 ([HMU00]) For two finite sets $A$ and $B$ one can construct a class of finite automata which accept all and only (encodings of) functions $f : A \to B$. The language accepted by such an automaton is a set of functions. Denote the set of all such automata by $\mathcal{M}$ and one automaton by $M \in \mathcal{M}$.

Corollary 2.3.8 Automata of Proposition 2.3.7 defined on the particular sets $A$ and $B$ and $\{0, 1\}$ (for respectively $A$ and $B$) accept all and only the sets of atoms.

Corollary 2.3.9 The automata of Proposition 2.3.7 are closed under the well-known operations on finite automata union (denoted $\cup$) and intersection (denoted $\cap$); and correspond respectively to union of sets of functions and intersection of sets of functions. The automata of Proposition 2.3.7 are not closed under concatenation.

Definition 2.3.10 (automata on guarded synchronous strings) Let $A^G$ be a two level finite automaton $A^G = (S, \mathcal{P}(A_B), S_0, \rho, F, [\cdot])$, consisting at the first level of a finite automaton on synchronous strings, $(S, \mathcal{P}(A_B), S_0, \rho, F)$ as in Definition 2.2.22, together with a map $[\cdot] : S \to \mathcal{M}$. The mapping associates with each state of the first level an automaton $M \in \mathcal{M}$ as defined in Proposition 2.3.7 which accepts atoms. The automata in the states make the second (lower) level. Denote the language of atoms accepted by $\lceil s \rceil$ with $L(\lceil s \rceil)$.

Definition 2.3.11 (acceptance) Take the definitions and notations for automata on synchronous strings from Section 2.2.4. We say that a guarded synchronous string $w$ is accepted by a two level automaton $A^G$ iff there exists an accepting run of the first level automaton which accepts $\tau(w)$ and for each state $s_i$ of the run there exists an accepting run of the automaton $\lceil s_i \rceil$ which accepts the corresponding atom $\nu_i$ of $w$.

It is easy to see that automata on guarded synchronous strings can be considered as ordinary finite automata. The two level definition that we give is useful in defining the fusion product and synchrony product operations over these automata.

Definition 2.3.12 (fusion product) For two automata over guarded synchronous strings $A^G_1 = (S_1, \mathcal{P}(A_B), S_0^1, \rho_1, F_1, [\cdot]_1)$ and $A^G_2 = (S_2, \mathcal{P}(A_B), S_0^2, \rho_2, F_2, [\cdot]_2)$ we define the fusion product automaton as

$$A^G_{12} = (S, \mathcal{P}(A_B), S_0, \rho, F, [\cdot])$$

where:

- $S = (S_1 \setminus F_1) \cup (S_2 \setminus S_0^2) \cup S'$;
- $S' = F_1 \times S_0^2$ and for $s \in S'$ denote $s|_{F_1} \in F_1$ the first component, and $s|_{S_0^2}$ the second component;
- $S_0 = S_0^1$.
• $F = F_2$;

• $\rho = (\rho_1 \setminus \{(s_1, a, s_2) \in \rho_1 \mid s_2 \in F_1\}) \cup (\rho_2 \setminus \{(s_1, a, s_2) \in \rho_2 \mid s_1 \in S_0^2\})$
  \[
  \cup \{\langle s_1, a, s \rangle \mid s \in S' \text{ and } \langle s_1, a, s \rangle_{F_1} \in \rho_1\}
  \cup \{\langle s, a, s_1 \rangle \mid s \in S' \text{ and } \langle s_{S_0^2}, a, s_1 \rangle \in \rho_2\};
  \]

• $\forall s \in S', \lceil s \rceil = \lceil s_{F_1} \rceil \cap \lceil s_{S_0^2} \rceil$.

The first two conditions ensure that we combine the final states of the first automaton with all the initial states of the second automaton, so to get all possible concatenations. The next two conditions set the initial states to be the initial states of $A^G_1$ and the final states to be the final states of $A^G_2$. The condition on the transition relation keeps all the transitions from both automata and modifies accordingly those transitions that have to do with the old final states of $A^G_1$ and the old initial states of $A^G_2$. The last condition makes sure that we concatenate only guarded synchronous strings that have the same atoms last and first (i.e., we keep in the new nodes of $S$ only those atoms that correspond to both the old final nodes of $A^G_1$ and to the old initial nodes of $A^G_2$). Note the use of $\cap$ to denote the operation of intersection of automata (which also results in intersection of their accepted languages).

**Definition 2.3.13 (synchrony product)** For two disjoint automata over guarded synchronous strings $A^G_1 = (S_1, \mathcal{P}(A_B), S_1^0, \rho_1, F_1, \lceil \cdot \rceil_1)$ and $A^G_2 = (S_2, \mathcal{P}(A_B), S_2^0, \rho_2, F_2, \lceil \cdot \rceil_2)$ we define the synchrony product as follows. Apply to the top level of these automata the synchrony product as defined in Theorem 2.2.26. For the lower level automata of the nodes do their intersection:

\[ \forall (s_1, s_2) \in S_1 \times S_2, \lceil \langle s_1, s_2 \rangle \rceil = \lceil s_1 \rceil \cap \lceil s_2 \rceil. \]

**Proposition 2.3.14** Automata on guarded synchronous strings are closed under union, fusion product, and synchrony product.

**Proof:** The union operation for automata on guarded synchronous strings is the same as given in Theorem 2.2.26 for automata on synchronous strings where any new first level nodes that are added have associated automata accepting any atom (i.e., accepting the set $\text{Atoms}$). For the fusion product it is clear that the top level automaton remains an automaton on synchronous strings and in each node the intersection of the automata for guards also gives an automaton for guards (because of closure under intersection; see Corollary 2.3.9). The same observation applies to the synchrony product.

**Theorem 2.3.15 (translation into automata)** For any $\alpha \in T_{SKAT}$ there is a corresponding automaton $A^G(\alpha)$ which accepts exactly the set of guarded synchronous strings $\hat{I}_{SKAT}(\alpha)$ (i.e., the interpretation of $\alpha$).

**Proof:** We follow a similar recursive construction of the automaton based on the structure of the actions as we did in Theorem 2.2.26. We give constructions for the basic actions (here the recursion stops) and for each action constructor of $SKAT$. In each case we have to show that the automaton accepts the same set of guarded synchronous strings as the interpretation of the action. To show this we use an inductive argument.
From Proposition 2.3.7 we know we can construct an automaton for guards to recognize a
given set of atoms. More precisely we can construct an automaton which recognizes the set of
atoms that make only the basic test \( p \) true, or make only some or none of the basic tests true.

**Base case:** For a basic test \( p \) the automaton consists of only one top level state \( s \) which
is both the initial and the final state. The second level automaton \( [s] \) is such constructed to accept
all and only the atoms which make the basic test \( p \) true. It is clear that this automaton accepts the
set of guarded synchronous strings \( \{ \nu \in Atoms \mid \nu(p) = 1 \} \) which corresponds to \( \hat{I}_{SKAT}(p) \).

For the special actions \( 0 \) and \( 1 \) the construction is similar to that for tests. For \( 0 \) the
automaton \( [s] \) accepts the empty set thus the whole automaton accepts \( \hat{I}_{SKAT}(0) = \emptyset \). For \( 1 \) the
automaton \( [s] \) accepts all possible strings (i.e., a universal automaton) encoding all possible;
thus the automaton \( A^G(1) \) accepts \( Atoms = \hat{I}_{SKAT}(1) \).

For a basic action \( a \in A_B \) we construct an automaton which at the top level is as the
automaton in Fig. 2.2(i) on page 29 and at the second level the automata \([s_1]\) and \([s_2]\) both
accept \( Atoms \); thus \( A^G(a) \) accepts \( \{ \nu(a) \nu' \mid \nu, \nu' \in Atoms \} = \hat{I}_{SKAT}(a) \).

**Inductive step:** Corresponding to the action constructors \( \cdot \) and \( \times \) we have respectively the
constructions of fusion product and synchrony product on automata given in Definition 2.3.12
and Definition 2.3.13.

Consider \( \alpha = \alpha_1 \cdot \alpha_2 \). By the inductive hypothesis we have \( \mathcal{L}(A^G(\alpha_1)) = \hat{I}_{SKAT}(\alpha_1) \)
and \( \mathcal{L}(A^G(\alpha_2)) = \hat{I}_{SKAT}(\alpha_2) \). From Definition 2.3.5 we know that
\( \hat{I}_{SKAT}(\alpha) = \hat{I}_{SKAT}(\alpha_1) \cdot \hat{I}_{SKAT}(\alpha_2) \) where \( \cdot \) is as in Definition 2.3.4. The construction for fusion product of Definition 2.3.12 generates \( A^G(\alpha) \) s.t. it accepts \( w = w_1w_2 \) where \( w_1 \in A^G(\alpha_1) \) and \( w_2 \in A^G(\alpha_2) \).

By the inductive hypothesis we have that \( w_1 \in \hat{I}_{SKAT}(\alpha_1) \) and \( w_2 \in \hat{I}_{SKAT}(\alpha_2) \). Moreover,
last \( (w_1) = \text{first} (w_2) \) because of the last constraint of Definition 2.3.12 (in the generation of the
automaton \( A^G(\alpha) \)). Therefore, \( w \) is contained in \( \hat{I}_{SKAT}(\alpha_1) \cdot \hat{I}_{SKAT}(\alpha_2) \) cf. Definition 2.3.4.

It remains to prove the opposite inclusion; i.e., that for any two \( w_1 \in \hat{I}_{SKAT}(\alpha_1) \) and \( w_2 \in \hat{I}_{SKAT}(\alpha_2) \) we have that if \( w_1w_2 \in \hat{I}_{SKAT}(\alpha_1 \cdot \alpha_2) \) then \( w_1w_2 \in \mathcal{L}(A^G(\alpha_1 \cdot \alpha_2)) \). From the
same inductive hypothesis we know that \( w_1 \in A^G(\alpha_1) \) and \( w_2 \in A^G(\alpha_2) \). Because \( w_1w_2 \in \hat{I}_{SKAT}(\alpha_1 \cdot \alpha_2) \) then we know (cf. Definition 2.3.4) that \( \text{last} (w_1) = \text{first} (w_2) \). According
to Definition 2.3.12 the last condition is satisfied for \( w_1 \) and \( w_2 \) and thus the string \( w_1w_2 \) is accepted
by the fusion product of \( A^G(\alpha_1) \) and \( A^G(\alpha_2) \).

Consider \( \alpha = \alpha_1 \times \alpha_2 \). We treat first the inclusion \( \mathcal{L}(A^G(\alpha)) \subseteq \hat{I}_{SKAT}(\alpha) \); the opposite
inclusion is simple and follows a similar reasoning as in the case before and Theorem 2.2.26.

By the inductive hypothesis we have \( \mathcal{L}(A^G(\alpha_1)) = \hat{I}_{SKAT}(\alpha_1) \) and \( \mathcal{L}(A^G(\alpha_2)) = \hat{I}_{SKAT}(\alpha_2) \).
From Definition 2.3.5 \( \hat{I}_{SKAT}(\alpha_1 \times \alpha_2) = \hat{I}_{SKAT}(\alpha_1) \times \hat{I}_{SKAT}(\alpha_2) \) where \( \times \) is the operation of
Definition 2.3.4 over sets of guarded synchronous strings. \( A^G(\alpha_1 \times \alpha_2) \) is constructed as the
synchrony product (i.e., Definition 2.3.13) of the two smaller automata \( A^G(\alpha_1) \) and \( A^G(\alpha_2) \).

Consider a guarded synchronous string \( w \) accepted by \( A^G(\alpha_1 \times \alpha_2) \). From Definition 2.3.11
we know that \( w \) is accepted if there exists an accepting run of the top level automaton of
\( A^G(\alpha_1 \times \alpha_2) \) on the synchronous string \( \tau(w) \), and for each state \( s_i \) of this accepting run the
lower level automata accept the corresponding \( i \)th element of \( \tau(w) \). Because Definition 2.3.13,
of synchrony product, uses the same construction for the top level automata as in the unguarded
case then Theorem 2.2.26 assures that the synchronous string accepted by \( A^G(\alpha_1 \times \alpha_2) \) comes
from the synchronous composition of two strings \( u \) and \( v \) accepted by the smaller automata.
Synchronous Kleene Algebra

A^G(α_1) respectively A^G(α_2). Moreover, the synchrony product construction makes the intersection of the automata in the states of the two A^G(α_1) and A^G(α_2) therefore, we know that π(u) ∈ Pref(π(v)) (or interchange u with v depending on the lengths). Because of this the requirements of Definition 2.3.4 for × over guarded synchronous strings are satisfied for u and v and thus w = u × v ∈ I_{SKA}(α_1) × I_{SKA}(α_2).

For the action constructors + and · we have the standard constructions from Fig. 2.2(i) respectively Fig. 2.2(iii) defined in Theorem 2.2.26 on page 28 which are independent of our special definition of the two level automata on guarded synchronous strings. Their proofs are standard as for finite automata and we skip them.

As an example let us consider the automaton A^G(δ) for the synchronous action δ = (p·b)^* + (d×n)·p·φ?·p, depicted in Fig. 2.10.

2.3.3 Completeness and decidability

Theorem 2.3.16 (Completeness) For any two actions α and β of T_{SKAT} we have that SKAT ⊢ α = β iff the corresponding sets of guarded synchronous strings I_{SKAT}(α) and I_{SKAT}(β) are the same.

Proof: The forward implication (or soundness) comes as a consequence of Theorem 2.3.6 (because the sets of guaranteed synchronous strings form a synchronous Kleene algebra with tests). Consider, for example, one case for axiom (14) when α = α_1 × (α_2 × α_3) and β = (α_1×α_2)×α_3. From Definition 2.3.5 I_{SKAT}(α_1×(α_2×α_3)) = I_{SKAT}(α_1)×(I_{SKAT}(α_2)×I_{SKAT}(α_3)) and I_{SKAT}((α_1×α_2)×α_3) = (I_{SKAT}(α_1)×I_{SKAT}(α_2))×I_{SKAT}(α_3). The equality of the two sets comes from the associativity of × over sets of guarded synchronous strings which was proven in Theorem 2.3.6.

For the backward implication take arbitrary α, β ∈ T_{SKAT} with I_{SKAT}(α) = I_{SKAT}(β). For the two actions construct the corresponding A^G(α) and A^G(β) as in Theorem 2.3.15 to accept I_{SKAT}(α) respectively I_{SKAT}(β). In Theorem 2.2.28 it was easy to apply the Myhill-Nerode minimization procedure on the equivalent deterministic automata for synchronous strings. Unfortunately, in the case of guarded synchronous strings our two levels definition makes it impossible to adapt the subset construction method for determination of automata on guarded
synchronous strings of Definition 2.3.10. When making the set construction it is not possible to decide for a new state (as a set of some old states) what is the associated automaton (i.e., which set of atoms should it accept?).

On the other hand, as we remarked before, the two level definition is just a trick to get the right definitions of operations for automata on guarded synchronous strings and to make easy the proof of Kleene’s theorem for constructing an equivalent automaton for a $\hat{I}_{SKAT}(\alpha)$. After having obtained such an automaton we do not need this definition anymore and we can see the automaton as a special finite automaton on a special alphabet $\Sigma = \mathcal{P}(A_B) \cup \text{Atoms}$. More precisely, for each state with its deterministic finite automaton accepting a set of atoms we can split it into two states with one transition between them for each atom accepted by the automaton in the old state. For such a finite automaton the standard subset construction works and the Myhill-Nerode procedure is then applicable to obtain a unique automaton accepting $\hat{I}_{SKAT}(\alpha)$ (respectively $\hat{I}_{SKAT}(\beta)$). Because of the assumption of the theorem these two automata, $A^{\hat{G}}(\alpha)$ and $A^{\hat{G}}(\beta)$, denote the same automaton up to isomorphism of states.

All that remains to do is to show that an equivalent of Lemma 2.2.27 holds in this case; i.e., that a similar method of eliminating states $E$ works for automata on guarded synchronous strings too. This is not hard to see as the automata that we work now with are the same as automata on synchronous strings except that between each two transitions labeled with a $\times$-action $\{\alpha\times\}$ there are all those transitions labeled with atoms. But the $E$ procedure is not influenced by these (it just concatenates them, as tests, to the synchronous actions, thus obtaining the guarded synchronous actions) and thus we get $\alpha \equiv E(A^{\hat{G}}(\alpha)) = E(A^{\hat{G}}(\beta)) \equiv \beta$. \hfill $\square$

2.3.4 $SKAT$, Hoare logic, and shared-variables concurrency

Propositional Hoare Logic (PHL) is the version of Hoare logic which does not involve the assignment axiom explicitly [Koz00]. PHL reasons about programs at a more abstract level where the assignment axiom instances are just particular cases of atomic actions. Kleene algebra with tests ($KAT$) subsumes PHL and has the same complexity (PSPACE-complete for the Horn theory with premises of the form $\alpha = 0$). Moreover, $KAT$ is complete for relational valid Horn formulas (i.e., all the rules of PHL are theorems of $KAT$), whereas PHL is incomplete. For instance, the following valid inference cannot be derived in PHL:

$$\begin{align*}
\{\psi\} & \text{if } \phi \text{ then } \alpha \text{ else } \alpha \{\psi\} \\
\{\psi\} & \alpha \{\psi\}
\end{align*}$$

Extensions of Hoare logic exist which treat procedure calls, goto jumps, pointers, or aliasing. Similar extensions can be devised for Kleene algebra with tests, e.g., some form of higher-order functions [AHK06b], non-local flow of control [Koz08], or local variables [AHK08]. We do not know about nested procedure calls and returns, which are known to be a context free properties (and not regular properties as is the style of $KAT$).

Regarding expressivity, $KAT$ can encode while programs (and more) which correspond to the notion of tail recursion (or iteration) from programming languages. It is known that tail recursion is strictly less expressive than (full) recursion. Recursion can be encoded with while programs and a stack. This corresponds to the context-free languages as opposed to the regular
languages where KAT resides. The stack can be expressed in First-Order Dynamic Logic (which is undecidable in general), but KAT relates only to the weaker Propositional Dynamic Logic.

SKAT includes KAT and thus all these expressiveness issues hold for our SKAT too. Encoding partial correctness assertions (PCAs) into SKAT is done as in [Koz00, HKT00]. The PCA \( \{\phi\} \alpha \{\psi\} \) intuitively says that if the program \( \alpha \) starts in a state where \( \phi \) holds (i.e., the precondition is true) then, whenever the program terminates,\(^7\) the postcondition \( \psi \) will hold. There are two equivalent ways of encoding PCAs in SKAT: \( \phi \alpha \neg \psi = 0 \) (it is not possible that program \( \alpha \) starts with precondition \( \phi \) and terminates with postcondition \( \neg \psi \)) or \( \phi \alpha = \phi \alpha \psi \) (testing the postcondition \( \psi \) after termination of \( \alpha \), started with precondition \( \phi \), is superfluous).

The definition of interference freedom of Owicke and Gries [OG76] says that an action \( a \) does not interfere with another action \( \alpha \) iff it does not change the postcondition of \( \alpha \) and when interleaved at any point in \( \alpha \) it does not change the precondition of the remaining actions (to be executed) of \( \alpha \). In [OG76] the actions that need checking for interference freedom are only the statements that can change the state of the system (i.e., \textit{await} and assignment). In our case these correspond to the basic actions.

Interference freedom in the synchrony model of SKAT is given by the \( \sim_c \) relation (defined in terms of \( \#_c \)) on the basic actions of \( A_B \). The difference is that in our case \( \#_c \) is given by an "oracle" whereas in [OG76] it is given by rules based on the syntax (i.e., the assignment axiom schemata, which in practice reduces to the instances of the axiom for each particular assignment). In the case of SKAT we assume an oracle because the basic actions have no assumed structure (basic actions are abstract entities). If particularized to assignments then the oracle giving the \( \#_c \) relation becomes the assignment axiom schemata.

Extending the conflict and compatibility relations to the whole actions of SKAT is left for future work, but initial ideas are presented in Section 2.5.2. If two basic actions are not interference free then they cannot be executed synchronously: \( a \#_c b \rightarrow a \times b = 0 \) (cf. axiom (22) of Section 2.2.1). This is extended to arbitrary actions \( a \#_c \gamma \rightarrow a \times \beta = 0 \) (if \( a \) and \( \gamma \) are not interference free then their synchronous execution yields the impossible action). The following example needs this assumption, that \( \gamma \) interferes with neither \( \alpha \) nor \( \beta \).

Example 2.3.1 Consider two programs: a conditional if \( \phi \) then \( \alpha \) else \( \beta \) and an arbitrary \( \gamma \). The conditional is written in SKAT as \( \phi \alpha + \neg \phi \beta \). Now put these two programs to run in parallel (synchronously) and therefore write \( (\phi \alpha + \neg \phi \beta) \times \gamma \). The following equality follows from the axioms of SKAT: \( (\phi \alpha) \times \gamma + (\neg \phi \beta) \times \gamma = (\phi \alpha \times \gamma) + (\neg \phi \beta \times \gamma) \) which by axiom (21') and rules of Boolean algebra becomes \( \phi (\alpha \times \gamma) + \neg \phi (\beta \times \gamma) \). If we write this back into the while language we get: if \( \phi \) then \( \alpha \times \gamma \) else \( \beta \times \gamma \).

2.4 Synchronous Kleene Algebra compared to Other Concurrency Models

We present two models of concurrency based on partial orders and compare them with SKA for expressivity issues. SKA does not belong to the class of models of concurrency that are

\(^7\)To talk about termination (total correctness) we need to use an extended version of the Hoare logic (not considered in this paper).
based on interleaving, but more to the class of models based on partial orders. It turns out that
pomsets (and thus event structures [NPW79, Win88]) are strictly more expressive than
based on interleaving, but more to the class of models based on partial orders. It turns out that
Synchronous Kleene Algebra compared to Other Concurrency Models

2.4.1 Mazurkiewicz trace theory

Definition 2.4.1 (Mazurkiewicz traces) Consider a symmetric and irreflexive binary relation
$I_{A_B}$ called the independence relation (i.e., not causal) on a set of basic actions, say $A_B$. Define
$\equiv_{A_B}$ as the least congruence in the monoid of strings over $A_B$, i.e., $(A_B^*, \cdot, 1)$ s.t. if $(a, b) \in I_{A_B}$
then $ab \equiv_{A_B} ba$. For arbitrary strings we say that $u \equiv_{A_B} v$ iff $\exists w_1 \ldots w_n$ with $u = w_1$ and
$v = w_n$ and $\forall i, \exists w', w'' \exists a, b$ s.t. $(a, b) \in I_{A_B}$ and $w_i = w'abw''$ and $w_{i+1} = w''abw''$. One
equivalence class generated by $\equiv_{A_B}$ is called a (Mazurkiewicz) trace and is denoted by $[w]_{\equiv_{A_B}}$
(the representative $w$ is said to generate $[w]_{\equiv_{A_B}}$).

If $u \equiv_{A_B} v$ then $u$ is a permutation of $v$. A trace represents a run of a concurrent sys-
tem. On the other hand a trace encodes several possible sequential runs which are considered
equivalent due to the independence of some of the basic actions involved. From this point of
view Mazurkiewicz traces talk about a special form of interleaving. The independence relation
makes two basic actions globally independent; i.e., the basic actions are independent of each
other no matter their position on the sequential runs.

In $SKA$ a $\times$-action $\alpha_\times \in A_B^\times$ is interpreted as the set of basic actions that compose it, e.g.,
$\{a, b, c\}$. Taking the same view (with sets of equivalent interleavings) we can say that $\alpha_\times$ en-
codes all the possible interleavings of these basic actions, e.g., $\{abc, acb, bac, bca, cab, cba\}$.
Therefore, in the context of $SKA$ the following definitions apply to the independence relation
of Mazurkiewicz traces.

Definition 2.4.2 Define the relation $I_{A_B}$ as: for all $a, b \in A_B$, if $a \sim_C b$ (i.e., $a\times b \neq 0$ cf. axiom
(22)) then $(a, b) \in I_{A_B}$. Extend this to $\times$-actions $\alpha_\times$ to say that if $\alpha_\times \sim_C \beta_\times$ (i.e., $\alpha_\times \times \beta_\times \neq 0$)
then $\forall a \in \{\alpha_\times\}, b \in \{\beta_\times\}: (a, b) \in I_{A_B}$.

Proposition 2.4.3 For a $\times$-action $\alpha_\times = a_1 \times \ldots \times a_n$, $I_{A_B}$ restricted to the basic actions of $\alpha_\times$ is
a total relation; i.e., $\forall 0 < i, j \leq n: (a_i, a_j) \in I_{A_B}$.

Proof: The proof is easy and uses a reductio ad absurdum argument. Suppose that for some
$0 < i \leq n$ and $0 < j \leq n$ it holds that $(a_i, a_j) \notin I_{A_B}$. This means that $a_i \not\equiv_C a_j$ (i.e., $a_i \not\equiv_C a_j$, for otherwise, by Definition 2.4.2, would imply $(a_i, a_j) \in I_{A_B}$ contradicting our assumption). By axiom (22) it means that $a_i \times a_j = 0$ which implies that $\alpha_\times = 0$ which contradicts the
statement of the proposition.

This proposition shows a first difference between the concurrency modelled in $SKA$ and the
concurrency of Mazurkiewicz traces. In the latter the independence relation is not necessarily
total (i.e., it may be defined as partial); this fact allows for some basic action to move back and
forth along the sequence of actions depending on which actions it is independent of. Therefore
$SKA$ cannot capture the concurrent behavior of Mazurkiewicz traces.
For general actions of SKA generated using also the \( \cdot \) operator the definitions above are not sufficient any more. Consider this simple example: \((a \times b) \cdot a\) in SKA has the following intended sequential runs: \(\{aba, baa\}\); whereas in Mazurkiewicz traces, because \((a, b) \in I_{AB}\) we get the following sequential runs: \([aba]_{\equiv AB} = \{aba, baa, aab\}\). This shows that Mazurkiewicz traces cannot capture the concurrent behavior intended in SKA because we need the independence relation to be local to each sequential step of a concurrent run.

In conclusion, for Mazurkiewicz traces the independence relation is global and partial. If we take the similar view in SKA we need a local and total independence relation. The locality comes from the perfect synchrony model we adopted, where all the concurrent actions are executed at each tick of a universal clock. The totality comes from our view of \(\times\)-actions as forming a set.

**Proposition 2.4.4 (incomparability of SKA and Mazurkiewicz traces)** Mazurkiewicz traces theory and synchronous Kleene algebra are incomparable:

- **a.** SKA cannot capture the concurrent behavior of the Mazurkiewicz traces because the independence relation defined by SKA is total, instead of partial;
- **b.** Mazurkiewicz traces cannot capture the concurrent behavior of SKA because the independence relation defined by SKA is inherently local.

### 2.4.2 Pomsets

Pomsets have long been advocated by Pratt [Pra86] and many of the initial theoretical results were published as [Gis84]. The theory of pomsets is among the first in concurrency theory to make a distinction between events \((E)\) and actions \((A)\). A pomset is a partially ordered set of events labeled (non-injectively) by actions. Pomsets extend the idea of strings, which are linearly ordered multisets, to partially ordered multisets. Normally a multiset is \(\mathbb{N}^A\) and assigns to each action of \(A\) a multiplicity from \(\mathbb{N}\). In pomset theory they are more: \(E^A\) which assigns to each action of \(A\) a set of events from \(E\), and more, events are ordered by the temporal partial order. Thus, an action may be executed several times and each execution of an action is an event. Formally a pomset is the isomorphism class (w.r.t. the events) of the structure \((E, A, <, \mu)\) where \(\mu : E \rightarrow A\) is the labeling function of the events by action names.

Two events which are incomparable by \(<\) are permitted to occur concurrently. An important feature of the pomset theory is that it is independent of the granularity of the atomicity; i.e., events may be either atomic or may have an even more elaborated structure (in [Gis84] operations over pomsets are defined by replacing events (with the same action name) by new pomsets). Moreover, the view of time does not matter as events may occupy time points or time intervals with no difference to the theory. There is also a large number of operations defined over pomsets (see [Pra86]), more than in the other theories we have seen.

A pomset describes only one execution of the concurrent system. A set of pomsets is called a process and describes the whole set of concurrent behaviors of a system (or process). Pomsets are more expressive then our synchronous actions. We know that a synchronous action represents a set of synchronous strings (as we called them). Each synchronous string is a particular pomset, formally defined as a synchronous pomset:
Definition 2.4.5 (synchronous pomsets) The class of pomsets called synchronous pomsets is precisely defined as those pomsets where the partial order respects the constraint:

\begin{equation}
\text{all maximal independent sets are disjoint, uniquely labeled, and completely ordered,}
\end{equation}

where an independent set of events is \(X \subseteq E\) s.t. \((e_i \not< e_j) \land (e_j \not< e_i)\) for all \(e_i, e_j \in X\). An independent set is uniquely labeled iff the labeling function is injective on \(X\); i.e., \(\mu|_X\) is injective. Two independent sets \(X_i, X_j\) are completely ordered iff whenever there exist \(e_i \in X_i\) and \(e_j \in X_j\) s.t. \(e_i < e_j\) then \(e_i < e_j\) for all \(e_i \in X_i\) and \(e_j \in X_j\).

Theorem 2.4.6 (SKA embedded in pomset theory) Synchronous strings are completely characterized by synchronous pomsets.

Proof: We need to prove two implications: for any \(w\) a synchronous string as in Definition 2.2.19 there is a synchronous pomset simulating it; and for any synchronous pomset there is a synchronous string.

Consider a synchronous string as pictured in Fig. 2.11. It is formed of sets of incomparable events named by unique actions (because of the axiom (18)). These sets are pairwise disjoint and all the events in one set that follows after a \(\cdot\) operator are in the relation \(<\) with all the events that precede them (because of the associativity and non-commutativity of \(\cdot\) we get the transitivity of the partial order). Therefore these are independent sets as in the definition above and are also maximal. The requirement of being completely ordered is clearly satisfied. Thus, we have the synchronous pomset.

For a synchronous pomset the fact that we consider the maximal independent sets to be disjoint gives much of the proof. These maximal independent sets make the \(\times\)-actions of the synchronous string. (In each \(\times\)-action all the basic actions composing it are considered independent.) The requirement of completely ordered ensures that one \(\times\)-action (i.e., all its composing basic actions) precedes the next \(\times\)-action (i.e., all the elements of the next independent set). Finally, the injective labeling ensures that \(\times\)-actions are actually interpreted as sets (and not as multisets), respecting axiom (18).

The good thing about the synchronous actions of SKA is that all the actions can be obtained (constructed) from a finite set of basic actions using a finite set of operations on actions. Is it the same situation for the synchronous pomsets?
Because of the $+$ operation the actions of $SKA$ define sets of behaviors (of synchronous strings). Therefore we need to talk not of pomsets but of sets of pomsets (i.e., processes). The same $+$ operation exists for sets of pomsets (i.e., their union) which on synchronous pomsets behaves exactly like the $+$ in $SKA$. For the $\cdot$ there is the $;$ operation on pomsets. The extension of $;$ to sets of pomsets is exactly the same as the extension of $\cdot$ to sets of synchronous strings. Similarly we find the Kleene $^*$ for pomsets.

For the $\times$ of $SKA$ we did not find a straightforward equivalent for synchronous pomsets. Moreover, we are not sure if there is a pomset definable operation (as in terminology of [Gis84]). The first candidate was the concurrence operation $||$ but this breaks the completely ordered requirement. The orthocurrence operation on pomsets is also not good. We could not find a satisfactory new definition for $\times$ over pomsets because in order to enforce the conditions of synchronous pomsets we needed to look through the whole (infinite) structure of the partial order on events starting with the smallest elements in the order.

### 2.4.3 Concurrent Kleene algebra

Recently Concurrent Kleene algebra (CKA) was proposed in [HMSW09] as a general formalism for reasoning about concurrent programs. CKA has, at first sight, striking resemblances with $SKA$. We discuss CKA in relation with $SKA$, focusing on the underlying ideas and intuitions of the two models.

CKA is defined as two quantales $(S, +, ;, 0, 1)$ and $(S, +, *, 0, 1)$ related by an exchange axiom ($; \text{ and } *$ correspond to respectively $\cdot$ and $\times$ in $SKA$). Quantales are idempotent semirings which are also complete lattices under the natural order $\leq$ of the semiring (i.e., have the extra constraint of a top element). What differentiates $SKA$ from CKA is the synchrony axiom of the first and the exchange axiom of the second, and as we see later, also the choice of models.

Both algebras can model Hoare-style reasoning about sequential programs. Moreover, both algebras can reason about some form of concurrent programs: CKA can model Jone’s rely/guarantee calculus [Jon81], whereas $SKA$ can reason about synchronous programs in the style of Qwicki and Gries (cf. Section 2.3.4).

The exchange axiom entails two properties of CKA relevant for our discussion:

$$
(\alpha; \beta); (\alpha'; \beta') \leq (\alpha; \alpha') * (\beta; \beta') 
$$

(2.2)

$$
\alpha; \beta \leq \alpha * \beta 
$$

(2.3)

Equation (2.2) is similar to the synchrony axiom. It is more general because it considers $\alpha$ and $\beta$ general actions and not only $\times$-actions. On the other hand, it is less informative than the synchrony axiom because it only states inclusion of behaviors and not equality. One may read (2.2) as: “All behaviors coming from putting two concurrent compositions in sequence are captured by putting the respective sequences in concurrent composition.”

Equation (2.3) states that the concurrent composition captures all the behavior of the sequential composition. This is the same as in the “concurrency as interleaving” approach where all the behaviors coming from all the possible interleavings are contained in the concurrent composition. CKA captures this because of (2.3) and the commutativity of $*$ (i.e., $\alpha; \beta + \beta; \alpha \leq \alpha * \beta$). Equation (2.3) does not hold in $SKA$ and has no similar counterpart either. In $SKA$ sequence
composition and synchronous composition of two complex actions have different behaviors. SKA departs from the interleaving approach.

Looking at the models, we have seen the sets of synchronous strings of SKA, and how these are related to the partial orders models. For CKA the models are sets of traces, where a trace is just a set of events of $E$ (i.e., models are just elements of $\mathcal{P}(\mathcal{P}(E))$). Moreover, $E$ is equipped with a dependency relation $\rightarrow$ (no transitivity or acyclicity requirements as with partial orders).

In CKA the dependency relation is not manipulated, it is given. CKA processes specify subsets of events, and each subset has attached the predefined $\rightarrow$ restricted to its events only. In SKA and pomsets the partial order is changed with each application of an operator; e.g., sequential composition adds dependencies. The approach of CKA is similar to that of separation logic where one reasons about a big (given) program by separating it into smaller independent programs. On the other hand, the partial orders model and SKA have a constructivist view where big programs are constructed from smaller programs (i.e. the partial order is constructed).

2.5 Final Remarks on Synchronous Kleene Algebra

We have presented two algebraic structures for modelling synchronous actions. The first, synchronous Kleene algebra, is a combination between Kleene algebra and the synchrony model (i.e., we added the synchrony combinator $\times$). The second is the extension of synchronous Kleene algebras with Boolean tests. This gives more expressive power. We have seen the application of SKAT to reasoning about parallel programs with shared variables in the Hoare-style of Owicki and Gries, and we have hinted to the application of $\ast$-free actions in giving semantics to the $CL$ contract logic.

We have focused on the theoretical aspects of the two new formalisms. Therefore, we have presented standard models (sets of respectively synchronous strings and guarded synchronous strings) and completeness results for the two algebras. The completeness proofs used a combinatorial argument based on two kinds of special automata that we defined. The equivalent of Kleene’s theorem shows how we can obtain for each action a corresponding automaton which accepts the same set of models corresponding to the interpretation of the action.

2.5.1 Related Work

We now make some final discussions of formalisms which are somehow related to SKA, but we could not find strong arguments as we did in Section 2.4.

Shuffle is an operation over regular languages (basically over words) which preserves regularity. Shuffle has been used to model concurrency in [Abr79, Gis84, BÉ96] with a position between the interleaving approach and the partial orders approach. Shuffle is a generalization of interleaving similar to what we discussed for the Mazurkiewicz traces but it does not take into consideration any other relation on the actions/events that it interleaves. We can view $\times$ as some kind of ordered shuffle: the shuffling of two sequences of actions in SKA walks step by step (on the $\cdot$ operation) and shuffles the basic actions found (locally).

mCRL2 is a specification language for distributed systems built in the style of process algebras [GMR+07]. (mCRL2 is the successor of the $\mu$CRL language [GP93]) The semantics is
given as SOS rules and a strong bisimulation relation is defined to capture the equality of processes. An axiomatization of the operators is given and (relative) completeness of the axiomatization w.r.t. the SOS semantics is shown. Recently a tool set has been released [GKM+08].

Many concepts of synchronous Kleene algebras are found in mCRL2. The building blocks of the language are a set of basic actions (parameterized by data types). The basic actions are grouped into sets of basic actions (called multiactions) which are assumed to occur at the same time. The operation on multiactions is the same as $\times$ on $A_B$ in SKA. Over multiactions are defined the basic operators which are essentially the nondeterministic choice, sequence, and conditional (and a few nonessential for process references or for attaching time to a process in the form of a delay). There is no Kleene star concept but recursion is achieved, as in process algebras, through process definitions and process references. The rest of the operators are for parallel composition and synchronization (as in process algebra terminology), and additionally for restriction, blocking, renaming, and communication.

Analyzing an mCRL2 specification means linearization of the specification into what is called a linear process specification (which uses only the basic operators of mCRL2 in a restricted way). This linearization concept is the same as our models for the actions as sets of synchronous strings. The linearizations of mCRL2 are very close to our synchronous strings, except that they need to have more specific notions like the timers on processes (if any was specified in the original mCRL2 process) or the data arguments of the multiactions.

$SKAT$ is a simple and clean formalism, but not as expressive as mCRL2. $SKAT$ is tractable, and when used in the logical formalisms that we mentioned it still yields tractable logics. On the other hand one might need the addition of timing notions or of parameters (like the data types of mCRL2) to the actions, depending on the needs of the particular application domain.

2.5.2 Open problems

Continuations of the work presented here may take two directions: theory oriented and application oriented. From a theoretical point of view it would be interesting to see particular uses of the demanding relation $\prec$ in the lines of thought that we draw in the end of Section 2.2.1.

Details concerning the conflict relation $\#_c$ were not given. An immediate question is how the conflict relation extends to the whole set of SKA actions? The first answer is to say that we add the equational implication (22) to the axioms of SKA (call this axiomatic system $SKA \#_c$).

Two compound actions $\alpha$ and $\alpha'$ are said to be in the conflict relation $\alpha \#_c \alpha'$ iff we cannot deduce $SKA \not\vdash \alpha \times \alpha' = 0$ but we can deduce $SKA \#_c \vdash \alpha \times \alpha' = 0$. This solution relies on the decidability of $SKA \#_c$, which should not be difficult to establish since $\#_c$ is a finite relation (defined on the finite set of basic actions $A_B$). A related question is how does the theory (the results) change if we allow the set $A_B$ to be possibly infinite?

Is there a canonical form for general actions of SKA (or SKAT) similar to what was done in Section 2.2 for the (restricted) $\ast$-free actions? Another interesting result, which is in the spirit of Kleene algebra theory, is to give a representation of automata on synchronous strings in terms of matrices over SKA and give an alternative proof of the completeness Theorem 2.2.28 similar to what is done in [Koz94]. This involves the definition of an operation over matrices to simulate the synchrony product of finite automata over synchronous strings.

In this paper we have focused on the theoretical results of the two new formalisms SKA
and \(SKAT\). The application that we sketched are to logics based on actions (deontic logic with synchronous actions and propositional Hoare logic with synchronous programs). We would like to see more investigations in this direction, with a more logical focus. The work on using the formalism of the \(^*\)-free synchronous actions in the \(CL\) contract logic can be investigated more.

Related to this is a technically challenging problem: consider the equational system defining only the \(^*\)-free actions in Definition 2.2.9 (i.e., axioms (1)-(9) of Table 2.1 together with axioms (14)-(21) of Table 2.2). Is there an algorithm to decide the unification problem for \(^*\)-free synchronous actions? And what is its complexity? A more simple unification problem is to give an algorithm to find the substitution solution to the following:

\[
\alpha \times X = \beta,
\]

for any \(\alpha, \beta\) \(^*\)-free actions \(\alpha\) and \(\beta\), and \(X\) a \(^*\)-free variable in \(SKA\).

Acknowledgements

I would like to thank Sergiu Bursuc for fruitful discussions and many comments on earlier drafts of the work presented in this chapter. Thanks go also to Martin Steffen, Olaf Owe, and Gerardo Schneider for their useful comments. Several reviewers of earlier drafts of this work have contributed with useful comments; special thanks go to the careful and meticulous JLAP reviewers.
Chapter 3

The Contract Logic $\mathcal{CL}$

In this chapter we present the $\mathcal{CL}$ logic which has been designed to represent and reason about contracts (being that software contracts, web services, interfaces, communication protocols, etc). From this point of view, $\mathcal{CL}$ needs to be expressive enough to capture behaviors of such systems. The purpose of this logic is not only to formalize such behaviors, but also to reason about them. Therefore, we aim at a decidable logic so to have hopes for automatic verification using formal tools like model-checking and run-time monitoring.

$\mathcal{CL}$ combines deontic logic (i.e., the logic of legal/normative concepts) [vW51] with propositional dynamic logic (PDL, the logic of actions) [FL77]. The deontic part of $\mathcal{CL}$ can express obligations, permissions and prohibitions over structured actions, as well as what happens when obligations or prohibitions are not respected. The dynamic part of $\mathcal{CL}$ expresses what happens after some action (possibly with complex structure) is performed.

We start by presenting first the deontic part of $\mathcal{CL}$ in Section 3.1. After having established the syntax and the semantics of the deontic modalities (which are applied over the deontic synchronous actions that we have seen in the previous chapter, i.e., in Section 2.2.2) we prove some basic results for the deontic modalities alone, like tree model property and decidability. Afterwards, in Sections 3.2 we give the syntax and semantics (over normative structures) of the dynamic part of $\mathcal{CL}$, hence the full contract logic is presented. We also establish some technical results like tree model property, avoidance of some standard SDL paradoxes, and a series of desired properties (i.e., validities) and unwanted implications (i.e., non-validities) for $\mathcal{CL}$.

### 3.1 Deontic Modalities over Synchronous Actions

In this section we introduce the deontic part of the $\mathcal{CL}$ logic and we work only with deontic modalities over synchronous actions (and the underlying propositional language). The logical expressions are constructed by the grammar of Table 3.1. We call an expression $C$ a (general) contract clause. A contract clause is build using the classical Boolean implication operator $\rightarrow$, where the other operators $\land, \lor, \neg, \leftrightarrow, \top, \oplus$ (exclusive or) are expressed in terms of $\rightarrow$ and $\bot$ as in propositional logic. The building blocks of a contract clause are the propositional constants $\phi$ drawn from a finite set $\Phi_B$ and the deontic modalities $O_C(\alpha), P(\alpha)$, and $F_C(\alpha)$. These represent respectively the obligation, permission, and prohibition of performing a given action $\alpha$. Intuitively $O_C(\alpha)$ states the obligation to perform $\alpha$, and the reparation $C$ in case the obligation
The Contract Logic $\mathcal{CL}$

\[
C := \phi \mid O_C(\alpha) \mid P(\alpha) \mid F_C(\alpha) \mid C \rightarrow C \mid \perp
\]
\[
\alpha := a \mid 0 \mid 1 \mid \alpha \times \alpha \mid \alpha \cdot \alpha \mid \alpha + \alpha
\]

Table 3.1: Deontic modalities over synchronous actions.

is violated, i.e., whenever $\alpha$ is not performed. The reparation may be any contract clause. The modality $O_C(\alpha)$ (resp. $F_C(\alpha)$) represents what is called contrary-to-duty (CTD) (resp. contrary-to-prohibition (CTP)) in dynamic deontic logic. Obligations without reparations are written as $O_\perp(\alpha)$ where $\perp$ (and conversely $\top$) is the Boolean false (respectively true). We usually write $O(\alpha)$ instead of $O_\perp(\alpha)$. Obligations with no reparation are sometimes in the literature called categorical because they must not be violated (i.e., there is no reparation for their violation, thus a violation would give violation of the whole contract). The prohibition modality $F_C(\alpha)$ states that the action $\alpha$ is forbidden and in case the prohibition is violated the reparation $C$ is enforced. Note that it is possible to express nested CTDs and CTPs. Permissions have no reparations associated because they cannot be violated; permissions can only be exercised. The expressions are interpreted over Kripke-like structures which we call normative structures.

Definition 3.1.1 (normative structure) A normative structure is $\mathcal{K}^N = (\mathcal{W}, R_{2A_B}, V, \varrho)$ where:

- $\mathcal{W}$ is a set of worlds (also called states);
- $2^{A_B}$ contains the labels of the structure as sets of basic actions from the finite set $A_B$.
  \[ R_{2A_B} : 2^{A_B} \rightarrow 2^{\mathcal{W} \times \mathcal{W}} \]  returns for each label a partial function on the set of worlds (written as a relation);
- $V : \Phi_B \rightarrow 2^{\mathcal{W}}$ is a valuation function of the propositional constants returning a set of worlds where the constant holds;
- $\varrho : \mathcal{W} \rightarrow 2^{\Psi}$ is a marking function which marks each world with markers from $\Psi = \{ \circ_a, \bullet_a \mid a \in A_B \}$. The marking function respects the restriction that no world can be marked by both $\circ_a$ and $\bullet_a$ (for any $a \in A_B$).

A pointed normative structure is a normative structure with a designated world $i$ (denoted by $< \mathcal{K}^N, i >$).

We denote by an indexed $t$ a node of a tree (or by $r$ the root) and by an indexed $s$ (or $i$ for initial) a state of a normative structure. We use the graphical notation $s \overset{\alpha}{\rightarrow} s'$ for the transitions of the normative structures too. Note that we consider both the trees and the normative structures to have the same set of basic labels $A_B$. In this chapter we are using the trees defined in Section 2.2.5 and the representation of the deontic actions that we gave there as these trees.

$\mathcal{K}^N$ is deterministic as for each label from one world there is at most one successor world. We use deterministic structures because in the deontic realm, as in legal contracts, each action (like “deposit 100$ in bank account”) must have a well determined behavior (i.e., the actions do not have a nondeterministic outcome). The deterministic restriction of Kripke structures was investigated in [BAHP81]. The marking function and the markers are needed to identify obligatory (i.e., $\circ$) and prohibited (i.e., $\bullet$) actions. Markers with different purposes were used in [Mey88] to identify violations of obligations, in [VdM90] to mark permitted transitions, and in [CM07] to identify permitted events.

\[1\] We often omit the brackets and simply write $\mathcal{K}^N, i$ when no confusion can arise.
As an example let us consider the normative structure in Fig. 3.1, which has five states and two constant propositions \( \Phi_B = \{ \phi, \phi' \} \). The valuation function assigns to each proposition a set of states, e.g., \( V(\phi) = \{ s_1, s_3, s_5 \} \).

A first difference between normative structures and the standard Kripke structures is that the labels in normative structures can be sets; e.g., there is one transition labeled with \( \{ p \} \) (i.e., \( R_{2A_B}(\{ d, n \}) = \{(s_1, s_2), (s_3, s_4)\} \)). The second difference is that normative structures have a marking function which in our example marks \( s_2 \) by \( \{ o_a, o_d \} \), \( s_5 \) by \( \{ b \} \), and \( s_3 \) and \( s_4 \) by \( \{ o_p \} \).

In order to relate the semantic domain of our language with the underlying algebra of actions on which the deontic modalities operate, we will give a formal relationship between normative structures and trees through a notion of simulation.

**Definition 3.1.2 (simulation)** For a tree \( T = (N, E, A_B) \) and a normative structure \( K^N = (W, R_{2A_B}, V, g) \) we define a relation \( S \subseteq N \times W \) which we call the simulation of the tree node by the state of the structure:

\[
(t, s) \text{ iff } \forall t \xrightarrow{\gamma} t' \in T, \exists s' \xrightarrow{\gamma'} s' \in K^N \text{ s.t. } \gamma \subseteq \gamma' \wedge t'Ss' \text{ and } \forall s \xrightarrow{\gamma'} s' \in K^N \text{ with } \gamma \subseteq \gamma' \text{ then } t'Ss'.
\]

We say that a tree \( T \), with root \( r \) is simulated by a normative structure \( K^N \) w.r.t. a state \( s \), denoted \( T \sim_s K^N \), iff \( r \sim s \).

Note two differences with the classical definition of simulation: first, the labels of the normative structure may be bigger (a superset) than the labels in the tree because respecting an obligatory action means executing an action which includes it (is bigger). We can drop this condition and consider only \( \gamma = \gamma' \), in which case we call the relation strong simulation and denote by \( S^s \). Second, any transition in the normative structure that can simulate an edge in the tree must enter under the simulation relation. This is because from the state \( s' \) onwards we need to be able to continue to look in the structure for the remaining tree (to see that it is simulated). We can weaken the definition by combining this condition with the one before into:

\[
\forall t \xrightarrow{\gamma} t' \in T, \forall s \xrightarrow{\gamma'} s' \in K^N \text{ with } \gamma \subseteq \gamma' \text{ then } t'Ss'.
\]

We call the resulting relation partial simulation and denote it by \( \check{S} \).

Consider the tree from Fig. 2.9 on page 37 and denote it by \( A^D(\alpha) \) where \( \alpha = p \cdot b + (d \times n) \cdot p \cdot p \). This tree is simulated by the normative structure \( K^N \) of Fig. 3.1 w.r.t. the state \( s_1 \).

It is easy to check that \( r \sim s_1 \); for the edge \( r \xrightarrow{\{ p \}} t_4 \) we find in \( K^N \) the transition \( s_1 \xrightarrow{\{ p, b \}} s_4 \) that respects the inclusion of labels. We also have that \( t_4 \sim s_4 \) since for the only edge \( t_4 \xrightarrow{\{ b \}} t_5 \)
there is the transition \( s_4 \xrightarrow{\{b\}} s_1 \) in \( K^N \) simulating it (the second constraint from the definition of simulation is satisfied trivially for \( t_4 \xrightarrow{\{b\}} t_5 \) because there is no other transition from \( s_4 \) in \( K^N \)). Moreover, for the transition \( s_1 \xrightarrow{\{p\}} s_4 \) the second condition for simulation is satisfied because there is no other transition from \( s_1 \) with a label that includes \( \{p\} \). For the second edge \( r \xrightarrow{\{d,a\}} t_1 \) we find the transition \( s_1 \xrightarrow{\{d,a\}} s_2 \) in \( K^N \) that respects all the simulation conditions (checking these is done as before and we leave this for the reader).

If we were to change the label of the transition \( (s_1, s_4) \) to \( \{p\} \) then the tree would still be simulated but it would also be strongly simulated. Trivially, any strong simulation relation is also a simulation relation.

Consider a simpler example of a tree \( A^D(b) \) interpreting the basic action \( b \). This too is strongly simulated by the structure \( K^N \) w.r.t. \( s_1 \) because, for the only edge of the tree labeled with \( \{b\} \) we find the transition \( s_1 \xrightarrow{\{b\}} s_5 \) which has exactly the same label, and the leaf node of the tree is trivially simulated by \( s_5 \) because there are no edges out of it (note that this holds for any leaf node of a tree). On top, the second simulation condition also holds because the only other transition that has a label that includes \( b \) is \( s_1 \xrightarrow{\{p,b\}} s_4 \) and for this transition \( s_4 \) trivially simulates the leaf node.

Given a tree that is simulated by a normative structure, we can isolate a maximal substructure of the latter that simulates the given tree. As we will see later, it is also useful to identify what parts of the normative structure do not simulate the tree, giving place to what we call the non-simulating remainder of the structure. These notions are formally defined in Definition 3.1.3 and 3.1.4.

**Definition 3.1.3 (maximal simulating structure)** Whenever we have \( T, S, K^N \), we call the structure \( K^{T,i}_{max} = (W', R'_{2\mathbb{A}B}, V', \varrho') \) the maximal simulating structure w.r.t. \( T \) and \( i \) of \( K^N \) the maximal substructure of \( K^N \) respecting the following, where \( K^N = (W, R_{2\mathbb{A}B}, V, \varrho) \):

\[ a. \quad i \in W', \quad V' = V|_{|W'} \text{ and } \varrho' = \varrho|_{|W'} \]

\[ b. \quad \forall t \xrightarrow{\gamma} t' \in T \text{ then } \forall s \xrightarrow{\gamma'} s' \in K^N \text{ s.t. } t S s \wedge \gamma \subseteq \gamma' \wedge t' S s' \]

\[ \text{where } s' \in W' \text{ and } s \xrightarrow{\gamma'} s' \in R'_{2\mathbb{A}B}. \]

**Definition 3.1.4 (non-simulating remainder)** We call the non-simulating remainder of \( K^N \) w.r.t. \( T \) and \( i \) the substructure \( K^{T,i}_{rem} = (W'', R''_{2\mathbb{A}B}, V'', \varrho'') \) of \( K^N \) that is maximal and respects the following:

\[ a. \quad s \xrightarrow{\gamma} s'' \in R''_{2\mathbb{A}B} \text{ iff } s \xrightarrow{\gamma} s'' \in K^{T,i}_{max} \wedge s \in K^{T,i}_{max} \text{ and } \exists s' \xrightarrow{\gamma'} s' \in K^{T,i}_{max}. \]

\[ b. \quad s \in W'' \text{ iff } s \text{ is part of a transition in } R''_{2\mathbb{A}B}, \]

\[ c. \quad V'' = V|_{|W''}, \text{ and } \varrho'' = \varrho|_{|W''}. \]

The formal definition states actually a simple and intuitive notion which we explain more through an example. Take the simple action \( b \) with the trivial tree \( A^D(b) \) which has only one edge \( r \xrightarrow{\{b\}} t_1 \). We have discussed above that this tree is strongly simulated by the structure of Fig. 3.1 w.r.t. node \( s_1 \). The maximal simulating structure \( K^{A^D(b),s_1}_{max} \) is the substructure obtained
Deontic Modalities over Synchronous Actions

Definition 3.1.5 (semantics) We give in Table 3.2 a recursive definition of the satisfaction relation \( \models \) of a formula \( \mathcal{C} \) w.r.t. a pointed normative structure \( < K^N, i > \); it is written \( K^N, i \models \mathcal{C} \) and is read as “\( \mathcal{C} \) is satisfied in the normative structure \( K^N \) at state \( i \)”. We write \( K^N, i \not\models \mathcal{C} \) whenever \( \models \) is not the case. We say that “\( \mathcal{C} \) is globally satisfied in \( K^N \)”, and write \( K^N \models \mathcal{C} \) iff \( \forall s \in K^N, K^N, s \models \mathcal{C} \). A formula is satisfiable iff \( \exists K^N, \exists s \in K^N \) s.t. \( K^N, s \models \mathcal{C} \). A formula is valid (denoted \( \models \)) iff \( \forall K^N, K^N \models \mathcal{C} \).

The propositional connectives have the classical semantics. More interesting and particular to our logic is the interpretation of the deontic modalities. For \( O_C \) the semantics has basically two parts: the second part is just the last line and states that if the obligation is violated (i.e., the complement \( \overline{\alpha} \) of the action is performed) then the reparation \( \mathcal{C} \) should hold at every possible violating state. This definition is similar to the definition of the box modality of PDL only that here it is applied to the complement of the action (i.e., it looks only at the leafs and it uses strong simulation to have exactly the same labels as the ones in the tree of the complement action). The first part of the semantics is the interpretation of the obligation. The first line says how to walk on the structure depending on the tree of the action \( \alpha \). The simulation relation is used because in the structure there may be transitions labeled with actions that are greater than the actions in \( \alpha \), which intuitively, if we do these actions then the obligation of \( \alpha \) is still respected. The simulation relation also takes care that all the choices of an action appear as transitions in the

\[
\begin{align*}
K^N, i \models \varphi & \iff i \in V(\varphi). \\
K^N, i \not\models \bot & \\
K^N, i \models C_1 \rightarrow C_2 & \iff \text{whenever } K^N, i \models C_1 \text{ then } K^N, i \models C_2. \\
K^N, i \models O_C(\alpha) & \iff A^D(\alpha) S_i K^N, \quad \text{and} \\
& \forall t \overset{\gamma}{\rightarrow} t' \in A^D(\alpha), \forall s \overset{\gamma}{\rightarrow} s' \in K^N \text{ s.t. } t S s \land \gamma \subseteq \gamma' \quad \text{then} \forall a \in A_B \text{ if } a \in \gamma \text{ then } o_a \in g(s'), \quad \text{and} \\
& \forall s \overset{\gamma}{\rightarrow} s' \in K^N A^D(\alpha), \quad \text{then} \forall a \in A_B \text{ if } a \in \gamma' \text{ then } o_a \not\in g(s'), \quad \text{and} \\
& K^N, s \models C \quad \forall s \in K^N \text{ with } t S^* s \land t \in \text{leafs}(A^D(\overline{\alpha})). \\
K^N, i \models F_C(\alpha) & \iff A^D(\alpha) S_i K^N \quad \text{then} \\
& \forall \sigma \in A^D(\alpha) \text{ a final path s.t. } \sigma S_i K^N, \quad \forall t \overset{\gamma}{\rightarrow} t' \in \sigma, \forall s \overset{\gamma}{\rightarrow} s' \in K^N \text{ with } t S s \land \gamma \subseteq \gamma' \quad \text{then} \\
& \forall a \in A_B \text{ if } a \in \gamma \text{ then } o_a \in g(s') \quad \text{and} \\
& K^N, s \models C \quad \forall s \in K^N \text{ with } t S s \land t \in \text{leafs}(\sigma). \\
K^N, i \models P(\alpha) & \iff A^D(\alpha) S_i K^N, \quad \text{and} \\
& \forall t \overset{\gamma}{\rightarrow} t' \in A^D(\alpha), \forall s \overset{\gamma}{\rightarrow} s' \in K^N \text{ s.t. } t S s \land \gamma \subseteq \gamma' \quad \text{then} \\
& \forall a \in A_B \text{ if } a \in \gamma \text{ then } o_a \not\in g(s'). \quad \text{Table 3.2: Semantics for the deontic modalities over synchronous actions.}
\end{align*}
\]

from \( K^N \) by deleting the states \( s_2 \) and \( s_3 \) (and the associated transitions too) as well as the transition \( s_4 \overset{b}{\rightarrow} s_1 \). The non-simulating reminder structure is obtained from \( K^N \) by deleting the states \( s_3, s_4, \) and \( s_5 \). For the more complex action from Fig. 2.9 the non-simulating reminder is the substructure that has only the worlds \( s_1 \) and \( s_5 \) and the transition between them.

We have now all the necessary definitions to introduce the semantics of our deontic modalities over synchronous actions.
structure. The second and third lines mark all the transitions (their ending states) of the structure which simulate edges from the tree with markers $\circ_\alpha$ corresponding to the labels of the simulated edge. This is needed both for the proof of the synchrony property $O_C(\alpha) \land O_C(\beta) \rightarrow O_C(\alpha \times \beta)$ and also in proving $O_C(\alpha) \rightarrow \neg F_C(\alpha)$ which relates obligations and prohibitions. Lines four and five ensure that no other reachable relevant transitions of the structure (i.e., from the non-simulating remainder structure) are marked with obligation markers $\circ$. This is essential in the proof of the key Lemma 3.1.8 of the synchrony result given in Theorem 3.1.17.

For the $F_C$ modality we use partial simulation $\tilde{S}_i$ in order to have our intuition that if an action is not present as a label of an outgoing transition of the model then the action is by default considered forbidden. In the second line we consider all final paths in order to respect the intuition that prohibition of a choice must prohibit all, i.e., $F(a + b) = F(a) \land F(b)$. Note that we are interested only in final paths simulated by the structure because for the other paths some of the transitions are missing in the structure and thus there is some action on the sequence which is forbidden. In the third line we consider all the edges on each final path in order to respect the intuition that forbidding a sequence means forbidding all the actions on that sequence. For a chosen edge we look for all the transitions of the normative structure from the chosen node which have a label greater than the label of the edge; this is in order to respect the intuition that forbidding an action implies forbidding any action that is greater, i.e., $F(a) \rightarrow F(a \times b)$. The last line states that if the prohibition is violated then the reparation $C$ must hold in all the states where the violation is observed.

The semantics of $P$ specifies that $\bullet$ markers should not be present in order to capture the principle that what is not forbidden is permitted. The semantics of $O$, $P$, or $F$ hint at the trace-based semantics of Process Logic [Pra79] and to some extent to the modalities of [VdM90].

### 3.1.1 Properties of the deontic modalities

The semantics of the deontic modalities is rather involved; it is based on an algebraic formalism for the actions which are interpreted as rooted trees. The information in the trees (compared to sets of traces [Pra79]) is used by the particular notion of simulation relation to know how to walk on the normative structure in the search of the markings to determine the truth value of the deontic modality. The rest of the complications in the semantics are necessary for capturing several intuitive properties of the deontic modalities which we discuss in this section.

The following validities are the counterparts of the ones found in SDL only that here they are in an ought-to-do setting where the deontic modalities are applied over actions. Related to each property we give examples taken (or adapted) from the contract that was used in [PPS07] as a case study for model checking legal contracts. (This example is reproduced in Appendix B.) The following examples give intuitions for the logical validities of Proposition 3.1.6:

- **Obligation of an action implies that the action is permitted:**
  $$\models O_C(\alpha) \rightarrow P(\alpha)$$

  E.g.: “Client is obliged to pay” then “Client has the right to pay”.

- **Permission of performing an action implies that the action is not forbidden:**
  $$\models P(\alpha) \rightarrow \neg F_C(\alpha)$$
E.g.: “Provider has the right to alter personal data” then “Provider is not forbidden to alter personal data”.

- If two action expressions represent the same action then the obligation of one action should imply the obligation of the other action:
  \[ \text{if } \alpha = \beta \text{ then } \models O_C(\alpha) \leftrightarrow O_C(\beta) \]

- The obligation of the violating action cannot appear in a contract (\( \models \neg O_C(0) \)); obligation of terminating the contract can be modeled. The obligation to do nothing can be trivially inserted in any contract (\( \models O_C(1) \)).

**Proposition 3.1.6 (validities)** The following statements hold:

\[
\begin{align*}
\models \neg O_C(0) & \quad (3.1) & \models O_C(\alpha) \rightarrow P(\alpha) & \quad (3.4) \\
\models O_C(1) & \quad (3.2) & \text{if } \alpha = \beta \text{ then } \models O_C(\alpha) \leftrightarrow O_C(\beta) & \quad (3.5) \\
\models P(\alpha) \rightarrow \neg F_C(\alpha) & \quad (3.3) & \models O_C(\alpha) \rightarrow \neg F_C(\alpha) & \quad (3.6)
\end{align*}
\]

**Proof:** For the proof of (3.1), i.e., \( \models \neg O_C(0) \), we need to show that there is no model which makes \( O_C(0) \) true. This is because of the definition of \( \neg O_C(0) \) as \( O_C(0) \rightarrow \bot \) which is true only if \( O_C(0) \) is false. By *reductio ad absurdum* suppose that it exists a model which makes \( O_C(0) \) true. This means (by the definition of the semantics of \( O \)) that the tree interpreting \( 0 \) must be simulated by the model. But this is not possible because of the special label \( \Lambda \) appearing in the tree of \( 0 \) which does not appear in the labels of the normative structures.

To prove (3.2), i.e., \( \models O_C(1) \), take any normative structure. The tree interpreting \( 1 \) is trivially simulated by any normative structure because the only edge of the tree is labeled with the empty set and thus any transition of the structure simulates the edge. The second condition in the semantics of \( O \) is satisfied as there is no basic label \( a \) in the label of the edge. It is clear that any edge on the first level of the structure enters into the maximal simulating structure and therefore the non-simulating remainder \( K_{\text{rem}}(1) \) is empty, and the third condition is trivially satisfied. Because \( T = 0 \) then there is no state \( s \) to satisfy the requirements of the last condition and thus it is trivially satisfied too. Note that \( O_C(1) \) is valid only in the *reflexive* structures. This means that the structures must satisfy the property that from each state there is a transition to itself.

For the proof of (3.4), i.e., \( \models O_C(\alpha) \rightarrow P(\alpha) \), take an arbitrary pointed normative structure \( K_N,i \) which makes \( O_C(\alpha) \) true. This means that \( A^P(\alpha) S_i K_N \). This is the first part from the semantics of \( P(\alpha) \). Moreover, from the semantics of \( O_C(\alpha) \) we have that \( \forall t \rightarrow t', t' \in A^P(\alpha), \forall s \rightarrow s' \in K_N \), tSs ∧ γ ⊆ γ' then \( \forall a \in \gamma \) we have \( o_a \in g(s') \). Because of the restriction on the marking function we get \( \forall a \in \gamma \) we have \( \bullet_a \notin g(s') \). This makes the second requirement in the semantics of \( P(\alpha) \).

For the proof of (3.3), i.e., \( \models P(\alpha) \rightarrow \neg F_C(\alpha) \), we use *reductio ad absurdum* and assume that it exists a pointed structure \( K_N,i \) which satisfies \( P(\alpha) \) and also \( F_C(\alpha) \). From the semantics of \( P \), knowing that \( S \subseteq \tilde{S} \), we conclude that \( A^P(\alpha) \tilde{S} K_N \). Now take any final path \( \sigma \) in the tree of \( \alpha \); from the semantics of \( P \) it holds that \( \sigma S K_N \). Moreover, for any edge \( t \rightarrow t' \in \sigma \) it holds, by the semantics of \( P \), that \( \forall s \rightarrow s' \in K_N \) with \( tSs \) then \( \forall a \in \gamma \), \( \bullet_a \notin g(s') \). But
Figure 3.2: Examples for natural obligations.

the semantics of $F_C(\alpha)$ requires that $\forall a \in \gamma'$ then $\bullet_a \in g(s')$ which is not possible as $\gamma \subseteq \gamma'$ (i.e., it exists at least one $\bullet_a \notin g(s')$ with $a \in \gamma'$).

To prove (3.5) notice that the semantics of $O$ is based on the interpretation of the actions as trees. Therefore, because the actions are equal, the tree interpretations denote the same tree (up to isomorphism). Thus, the semantics for $O_C(\alpha)$ is the same as that for $O_C(\beta)$ because they are working with the same tree $A^D(\alpha) = A^D(\beta)$.

The proof of (3.6) can be obtained from (3.4) and (3.3).

In Theorem 3.1.17 (see below) we give a property for obligations over synchronous actions. The theorem states that if "there exists an obligation to do action $\alpha$ and there is also an obligation to do action $\beta$" (in the same current world) then we should be able to infer that "there is an obligation to do both actions $\alpha$ and $\beta$ at the same time". This property does not hold for general obligations (with the definition that we gave before), but only for some restricted obligations, which we call natural obligations.

The purpose of natural obligations is not necessarily a technical one but also a practical one. The naturalness constraint refers mainly to choices of actions; when deciding which of the actions to choose the model should not influence the decision. Consider the model of Fig. 3.2-(i) in which $O(a + b)$ holds at state $s_0$. Change this model by adding a $c$ to the left label and a $d$ to the right label, as in Fig. 3.2-(ii). $O(a + b)$ still holds at state $s_0$, but is not a natural obligation; intuitively, when deciding which of $a$ or $b$ to choose one needs to take into account the two distinct actions $c$ and $d$. If we were to add the same label $c$ to both branches then the naturalness constraint is satisfied, as one does not care about the extra action $c$ when choosing.

**Definition 3.1.7 (natural obligations)** An obligation $O_C(\alpha)$ is called natural iff in addition to the semantics of Definition 3.1.5 the following naturalness constraint is respected:

$$\exists \gamma \text{ s.t. } A^D(\alpha \times \gamma) S^s K^{A^D(\alpha),i}_\max S^s A^D(\alpha \times \gamma)$$

(3.7)

As a side remark, note that the naturalness constraint amounts to the equality of two trees, or equivalently, to the equality of two deontic actions (see Lemma 3.1.24).

The aim now is to prove an important result for obligations applied over synchronous actions and conjunction of obligations. To prove this result in Theorem 3.1.17 we need a series of helper results which we develop in the following lemmas.

The following lemma guarantees that the conjunction of obligations implies equality between the structures of the conjuncts, or strict inclusion of one into the other.
Lemma 3.1.8 If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then $K^I_{SKA}(\alpha)_j = K^I_{SKA}(\beta)_j$ otherwise $K^I_{SKA}(\alpha)_j \subseteq K^I_{SKA}(\beta)_j$ otherwise $K^I_{SKA}(\alpha)_j \supset K^I_{SKA}(\beta)_j$.

Proof: Take an arbitrary pointed structure $K^N, i$ and suppose $K^N, i \models O_C(\alpha) \land O_C(\beta)$. The proof of this lemma uses reductio ad absurdum and is based on the fact that lines two and three in the semantics of obligation add $\circ$ markers to the states, and line four removes $\circ$ markers thus resulting in a contradiction.

If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then $K^N, i \models O_C(\alpha)$ and $K^N, i \models O_C(\beta)$. From the first we have by the semantics that $I_{SKA}(\alpha) \subseteq K^N$ which means that there exists the maximal simulating structure $K^I_{SKA(\alpha),i}$. From the semantics of $O_C(\beta)$ we obtain similarly $K^I_{SKA(\beta),i}$. Both maximal simulating structures are substructures of the same $K^N$.

Suppose that there exists a transition $k \overset{\gamma}{\rightarrow} k' \in K^I_{SKA(\alpha),i}$ s.t. $k \overset{\gamma}{\rightarrow} k' \notin K^I_{SKA(\beta),i}$ and there is a transition $s \overset{\gamma'}{\rightarrow} s' \in K^I_{SKA(\beta),i}$ s.t. $s \overset{\gamma'}{\rightarrow} s' \notin K^I_{SKA(\alpha),i}$. Without loss of generality we will work with the transition $k \overset{\gamma}{\rightarrow} k'$ which from the semantics of $O_C(\alpha)$ we have that $\forall a \in A_B$ if $a \in \gamma$ then $a \in g(k')$. On the other hand the transition $k \overset{\gamma}{\rightarrow} k'$ is not part of $K^I_{SKA(\beta),i}$ and because $k \in K^I_{SKA(\beta),i}$ and we know that it exists at least one transition in $K^I_{SKA(\beta),i}$ (for example the transition $s \overset{\gamma'}{\rightarrow} s'$) then it means that $k \overset{\gamma}{\rightarrow} k' \in K^I_{SKA(\beta),i}$. By the semantics of $O_C(\beta)$ we know that $\forall a \in A_B$ if $a \in \gamma$ then $a \notin g(k')$. This results in a contradiction and therefore the initial supposition is wrong.

Corollary 3.1.9

a. If $K^I_{SKA(\alpha),i} = K^I_{SKA(\beta),i}$ then

1. $TK^I_{SKA(\alpha),i} = TK^I_{SKA(\beta),i}$ and
2. $K^I_{SKA(\alpha),i} = K^I_{SKA(\beta),i}$.

b. If $K^I_{SKA(\alpha),i} \subseteq K^I_{SKA(\beta),i}$ then

1. $TK^I_{SKA(\alpha),i} \subseteq TK^I_{SKA(\beta),i}$ and
2. $\forall k \overset{\gamma}{\rightarrow} k' \in K^I_{SKA(\alpha),i}$ either $k \overset{\gamma}{\rightarrow} k' \in K^I_{SKA(\beta),i}$ or $k \overset{\gamma}{\rightarrow} k' \in K^I_{SKA(\beta),i}$.

c. If $K^I_{SKA(\alpha),i} \supset K^I_{SKA(\beta),i}$ then

the same as before but interchange $\alpha$ with $\beta$.

Lemma 3.1.10 For any $\alpha, \beta, \gamma', \gamma'' \in \mathcal{A}^D$ if $I_{SKA}(\alpha \times \gamma') = I_{SKA}(\beta \times \gamma'') = T$ then $\exists \gamma'' \in \mathcal{A}^D$ s.t. $T = I_{SKA}(\alpha \times \beta \times \gamma''')$.

Proof: From the completeness result of the algebra of actions we get that because $I_{SKA}(\alpha \times \gamma') = I_{SKA}(\beta \times \gamma'')$ we have $\alpha \times \gamma' = \beta \times \gamma'' = \theta$. We need to prove that $\exists \gamma''' \in \mathcal{A}^D$ s.t. $\alpha \times \beta \times \gamma''' = \theta = \alpha \times \gamma' = \beta \times \gamma''$ which by the completeness results means that $T = I_{SKA}(\alpha \times \beta \times \gamma''')$.

The interpretation function $I$ is applied to the canonical almost normal form, and therefore we consider the actions $\alpha \times \gamma'$ and $\alpha \times \beta \times \gamma'''$ to be in $canf$. Because the canonical form is
defined inductively it is w.l.o.g. that we look only at the first levels of the actions (i.e., only at the $\times$-actions $\alpha_i^j$ of the canonical form). For a simple notation we denote the $\times$-actions on the first level of $\alpha$ by $\alpha_1, \alpha_2, \ldots, \alpha_k$; note that there are $k$ actions in total. For the action $\beta$ we denote the $\times$-actions on the first level by $\beta_1, \beta_2, \ldots, \beta_l$. For the action $\theta$ we denote the $\times$-actions by $\tau_i$.

To prove the lemma we use the proof principle *reductio ad absurdum* and suppose that $\alpha \times \beta \times \gamma''' \neq \theta$ is the case. According to the above this supposition is equivalent to saying that the $\times$-actions on the first level of $\theta$ are not constructed from the actions on the first level of $\alpha \times \beta$. This may be from several reasons.

First consider that a $\times$-action of $\alpha \times \beta$, say $\alpha_1 \times \beta_1$ is not contained in any of the $\times$-actions $\tau_i$ on the first level of $\theta$. Consider $\tau^i_{\alpha_1}$ to be those $\tau_i$ which contain $\alpha_1$; and similarly consider $\tau^j_{\beta_1}$ those $\tau_j$ which contain $\beta_1$. From the supposition we know that $\beta_1$ does not appear in any of the $\tau^{i}_{\alpha_1}$; and similarly $\alpha_1$ does not appear in any $\tau^{j}_{\beta_1}$. From the hypothesis $\theta = \beta \times \gamma''$ we know that in all $\tau$ it appears one of the $\beta_j$ $\times$-actions. This means that in each of the $\tau^{i}_{\alpha_1}$ it appears one of the $\beta_j$ where $j \neq 1$. Consider w.l.o.g. one of these actions $\tau^{i}_{\alpha_1} = \alpha_1 \times \beta_2 \times \gamma$ for some $\gamma$ which may also be empty. From the same hypothesis $\theta = \beta \times \gamma''$ and knowing that $\alpha_1 \times \beta_2 \times \gamma$ is a $\times$-action on the first level of $\theta$ then it means that $\alpha_1 \gamma$ is an action on the first level of $\gamma''$. This means that between the actions $\tau$ of the first level of $\theta$ there exists each of the actions $\alpha_1 \times \gamma \times \beta_j$ with $j \neq 2$ (because we already have the index 2). In other words, the action $\alpha_1 \times \gamma$ must be combined with any of the actions $\beta_j$ including $\beta_1$.

We thus obtained the contradiction (i.e., there exists an action $\tau$ which contains $\alpha_1 \times \beta_1$). Therefore, each of the $\alpha_i \times \beta_j$ of $\theta = \alpha \times \beta \times \gamma'''$ are contained in $\tau_i$. In other words we have proven that all the $\times$-actions on the first level of the action $\alpha \times \beta$ are found among the $\times$-actions on the first level of $\theta$. Moreover, the discussion above also proves that $\forall \tau \in \theta, \tau = \alpha_i \beta_j \gamma$; which says that there is no $\times$-action on the first level of $\theta$ which does not contain an action from the first level of $\alpha \times \beta$.

The only way to still have the (bad) supposition is to say that it is not the case that for all pairs $\alpha_i \beta_j$ there exists a same $\gamma$ such that $\alpha_i \times \beta_j \times \gamma = \tau$ is a $\times$-action on the first level of $\theta$. To explain it differently, this supposition wants to contradict the second $\times$-operator in the conclusion of the lemma $(\alpha \times \beta) \times \gamma''$ which by the definition it must be that for each $\gamma$ an action on the first level of $\gamma''$ it must be combined with each action $\alpha_i \times \beta_j$ of $\alpha \times \beta$.

We take an arbitrary pair $\alpha_i \times \beta_j$, say $\alpha_1 \times \beta_1$ and w.l.o.g. suppose it has some extra action $\gamma_\kappa$ which may be also empty. Thus $\alpha_1 \times \beta_1 \times \gamma_\kappa$ is an action on the first level of $\theta$. From the hypothesis $\beta \times \gamma'' = \theta$ and knowing that $\beta_1$ is combined with the action $\alpha_1 \times \gamma_\kappa$ it implies that all other $\beta_j$ with $j \neq 1$ must be combined with the same action. Therefore, the following are also actions $\tau$: $\alpha_1 \times \beta_2 \times \gamma_\kappa$, $\ldots$, $\alpha_1 \times \beta_l \times \gamma_\kappa$. On the other hand, from the hypothesis $\alpha \times \gamma' = \theta$ and knowing that $\alpha_1 \times \gamma_\kappa$ is a $\tau$ action it means that all other $\alpha_i$ actions must be combined with $\beta_1 \times \gamma_\kappa$. Therefore, we also have as $\tau$ actions: $\alpha_2 \times \beta_1 \times \gamma_\kappa$, $\ldots$, $\alpha_k \times \beta_1 \times \gamma_\kappa$.

We continue to apply recursively the same reasoning on the new deduced actions like $\alpha_2 \times \beta_1 \times \gamma$ and we obtain in the end that all the actions $\alpha_k \times \beta_k$ appear among the actions $\tau$ on the first level of $\theta$ combined with the same action $\gamma_\kappa$. Thus, the second false supposition is contradicted.

The last way of contradicting the lemma is trivial and it suppose that it is not the case that all the $\tau$ actions of $\theta$ come from combination by $\times$ with the actions $\alpha_i \times \beta_j$. More clearly this tries to say that there exit other $\tau$ actions that do not follow the pattern deduced by the first two
Lemma 3.1.11 For any $K^N$ a normative structure and $\alpha, \beta$ two distinct actions we have that if $K^N, i \models O_{C}(\alpha) \land O_{C}(\beta)$ then $I_{SKA}(\alpha \times \beta) \ S, K^N$.

Proof: We use Lemma 3.1.10 and mainly the naturalness constraint on obligations from Definition 3.1.7.

From the statement of the lemma $K^N, i \models O_{C}(\alpha) \land O_{C}(\beta)$ by applying the Lemma 3.1.8 we get that $K_{max}^{I_{SKA}(\alpha), i} = K_{max}^{I_{SKA}(\beta), i}$ (we treat the two cases with strict inclusion at the end). This implies (see Corollary 3.1.9) that the corresponding trees which unfold these maximal substructures are the same; i.e., $TK_{max}^{I_{SKA}(\alpha), i} = TK_{max}^{I_{SKA}(\beta), i} = TK_{max}^N$.

Moreover, from the hypothesis of the lemma we get that $K^N, i \models O_{C}(\alpha)$ and $K^N, i \models O_{C}(\beta)$. Considering the naturalness constraint it implies that:

$$\exists \gamma' \ \text{ s.t. } I_{SKA}(\alpha \land \gamma') = TK_{max}^{I_{SKA}(\alpha), i}$$

$$\exists \gamma'' \ \text{ s.t. } I_{SKA}(\beta \land \gamma'') = TK_{max}^{I_{SKA}(\beta), i}$$

From these and knowing that the maximal simulating structures are the same we get that $I_{SKA}(\alpha \land \gamma') = I_{SKA}(\beta \land \gamma'') = TK_{max}$. By applying the Lemma 3.1.10 we get that $TK_{max} = I_{SKA}(\alpha \land \beta \land \gamma'')$.

Following the Definition 3.1.2 of the simulation relation $S_i$, in order to prove the conclusion $I_{SKA}(\alpha \land \beta) \ S_i K^N$ we need to prove that:

1. $\forall t \xrightarrow{\gamma} t' \in I_{SKA}(\alpha \land \beta), \exists i \xrightarrow{\gamma'} k' \in K^N$ s.t. $\gamma \subseteq \gamma'$ and $t' \ S k'$.
2. $\forall i \xrightarrow{\gamma'} k' \in K^N$ with $\gamma \subseteq \gamma'$ then $t' \ S k'$.

Using the results of the previous lemmas the proofs of (1) and (2) become simple. As $I_{SKA}(\alpha \land \beta \land \gamma'') = TK_{max}^N$ which is the tree unfolding of the substructure $K_{max}^N = K_{max}^{I_{SKA}(\alpha), i} = K_{max}^{I_{SKA}(\beta), i}$ of $K^N$, then it is simple to see that for any edge $r \xrightarrow{\gamma} t' \in I_{SKA}(\alpha \land \beta)$ there is a transition $i \xrightarrow{\gamma'} k' \in TK_{max}^N$ which clearly $\gamma \subseteq \gamma'$ depending on $\gamma''$. Therefore, $i \xrightarrow{\gamma'} k' \in K_{max}^N$ and thus $i \xrightarrow{\gamma'} k' \in K^N$. The fact that $t' \ S k'$ is thus is obvious by applying a similar recursive reasoning and descending one level in the tree. Note that the recursive reasoning stops when the tree node $t'$ has no more children (i.e., no more edges $t' \xrightarrow{\gamma''} t''$ exist in $I_{SKA}(\alpha \land \beta)$); and this is always the case as the tree is finite.

For proving (2) we use a similar recursive reasoning as before. From the condition $\gamma \subseteq \gamma'$ it implies that $\gamma' = \gamma \land \gamma''$. Because $\gamma'$ is a label of a transition in $I_{SKA}(\alpha \land \beta \land \gamma'')$ then $\gamma' = \alpha \land \beta \land \gamma''$. Because it contains $\alpha$ it enters under the application of the hypothesis $I_{SKA}(\alpha) \ S_i K^N$ (and similarly because it contains $\beta$ we can apply $I_{SKA}(\beta) \ S_i K^N$). Applying the hypothesis leads to the fact there there are the edges $r \xrightarrow{\gamma} t'_{\alpha} \in I_{SKA}(\alpha)$ and $r \xrightarrow{\gamma} t'_{\beta} \in I_{SKA}(\beta)$.
$I_{SKA}(\beta)$ with $t_\alpha' S k'$ and $t_\beta' S k'$. On the other hand $t'$ comes from the combination of the two $t_\alpha'$ and $t_\beta'$ and thus a simple recursive reasoning gives $t' S k'$. The recursive reasoning stops again when the node $t'$ has no more children.

Note that if we consider inclusion among the maximal simulating structures (instead of the equality as we did) then the discussion above does not change. The $TK_{max}^{SKA(\alpha \otimes \beta), i}$ is the same as the interpretation $I_{SKA}(\alpha \times \beta \times \gamma^m)$.

In what follows we present two corollaries of Lemmas 3.1.8 and 3.1.11: the first shows what is the maximal simulating structure with respect to $I_{SKA}(\alpha \times \beta)$; and the second states that the obligation of $\alpha \times \beta$ respects the naturalness constraint. Corollary 3.1.12 is used in the proofs of both Lemma 3.1.14 and Lemma 3.1.15.

**Corollary 3.1.12** For any $K^N$ a normative structure and $\alpha, \beta$ two distinct actions we have that if $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then either

$K_{max}^{SKA(\alpha), i} = K_{max}^{SKA(\beta), i} = K_{max}^{SKA(\alpha \otimes \beta), i}$ or

$K_{max}^{SKA(\alpha), i} \subset K_{max}^{SKA(\beta), i} = K_{max}^{SKA(\alpha \otimes \beta), i}$ or

$K_{max}^{SKA(\beta), i} \subset K_{max}^{SKA(\alpha), i} = K_{max}^{SKA(\alpha \otimes \beta), i}$.

**Corollary 3.1.13** If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then $O(\alpha \times \beta)$ is a natural obligation.

**Lemma 3.1.14** If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then

$\forall t \xrightarrow{s} t' \in I_{SKA}(\alpha \times \beta)$ and $\forall s \xrightarrow{\gamma'} s' \in K^N$ s.t. $t S s \land \gamma \subseteq \gamma'$ is the case that

$\forall a \in A_B$ if $a \in \gamma$ then $o_a \in O(s')$.

**Proof:** It is simple to see, by looking at Definition 3.1.3, that all transitions $s \xrightarrow{\gamma'} s'$ mentioned in the lemma make up exactly the maximal simulating structure $K_{max}^{SKA(\alpha \otimes \beta), i}$. By Corollary 3.1.12 this is the same as the maximal simulating structures for $I_{SKA}(\alpha)$ and $I_{SKA}(\beta)$.

To finish the proof we take one arbitrary edge $t \xrightarrow{\alpha \otimes \beta_k} t' \in I_{SKA}(\alpha \times \beta)$ and one arbitrary transition $s \xrightarrow{\gamma} s' \in K_{max}^{SKA(\alpha \otimes \beta), i}$ s.t. $t S s$ and $\gamma = \alpha_x \times \beta_x \times \gamma'$ where $\gamma'$ may also be 1. These satisfy the conditions in the lemma. The edge $t \xrightarrow{\alpha \otimes \beta_k} t'$ comes from the combination of two edges $t \xrightarrow{\alpha} t' \in I_{SKA}(\alpha)$ and $t \xrightarrow{\beta_k} t' \in I_{SKA}(\beta)$. On the other hand we have for the transition $s \xrightarrow{\gamma} s'$ that both $\alpha_x \subseteq \gamma$ and $\beta_x \subseteq \gamma$ hold. This means that we can apply the hypothesis of the lemma (i.e., apply the definition for $O_C$ to both $O_C(\alpha)$ and $O_C(\beta)$) to get that $o_a \in O(s'), \forall a \in \alpha_x$ and $o_a \in O(s'), \forall a \in \beta_x$ (because the definition says that for all transitions this happens). This implies the result of the lemma, i.e., $o_a \in O(s'), \forall a \in \alpha_x \times \beta_x$.

**Lemma 3.1.15** If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then

$\forall s \xrightarrow{\gamma} s' \in K_{max}^{SKA(\alpha \otimes \beta), i}$ then $\forall a \in A_B$ if $a \in \gamma$ then $o_a \notin O(s')$.

**Proof:** Following from Corollary 3.1.12 is that $K_{max}^{SKA(\alpha \otimes \beta), i} = K_{max}^{SKA(\alpha), i} = K_{max}^{SKA(\beta), i}$ (the cases for inclusion are treated at the end). From the hypothesis $K^N, i \models O_C(\alpha)$ we have that

$\forall s \xrightarrow{\gamma} s' \in K_{max}^{SKA(\alpha), i}$ then $\forall a \in A_B$ if $a \in \gamma$ then $o_a \notin O(s')$ which makes our proof goal also true by replacing $K_{max}^{SKA(\alpha), i}$ with its equal $K_{max}^{SKA(\alpha \otimes \beta), i}$.

In the case when $K_{max}^{SKA(\alpha), i} \subset K_{max}^{SKA(\beta), i} = K_{max}^{SKA(\alpha \otimes \beta), i}$ then we work as before but consider the structure for $\beta$ instead.
Lemma 3.1.16 If $K^N, i \models O_C(\alpha) \land O_C(\beta)$ then 
$K^N, s \models C \ \forall s \in N$ with $t S^* s \land t \in \text{leaves}(I_{SKA}(\alpha \times \beta))$.

Proof: The conclusion of the lemma should be read as: the formula $C$ holds in all those states $s \in K^N$ which can be reached by “following” the tree interpretation of the action complement $\alpha \times \beta$ to the leafs. By “to follow” we mean that the normative structure simulates strictly the tree $I_{SKA}(\alpha \times \beta)$. The simulation must be strict so that we follow exactly the tree.

Recall the Definition 2.2.17 of the action complement. The complement of a compound action $\overline{\alpha}$ works on each level of the complemented action $\alpha$. For the proof of this lemma it is enough to look at the behavior for only the first level, and for the rest we apply a similar recursive reasoning. Moreover, note that we need to look only at the leafs of the trees (i.e., at the states from the end of the final paths of the tree interpretation of the complemented action). Thus, the first level in the complement contains the choice $+_{\gamma \in \Gamma} \gamma$ (defining the full branches; we look at the other full branches when we reason recursively at lower levels of the tree).

Thus, we need to prove that $\forall t \xrightarrow{\gamma} t' \in I_{SKA}(+_{\gamma \in \Gamma} \gamma)$ with $\gamma \in A'_x$ a $\times$-action s.t. $\forall \alpha^i_x \times \beta^j_x \subseteq \alpha^i_x \times \beta^j_x$ a $\times$-action on the first level of the tree of the complemented action $\alpha \times \beta$ we have that $\alpha^i_x \times \beta^j_x \not\subseteq \gamma$ then it is the case that if $\exists s \xrightarrow{\gamma} s' \in K^N$ then $K^N, s' \models C$. Take an arbitrary transition $t \xrightarrow{\gamma} t'$ for which the above hold and for which $\exists s \xrightarrow{\gamma} s' \in K^N$ and we show that $K^N, s' \models C$.

From the condition $\forall \alpha^i_x \times \beta^j_x, \alpha^i_x \times \beta^j_x \subseteq \gamma$ we can conclude that either $\forall \alpha^i_x, \alpha^i_x \not\subseteq \gamma$ or $\forall \beta^j_x, \beta^j_x \not\subseteq \gamma$. This is done by using the proof principle redactio ad absurdum and we suppose that neither of the $\forall \alpha^i_x, \alpha^i_x \not\subseteq \gamma$ nor $\forall \beta^j_x, \beta^j_x \not\subseteq \gamma$ hold. This means that $\exists \alpha^i_x \times \beta^j_x \subseteq \gamma \land \beta^j_x \subseteq \gamma$ which implies that $\alpha^i_x \times \beta^j_x \subseteq \gamma$. By looking again at the definition of the $\times$ operation we see that $\alpha^i_x \times \beta^j_x$ must be an action among the $\alpha^i_x \times \beta^j_x$. Therefore, the conclusion that we have just drawn before enters into contradiction with the initial condition $\forall \alpha^i_x \times \beta^j_x, \alpha^i_x \times \beta^j_x \not\subseteq \gamma$.

By using one of the hypothesis of the lemma, say $K^N, i \models O_C(\alpha)$ we conclude from the definition of the semantics of $O_C$ that the transition that we work with $t \xrightarrow{\gamma} t'$ respects the fact that $\forall \alpha^i_x, \alpha^i_x \not\subseteq \gamma$ and thus in the end state of the transition $s \xrightarrow{\gamma} s' \in K^N$ we have $K^N, s' \models C$. This is the conclusion of the lemma.

Theorem 3.1.17 (synchrony property) For natural obligations we have:

$\models O_C(\alpha) \land O_C(\beta) \rightarrow O_C(\alpha \times \beta)$.

Proof: We need to prove that $K^N, i \models O_C(\alpha \times \beta)$ under the assumption $K^N, i \models O_C(\alpha) \land O_C(\beta)$. Using Lemma 3.1.11 we have that $A^D(\alpha \times \beta) S_i K^N$ which is the first requirement in the semantics of $O_C$. For the proof of Lemma 3.1.11 the naturalness constraint is essential. The proofs of Lemma 3.1.14 and Lemma 3.1.15 are also based on the naturalness constraint. These two lemmas give the second and the third requirement in the semantics of $O_C(\alpha \times \beta)$. The last requirement is proven as Lemma 3.1.16.

We now show how the above result can be generalized to the conjunction of obligations containing different reparations.

Corollary 3.1.18 $\models O_{C_1}(\alpha) \land O_{C_2}(\beta) \rightarrow O_{C_1 \lor C_2}(\alpha \times \beta)$.
The following corollary points out conflicts that are avoided in the logic because of the semantics. These are usual requirements when reasoning about legal contracts. A contract with two clauses “Obliged to pay” and “Forbidden to pay” can never be respected. The same with a contract stating “Obilged to go west” and “Obilged to go east” (as “go west” and “go east” cannot be done at the same time, i.e., are conflicting).

**Corollary 3.1.19 (conflicts)** The following statements hold:

\[ \models \neg (O_C(\alpha) \land F_C(\alpha)) \]  
(3.8)

\[ \models \neg (P(\alpha) \land F_C(\alpha)) \]  
(3.9)

\[ \text{if } \alpha \not\# C \beta \text{ then } \models \neg (O_C(\alpha) \land O_C(\beta)) \]  
(3.10)

**Proof:** The proof of (3.8) follows by propositional reasoning from (3.6) and the proof of (3.9) follows from (3.3). The proof of (3.10) follows from (3.1) and Theorem 3.1.17 as we show next.Because \( \alpha \not\# C \beta \) then \( \alpha \times \beta = 0 \), by axiom (22), and therefore \( O_C(\alpha \times \beta) \) is \( O_C(0) \). From Theorem 3.1.17 we get that \( \models \neg (O_C(\alpha) \land O_C(\beta)) \leftarrow \neg O_C(\alpha \times \beta) \) and from the above we have that \( \models \neg (O_C(\alpha) \land O_C(\beta)) \leftarrow \neg O_C(0) \). By modus ponens using (3.1) we get \( \models \neg (O_C(\alpha) \land O_C(\beta)) \).

We show now some examples of the validities presented in Proposition 3.1.20 below.

- **Prohibition of an action implies that any bigger action is prohibited:**
  \[ F_C(\alpha) \rightarrow F_C(\alpha \times \beta) \]

E.g.: “Client is forbidden to supply false information” then we also know that “Client is forbidden to supply false information and at the same time supply correct information”.

We comment more on this property. One may think of an example like “One is forbidden to smoke” but still “One is permitted to smoke and (at the same time) sit outside in the open air”. This example seems to contradict the above property. The confusion comes from the wording of the above example. A more correct wording would be: “One is forbidden to smoke and (at the same time) sit in a public place” where this action is no longer smaller than the action “smoke and (at the same time) sit outside in the open air” and thus the prohibition and the permission go along together.

- **The prohibition of a choice of two actions \( \alpha \) or \( \beta \) is the same as having both prohibition of \( \alpha \) and prohibition of \( \beta \):**
  \[ F_C(\alpha + \beta) \leftrightarrow F_C(\alpha) \land F_C(\beta) \]

E.g.: “Client is forbidden to pay in dollars or to pay in euros” implies that “Client is forbidden to pay in dollars” and that “Client is forbidden to pay in euros”.

\[^2^\text{Take concurrent actions, which specify that two or more actions are done at the same time; e.g. “drink and drive at the same time”. One concurrent action is considered bigger than another concurrent action if and only if all the actions in the smaller concurrent action are specified in the bigger action too; e.g. “drink and drive and talk to mobile at the same time” is bigger than “drink and drive at the same time”.} \]
• Permission of a choice of actions is the same as permission of all the actions in the choice; i.e., validity (3.13).

**Proposition 3.1.20** The following statements hold:

\[
\models F_C(\alpha) \rightarrow F_C(\alpha \times \beta) \quad (3.11)
\]

\[
\models F_C(\alpha + \beta) \leftrightarrow F_C(\alpha) \land F_C(\beta) \quad (3.12)
\]

\[
\models P(\alpha + \beta) \leftrightarrow P(\alpha) \land P(\beta) \quad (3.13)
\]

**Proof:** We give first quick proof arguments. The proof of the first validity is based on the fact that paths in \(A^D(\alpha \times \beta)\) contain (i.e., have bigger labels) than paths of \(A^D(\alpha)\). The proof of the second validity is based on the fact the the paths of \(A^D(\alpha + \beta)\) which satisfy the condition in the semantics are the same as the paths of \(A^D(\alpha)\) and \(A^D(\beta)\) together.

For the proof of (3.11) consider an arbitrary pointed structure \(< K^N, i >\) which satisfies \(F_C(\alpha)\). In order to show that \(K^N, i \models F_C(\alpha \times \beta)\) we need to take an arbitrary final path \(\sigma \in A^D(\alpha \times \beta)\) which satisfies \(\sigma \in S, K^N\) and show that for any edge \(t \xrightarrow{\gamma} t'\) on this path we have \(\forall s \xrightarrow{\gamma'} s' \in K^N\) with \(tS s \wedge \gamma \subseteq \gamma'\) then \(\forall a \in A_B\) if \(a \in \gamma'\) then \(\bullet_a \in \varrho(s')\). Note that if a path \(\sigma \in A^D(\alpha \times \beta)\) exists then it exists also a path \(\sigma' \in A^D(\alpha)\) which has all the labels on the edges smaller than the corresponding ones in \(\sigma\). Therefore, together with the assumption \(\sigma \in S, K^N\) it means that \(\sigma'\) also satisfies \(\sigma' \in S, K^N\). Because of this, we can apply the semantics for the expression \(F_C(\alpha)\) to deduce that for all edges \(t \xrightarrow{\gamma} t' \in \sigma'\) all transitions \(s \xrightarrow{\gamma'} s' \in K^N\) satisfying \(\gamma \subseteq \gamma'\) also satisfy \(\forall a \in A_B\) if \(a \in \gamma'\) then \(\bullet_a \in \varrho(s')\). For these edges we can find corresponding edges in \(\sigma\) that have labels \(\gamma''\) which includes \(\gamma\). Because \(\gamma \subseteq \gamma''\) it means that all the transitions \(s \xrightarrow{\gamma'} s' \in K^N\) that respect \(\gamma'' \subseteq \gamma'\) are among (possibly fewer than) the transitions before, for \(\sigma\). But all these transitions we know that respect \(\forall a \in A_B\) if \(a \in \gamma'\) then \(\bullet_a \in \varrho(s')\). The proof is finished.

It should be simple to see that the opposite implication does not always hold; i.e., \(\not\models F_C(\alpha \times \beta) \rightarrow F_C(\alpha)\). This is because we cannot guarantee that by taking all the paths \(\sigma' \in A^D(\alpha \times \beta)\) which satisfy \(\sigma' \in S, K^N\) we will consider all the paths \(\sigma \in A^D(\alpha)\), because there may be paths with labels smaller that those in \(A^D(\alpha \times \beta)\) which are still good paths for \(A^D(\alpha)\) (see Proposition 3.1.21 for a countexample).

The proof of \(\models F_C(\alpha + \beta) \leftrightarrow F_C(\alpha) \land F_C(\beta)\) is simpler. It is easy to see that the tree \(A^D(\alpha + \beta)\) contains all the final paths \(\sigma\) of the two trees \(A^D(\alpha)\) and \(A^D(\beta)\) which satisfy \(\sigma \in S, K^N\). Therefore, the double implication is immediate: if we consider \(F_C(\alpha + \beta)\) true than the traces in \(A^D(\alpha + \beta)\) respect all the conditions of the semantics and thus all the traces in \(A^D(\alpha)\) respect the conditions in the semantics, making \(F_C(\alpha)\) true (and the same for \(F_C(\beta)\)).

The proof of (3.13) is similar to the proof of (3.12).

In the design decisions for \(\mathcal{CL}\) we give special attention to what we call **unwanted implications**. This kind of “properties” are rather scarce and neglected in the literature. We give here unwanted implications that are related only to the deontic modalities (Proposition 3.1.21).

• The prohibition of doing two actions at the same time does not imply that any of the two actions is prohibited (i.e., the converse of (3.11) does not always hold).
The following statements hold:

E.g.: “One is forbidden to drink and drive at the same time” does not imply that “One is forbidden to drink” and neither that “One is forbidden to drive”.

- Obligation of an action $\alpha$ does not imply obligation of any concurrent action that contains $\alpha$. Similarly, obligation of a concurrent action does not imply obligation of any of its composing actions.

$$\not\models O_C(\alpha) \rightarrow O_C(\alpha \times \beta);$$
$$\not\models O_C(\alpha \times \beta) \rightarrow O_C(\alpha).$$

E.g.: “Obligation to drive” should not imply “Obligation to drive and drink at the same time”. For the second unwanted implication consider “Obligation to smoke and sit outside” which should not imply “Obligation to smoke”.

- Similarly with permissions:

$$\not\models P(\alpha) \rightarrow P(\alpha \times \beta)$$
$$\not\models P(\alpha \times \beta) \rightarrow P(\alpha)$$

E.g.: “Permitted to smoke and sit outside in open air” does not imply “Permitted to smoke” because if one sits inside a restaurant then one is forbidden to smoke.

- Obligation of a choice of actions constrains that only the actions in the choice can be done but the choice itself is left open, the one on which the obligation is enforced has the freedom of choosing. Therefore, none of the actions in the choice is obligatory by itself because the freedom of choosing would be lost.

$$\not\models O_C(\alpha) \rightarrow O_C(\alpha + \beta);$$
$$\not\models O_C(\alpha + \beta) \rightarrow O_C(\alpha).$$

E.g.: “Client is obliged to pay or to delay payment” should not imply that “Client is obliged to delay payment”. For the second unwanted implication “Obliged to mail the letter” should not imply “Obliged to mail the letter or burn the letter”.

- As consequence of the above we have:

$$\not\models O_C(\alpha + \beta) \rightarrow O_C(\alpha \times \beta);$$
$$\not\models O_C(\alpha \times \beta) \rightarrow O_C(\alpha + \beta).$$

**Proposition 3.1.21 (unwanted implications)** The following statements hold:
\[ \not\models O_C(\alpha) \rightarrow O_C(\alpha \times \beta) \quad (3.14) \]
\[ \not\models O_C(\alpha \times \beta) \rightarrow O_C(\alpha) \quad (3.15) \]
\[ \not\models O_C(\alpha + \beta) \rightarrow O_C(\alpha \times \beta) \quad (3.16) \]
\[ \not\models O_C(\alpha \times \beta) \rightarrow O_C(\alpha + \beta) \quad (3.17) \]
\[ \not\models O_C(\alpha) \rightarrow O_C(\alpha + \beta) \quad (3.18) \]
\[ \not\models O_C(\alpha + \beta) \rightarrow O_C(\alpha) \quad (3.19) \]

**Proof:** The proof is simple by giving for each not valid statement a counterexample, all of which are collected in Fig.3.3. The model of Fig. 3.3(i) makes \( O_C(\alpha) \) true in state \( i \) but \( O_C(\alpha \times \beta) \) does not hold (for (3.14)) and neither does \( O_C(\alpha + \beta) \) (for (3.18)). In the model of Fig. 3.3(ii) \( O_C(\alpha \times \beta) \) holds in state \( i \) but \( O_C(\alpha) \) does not hold (for (3.15)) and also \( O_C(\alpha + \beta) \) does not hold (for (3.17)). In the model of Fig. 3.3(iii) \( O_C(\alpha + \beta) \) holds in state \( i \) but \( O_C(\alpha \times \beta) \) does not hold (for (3.16)) and also \( O_C(\alpha) \) does not hold (for (3.19)). The model of Fig. 3.3(iv) makes \( F_C(\alpha \times \beta) \) true in state \( i \) but the same state does not make \( F_C(\alpha) \) true (for (3.20)). For (3.21) take the structure in Fig. 3.3(ii) which is a model for \( P(\alpha \times \beta) \) but not a model for \( P(\alpha) \). For (3.22) Fig. 3.3(i) is an obvious example, and as well for (3.23) because the model satisfies \( P(\alpha) \) but not \( P(\alpha + \beta) \). \( \square \)

### 3.1.2 Decidability for deontic modalities

We prove that the deontic modalities over synchronous actions have the **tree model property**.

**Definition 3.1.22** A pointed tree structure \( < TK^N, \varepsilon > = (W^T, R^T_{2A^B}, \gamma^T, g^T) \), is a pointed normative structure \( < K^N, s > \) satisfying the restrictions of Definition 2.2.30 and:

a) the nodes are characterized by strings over natural numbers \( W^T \subset \mathbb{N}^* \), with \( s = \varepsilon \);

b) for each label \( \alpha \in 2^{A^B} \) the partial function \( R^T_{2A^B}(\alpha) : W^T \rightarrow W^T \) respects the restriction: \( R^T_{2A^B}(\alpha)(x) = xi \) where \( x, xi \in W^T \) and \( i \in \mathbb{N} \);

c) for any \( \alpha \neq \beta \in 2^{A^B} \) then \( R^T_{2A^B}(\alpha)(x) \neq R^T_{2A^B}(\beta)(x) \) for any \( x \in W^T \).

Condition a) above labels each state with a (unique) string over natural numbers. Condition b) guarantees that the structure contains no cycles, and c) that 2 transitions starting in the same state and with different labels (actions) go to different states.

The following lemma shows that it is always possible to obtain a tree structure from a given pointed normative structure (the standard unwinding construction).

**Lemma 3.1.23 (tree model)** Given a pointed normative structure \( < K^N, i > \) we can construct an associated tree structure \( < TK^N, \varepsilon > \).

**Proof:** The technique that we use is known in modal logics as the tree unfolding of a Kripke structure [Sah75, HKT00]. For a pointed normative structure \( K^N, i = (W, R_{2A^B}, \nu, g) \) we can view the set of worlds \( W = \{0, 1, 2, \ldots \} \) to be the natural numbers \( \mathbb{N} \); and we define the set \( W^T[i] \subset \mathbb{N}^* \) to be the set of finite paths starting from \( i \). Moreover, we enrich the paths to contain also the labels by which the path was formed. For this we interlace between the nodes
labels from $2^{A_B}$. More precisely, $i$ is considered the empty set $\varepsilon$, the paths of depth one are $\varepsilon\alpha s$ such that $s \in W$ and $i \xrightarrow{\alpha} s$ is a transition in $K^N$. We define a function $\rho: W^T[i] \rightarrow W$ which assigns to each path the state in which the path ends; e.g. $\rho(\varepsilon\alpha s'\beta s'') = s''$. Note that two paths $x\alpha s$ and $x\beta s$ are regarded as different. Consider the set $W[i] = \{\rho(x) \mid x \in W^T[i]\}$ of states reachable (by any path) from the node $i$. The function $\rho: W^T[i] \rightarrow W[i]$ is a surjection therefore it exists the corresponding function $\rho^{-1}$ which returns sets of traces from $W^T[i]$.

For $< K^N, i >$ we construct the pointed structure $TK^N, \varepsilon = (W^T[i], R^T_{2A_B}, V^T, \varrho^T)$ as follows. The function $R^T_{2A_B}$ assigns a partial function $R^T_{2A_B}(\alpha): W^T[i] \rightarrow W^T[i]$ to each $\alpha$ (we write the partial functions as sets of pairs of argument/value) which is defined as:

$$R^T_{2A_B}(\alpha) = \{(x, x\alpha s) \mid (\rho(x), s) \in R_{2A_B}(\alpha)\}.$$ 

The valuation function $V^T$ is defined in terms of $\varrho$:

$$V^T(\phi) = \rho^{-1}(V(\phi)).$$

We used the standard pointwise extension of the function $\rho^{-1}$ over a set of elements as argument. The marking function $\varrho^T$ is defined in terms of $\varrho$:

$$\varrho^T(x) = \varrho(\rho(x)).$$

It is easy to see that $TK^N, \varepsilon$ is a tree structure with root node $\varepsilon$. We can check that $TK^N, \varepsilon$ is the normative structure. The restrictions imposed by Definition 3.1.22 on the function $R^T_{2A_B}$ are met. Precisely, for any of the partial functions $R^T_{2A_B}(\alpha)$ it cannot be the case that $R^T_{2A_B}(\alpha)(x) = y\alpha s$ where $x \neq y$ (i.e., the first restriction is met). Take now two different actions $\alpha \neq \beta$ then it cannot be the case that $R^T_{2A_B}(\alpha)(x) = R^T_{2A_B}(\beta)(x)$ because $R^T_{2A_B}(\alpha)(x) = x\alpha s \neq x\beta s' = R^T_{2A_B}(\beta)(x)$ even if $s = s'$.

The following lemma characterizes naturalness in terms of tree structures.

**Lemma 3.1.24 (natural obligation)** The naturalness constraint of Definition 3.1.7 is equivalent to the following:

$$\exists \gamma \text{ s.t. } A^D(\alpha \times \gamma) \equiv TK^A_{\max}.$$  \hfill (3.24)

**Proof:** We prove the double implication (3.7) iff (3.24); or more precisely,

$$A^D(\alpha \times \gamma) S^s K^A_{\max} i \wedge K^A_{\max} i S^s A^D(\alpha \times \gamma) \iff A^D(\alpha \times \gamma) \equiv TK^A_{\max}. $$

The right to left direction is immediate. Thus, we concentrate on the left to right direction and consider two cases, in a recursive argument:

1. For any edge $r \xrightarrow{\beta} t \in A^D(\alpha \times \gamma)$ we want to find an edge $\varepsilon \xrightarrow{\beta} \varepsilon\beta x \in TK^A_{\max}(\alpha, \varepsilon)$. We have, from the first strong simulation, that $r \xrightarrow{\beta} t$ it exists $\beta \xrightarrow{\beta} s \in K^A_{\max}(\alpha, \beta)$. From Lemma 3.1.23 it is clear that it exists $\varepsilon \xrightarrow{\beta} \varepsilon\beta s \in TK^A_{\max}(\alpha, \varepsilon)$ because $\rho(\varepsilon) = i$. Moreover, from the first strong simulation we know that $tS^s s$ and thus we can use the same algorithm as before to go down the tree $A^D(\alpha \times \gamma)$ working with $t$ and $s$, instead of $r$ and $i$. In this way we prove that all the tree $A^D(\alpha \times \gamma)$ is in the tree $TK^A_{\max}(\alpha, \varepsilon)$. 
Lemma 3.1.23 it means that \( \rho_{TK} \).

The proof of (3.26) follows from (3.25) by replacing \( \rho \) by a similar recursive argument, means that \( \epsilon s \) which means we can use the same argument to go down the tree to the node \( \epsilon s \) to find edge \( t \rightarrow t' \in A^D(\alpha \times \gamma) \) for any edge \( \epsilon s \rightarrow \epsilon s \beta s' \in TK_{\max}^D(\alpha, \epsilon) \).

Corollary 3.1.25 (naturalness in terms of actions) The naturalness constraint reduces to showing that it exists a deontic action \( \gamma \) s.t. \( \alpha \times \gamma = \alpha^T \), where \( \alpha^T \) is the action in canonical form corresponding to the tree \( TK_{\max}^A(\alpha, \epsilon) \).

Proof: This is a consequence of Lemma 3.1.24 and of the completeness result of Theorem 2.2.28 which says that for any tree as in Theorem 2.2.31 there is a corresponding action in canonical form.

Theorem 3.1.26 (unfolding) For a pointed structure \( < K^N, i > \) we have:

\[
TK^N, x \models C \iff K^N, \rho(x) \models C \quad (3.25) \quad TK^N, \epsilon \models C \iff K^N, i \models C \quad (3.26)
\]

Proof: The proof of (3.26) follows from (3.25) by replacing \( x \) with \( \epsilon \) (and thus \( \rho(\epsilon) = i \)). The proof of (3.25) is done by induction on the structure of the formula \( C \). It has lengthy but easy cases as it needs to prove each of the conditions in the definitions of the semantics of the deontic modalities.

Basis: The case for when \( C = \bot \) is trivial. The second base case is when \( C = \phi \). Then \( TK^N, x \models \phi \iff x \in V^T(\phi) \) which means that \( \rho(x) \in \rho(V^T(\phi)) \). By the definition of \( V^T \) from Lemma 3.1.23 it means that \( \rho(x) \in \rho(\rho^{-1}(V(\phi))) \) which is \( \rho(x) \in V(\phi) \). This is equivalent to \( K^N, \rho(x) \models \phi \) and the proof is finished.

Case for \( C = P(\alpha) \). We prove that \( TK^N, x \models P(\alpha) \iff K^N, \rho(x) \models P(\alpha) \).

First we prove that \( A^D(\alpha) S_x TK^N \iff A^D(\alpha) S_{\rho(x)} K^N \), which is equivalent to proving \( r S x \iff r S \rho(x) \), where \( r \) is the root of \( A^D(\alpha) \). Because we use this result in several places we refer to it as to the simulation result. The Definition 3.1.2 says that from \( r S x \) we have that \( \forall t \rightarrow t \in A^D(\alpha) \) then \( \exists x \rightarrow x \gamma s \in TK^N \) s.t. \( \alpha_s \leq x \gamma \) and \( t S x \gamma s \) (where, all through this proof, we consider \( s \in N \)). More precisely, \( x \rightarrow x \gamma s \in TK^N \) means that \( (x, x \gamma s) \in R^T_{\alpha_s}(\gamma) \), which, by the definition of \( R^T_{\alpha_s} \) from Lemma 3.1.23, implies that \( (\rho(x), s) \in R_{\alpha_s}^T(\gamma) \). Thus we have that \( \forall t \rightarrow t \in A^D(\alpha) \) then \( \exists \rho(x) \rightarrow s \in K^N \) s.t. \( \alpha_s \leq x \gamma \). By applying a recursive reasoning with \( t S x \gamma s \) we also get that \( t S s \in K^N \). To finish the simulation result we prove the second condition from the definition of the simulation relation. We use reductio ad absurdum and assume \( \exists \rho(x) \rightarrow s \in K^N \) with \( \alpha_s \leq x \gamma \) for which \( t S s \) is not the case. From Lemma 3.1.23 we know that \( \exists x \rightarrow x \gamma s \in TK^N \), and from \( A^D(\alpha) S_x TK^N \) we also know that \( \forall x \rightarrow x \gamma s \in TK^N \) with \( \alpha_s \leq x \gamma \) we have \( t S x \gamma s \) which, by a similar recursive argument, means that \( t S s \), hence the contradiction. The proof for the right to left direction is analogue, using Lemma 3.1.23.

We know continue to prove the second condition from the definition of the semantics of \( P \); i.e.
∀r \xrightarrow{γ} t \in A^P(α), \forall x \xrightarrow{γ'} xγ's ∈ TK^N \text{ s.t. } rSx \land γ \subseteq γ' \\
then ∀a \in A_B if a ∈ γ then \bullet_a \not\in \mathcal{g}^T(xγ's) \\
⇔ \\
∀r \xrightarrow{γ} t \in A^P(α), \forall ρ(x) \xrightarrow{γ'} s \in K^N \text{ s.t. } rSρ(x) \land γ \subseteq γ' \\
then ∀a \in A_B if a ∈ γ then \bullet_a \not\in \mathcal{g}(s).

Consider the implication “⇒”. For an arbitrary ρ(x) \xrightarrow{γ'} s ∈ K^N we know from Lemma 3.1.23 that \exists x \xrightarrow{γ'} xγ's ∈ TK^N and from the hypothesis we know that ∀a ∈ A_B if a ∈ γ then \bullet_a \not\in \mathcal{g}^T(xγ's). By the definition of \mathcal{g}^T, from Lemma 3.1.23, we know that \mathcal{g}^T(xγ's) = \mathcal{g}(\rho(xγ's)) = \mathcal{g}(s), and thus, we have our conclusion ∀a ∈ A_B if a ∈ γ then \bullet_a \not\in \mathcal{g}(s).

Consider now “⇐”. For an arbitrary x \xrightarrow{γ'} xγ's ∈ TK^N we know that we have \exists ρ(x) \xrightarrow{γ'} s ∈ K^N with ∀a ∈ A_B if a ∈ γ then \bullet_a \not\in \mathcal{g}(s). Consider now, by reductio ad absurdum, that \exists a ∈ A_B with a ∈ γ and \bullet_a \in \mathcal{g}^T(xγ's). By the definition of \mathcal{g}^T we have that \bullet_a \in \mathcal{g}(s) which is a contradiction with the hypothesis. Thus the case is finished.

**Inductive step:**

**Case for C = O_C(α).** We prove that TK^N, x \models O_C(α) iff K^N, ρ(x) \models O_C(α) under the inductive hypothesis ∀x \in TK^N then TK^N, x \models C ⇔ K^N, ρ(x) \models C.

We have proven that A^P(α) \mathcal{S}_x TK^N iff A^P(α) \mathcal{S}_{ρ(x)} K^N in the simulation result. We now prove the second requirement from the definition of the semantics of obligations, i.e., we prove the double implication:

∀r \xrightarrow{γ} t \in A^P(α), \forall x \xrightarrow{γ'} xγ's \in TK^N \text{ s.t. } rSx \land γ \subseteq γ' \\
then ∀a \in A_B if a ∈ γ then \circ_a \in \mathcal{g}^T(xγ's) \\
⇔ \\
∀r \xrightarrow{γ} t \in A^P(α), \forall ρ(x) \xrightarrow{γ'} s \in K^N \text{ s.t. } rSx \land γ \subseteq γ' \\
then ∀a \in A_B if a ∈ γ then \circ_a \in \mathcal{g}(s)

The proof is similar to what we did for permissions. We consider only the “⇒” implication. For an arbitrary ρ(x) \xrightarrow{γ'} s ∈ K^N we know that we have x \xrightarrow{γ'} xγ's ∈ TK^N and from the hypothesis we know that ∀a ∈ A_B if a ∈ γ then \circ_a \in \mathcal{g}^T(xγ's). By the definition of \mathcal{g}^T we have that ∀a ∈ A_B if a ∈ γ then \circ_a \in \mathcal{g}(s).

To prove the third condition from the definition of the semantics of O_C(α) we prove the following double implication:

∀x \xrightarrow{γ} xγs \in TK^N then ∀a \in A_B if a ∈ γ then \circ_a \not\in \mathcal{g}^T(xγs) \\
⇔ \\
∀ρ(x) \xrightarrow{γ'} s \in K^N, ρ(x) then ∀a \in A_B if a ∈ γ then \circ_a \not\in \mathcal{g}(s)

We do the proof of “⇒” using the reductio ad absurdum principle (the proof of “⇐” is analogous). Suppose that \exists ρ(x) \xrightarrow{γ'} s ∈ K^N, ρ(x) and \exists a ∈ A_B with a ∈ γ s.t. \circ_a \in \mathcal{g}(s). This implies that there also exists the transition x \xrightarrow{γ} xγs ∈ TK^N for which, from the definition of \mathcal{g}^T from Lemma 3.1.23, it also holds that \circ_a \in \mathcal{g}^T(xγs) = \mathcal{g}(s). This requires two subcases:

a. \ x \xrightarrow{γ} xγs \in TK^N which, from the precondition of the implication, means that \circ_a \not\in \mathcal{g}^T(xγs), resulting in a contradiction;
b. $x \xrightarrow{\gamma} x\gamma s \not\in TK_{ren}^{AP}(a,x)$. From the assumption $\exists \rho(x) \xrightarrow{\gamma} s \in K_{ren}^{AP}(a,\rho(x))$, by way of Definition 3.1.4, it means that $\rho(x) \in K_{max}^{AP}(a,\rho(x))$ and $\exists \rho(x) \xrightarrow{\alpha_x}s' \in K_{max}^{AP}(a,\rho(x))$. This implies that $x \in TK_{max}^{AP}(a,x)$ and $\exists x \xrightarrow{\alpha_x}x\alpha x s' \in TK_{max}^{AP}(a,x)$, which, together with the assumption of this case and by way of Definition 3.1.4, means that $x \xrightarrow{\gamma} x\gamma s \in TK_{max}^{AP}(a,x)$. By the Definition 3.1.3 of the maximal simulating structure it implies that $\exists r \xrightarrow{\alpha'_x} t \in A^D(\alpha)$ s.t. $r s x \land (\alpha'_{\times} <_x \gamma) \land t s x\gamma s$. By the reasoning we did at the beginning for the $A^D(\alpha)$ $S_x TK^N$ we conclude that $t S \rho(x) \land (\alpha_{\times} <_x \gamma) \land t S s$ which means that $\rho(x) \xrightarrow{\gamma} s \in TK_{max}^{AP}(a,\rho(x))$. Thus we have a contradiction.

We need to prove the last condition from the definition of the semantics of $O_C(\alpha)$; i.e., we prove the double implication:

$$TK^N, x \models C \quad \text{\forall } x \in TK^N \text{ with } t S^s x \land t \in leafs(A^D(\overline{a}))$$

$$\iff$$

$$K^N, s \models C \quad \text{\forall } s \in K^N \text{ with } t S^s s \land t \in leafs(A^D(\overline{a}))$$

Here we use the induction hypothesis. We prove only the forward implication by *reductio ad absurdum* and assume that $\exists s \in K^N$ s.t. $s$ is reached by following exactly (because of the strong simulation condition $S^*$) one final path in the tree of the complemented action $A^D(\overline{a})$. For this state we assume $K^N, s \not\models C$. We have, thus, the sequence of transitions in $K^N$: $r \xrightarrow{\alpha_1^1} 1, 1 \xrightarrow{\alpha_2^2} 2, \ldots, n - 1 \xrightarrow{\alpha_n^n} n$ (recall that we consider the states of $K^N$ to be labeled with natural numbers) where $n = s$. For each of these transitions there is a transition in $TK^N$: $\exists \epsilon \xrightarrow{\alpha_1^1} \epsilon \alpha_1^1, 1, \exists \epsilon \alpha_1^1 \xrightarrow{\alpha_2^2} \epsilon \alpha_1^1 \alpha_2^2, \ldots, \exists \epsilon \alpha_1^1 \ldots n - 1 \xrightarrow{\alpha_n^n} \epsilon \alpha_1^1 \ldots \alpha_n^n n$. From this and the left part of the implication we have that $TK^N, \epsilon \alpha_1^1 \ldots \alpha_n^n n \models C$. By the inductive hypothesis it means that $K^N, \rho(\epsilon \alpha_1^1 \ldots \alpha_n^n n) \models C$ which is $K^N, n \models C$ (or $K^N, s \models C$). Hence, the contradiction and the end of the proof.

For normal obligations we need to treat the normality condition too; this means proving the following double implication:

$$\exists \gamma \text{ s.t. } A^D(\alpha \times \gamma) \models TK_{max}^{AP}(a,x)$$

$$\iff$$

$$\exists \gamma' \text{ s.t. } A^D(\alpha \times \gamma') \models TK_{max}^{AP}(a,\rho(x))$$

We actually prove that $TK_{max}^{AP}(a,x) \models TK_{max}^{AP}(a,\rho(x))$ which implies that $\gamma = \gamma'$ solves the double implication. Note first that $TK_{max}^{AP}(a,\rho(x))$ is the tree unfolding of the $K_{max}^{AP}(a,\rho(x))$ maximal simulating structure of $K^N$ w.r.t. the state $\rho(x)$, whereas, $TK_{max}^{AP}(a,x)$ is the maximal simulating structure coming from the tree unfolding of $K^N$ w.r.t. the state $\rho(x)$. We use a similar proof argument as done for Lemma 3.1.24; we use a recursive reasoning working on levels of the two trees, beginning at the first level of edges, those starting in the roots of the two trees.

Pick some arbitrary edge $\epsilon \xrightarrow{\gamma} \epsilon \gamma s \in TK_{max}^{AP}(a,\rho(x))$ for which we want to find a corresponding edge in $TK_{ren}^{AP}(a,x)$. This means that it exists $\rho(x) \xrightarrow{\gamma} s$ a transition in $K_{ren}^{AP}(a,\rho(x))$. Because $x$ is the root of $TK_{max}^{AP}(a,x)$ we can find the edge $x \xrightarrow{\gamma} x\gamma s \in TK_{max}^{AP}(a,x)$, which is the edge we were looking for.

---

3Note that there should be two $\rho$ functions, one coming from the unfolding of $K^N$ (which is the one in the double implication) and another $\rho'$ function (which is not visible) coming from the unfolding of the $K_{max}^{AP}(a,\rho(x))$. Actually here $\rho'(\epsilon) = \rho(x)$. 
For the forward direction pick some arbitrary edge \( x \xrightarrow{\gamma} x\gamma s \in TK^{A^D(\alpha),x}_{\text{max}} \). This means that we have an edge \( \rho(x) \xrightarrow{\gamma} s \in K^{A^D(\alpha),\rho(x)}_{\text{max}} \), which means that we have the desired edge \( \varepsilon \xrightarrow{\gamma} \varepsilon\gamma s \in TK^{A^D(\alpha),\rho(x)}_{\text{max}} \), as \( \rho(x) = \rho'(\varepsilon) \).

The case for \( C = F_C(\alpha) \) follows similar reasoning as for \( O_C \) only that care must be taken when dealing with the partial simulation relation \( \tilde{S} \).

The case for the propositional implication uses simple structural induction.

**Corollary 3.1.27** (tree model property)

If \( C \) has a model \( K^N \) then it has a tree model \( TK^N \).

**Proof:** This follows immediately from equation (3.26) of the Theorem 3.1.26 which says that if a formula \( C \) is true in a state \( i \) of a model \( K^N \) then there exists a tree model \( TK^N \), as in Lemma 3.1.23, in which the formula is true at state \( \varepsilon \).

Next we prove that the deontic modalities alone have the finite model property. Note first that it is rather hard to use the filtration technique in our case. In PDL the Fischer-Ladner closure was needed so to determine the subformulas of a dynamic modality with a complex action inside (e.g. \( [a \cdot (b + c)]\varphi \)). In our case we do not know what are subformulas of an obligation of a complex action like \( O_C(a \cdot (b + c)) \). We use, instead, the selection technique for proving the finite model property [BdRV01, sec.2.3].

Before proving the finite model property (Theorem 3.1.33) we give some necessary definitions and prove auxiliary results.

**Definition 3.1.28** (action length) The length of an action \( \alpha \) is defined (inductively) as a function \( l : A^D \rightarrow \mathbb{N} \) from actions to natural numbers.

- \( l(1) = l(0) = 0 \),
- \( l(a) = 1 \), for any basic action \( a \) of \( A_B \),
- \( l(\alpha \times \beta) = l(\alpha + \beta) = \max(l(\alpha), l(\beta)) \),
- \( l(\alpha \cdot \beta) = l(\alpha) + l(\beta) \).

The length function counts the number of actions in a sequence of actions given by the \( \cdot \) constructor. We say that \( \alpha(n) \) identifies the action on position \( 0 < n \leq l(\alpha) \) in the action \( \alpha \). For \( n = 0 \), \( \alpha(0) = 1 \) returns the implicit \textit{skip} action, which is natural because every action \( \alpha \) can have as starting action \( 1 \), i.e., \( \alpha = 1 \cdot \alpha \). For example, for action \( \alpha = (a + b) \cdot 1 \cdot c \) we have \( l(\alpha) = 2 \), \( \alpha(1) = a + b \) and \( \alpha(2) = c \). Note that \( \alpha(\cdot) \) ignores 1’s.

The following proposition states that complementing a (complex) action does not increase its length.

**Proposition 3.1.29** For any action \( \alpha \) we have \( l(\overline{\alpha}) \leq l(\alpha) \).
Proof: A careful inspection of the Definition 2.2.17 of action complement easily shows that $\overline{\alpha}$ does not add $\cdot$ combinators at bigger depth than those in $\alpha$. The complement operation is applied recursively and at each recursive step it generates paths of length 1 for paths of length 1 or greater in the original action $\alpha$; or it generates paths of length $1 + \text{length generated in the next recursive step}$. This happens for each $\cdot$ found in $\alpha$. Therefore, $\overline{\alpha}$ cannot have paths of greater depth than the paths in $\alpha$. ∎

We now relate the height of the tree of an action with its length.

**Corollary 3.1.30** For any action $\alpha$ we have $h(A^D(\overline{\alpha})) \leq h(A^D(\alpha)) = l(\alpha)$.

Proof: This is a corollary of both Theorem 2.2.31 and Proposition 3.1.29. ∎

**Definition 3.1.31 (depth of formula)** We define the depth of a formula inductively as a function $d$ from formulas to natural numbers:

- $d(\phi) = d(\bot) = 0$;
- $d(C_1 \rightarrow C_2) = \max(d(C_1), d(C_2))$;
- $d(P(\alpha)) = l(\alpha)$;
- $d(O_C(\alpha)) = d(F_C(\alpha)) = l(\alpha) + d(C)$.

**Lemma 3.1.32 (bounded depth tree model)** Take a formula $C$ with depth $k$. If $TK^N, \varepsilon \models C$ then $C$ holds in the root of the tree structure $TK^N, \varepsilon$ restricted to paths of maximum depth $k$ (i.e., where all nodes of depth $> k$ are removed).

Proof: We use induction on the structure of the formula $C$.

**Base case:** The proof for formulas $\bot$ and $\phi$ which have depth 0 is simple as we need to inspect only the root node $\varepsilon$ therefore we need only nodes of depth 0 in the tree structure.

For the formula $P(\alpha)$ which has depth $l(\alpha)$ we need to inspect only those nodes of $TK^N, \varepsilon$ that respect the simulation relation. Therefore, the maximum depth of a node is the maximum length in the final paths of $A^D(\alpha)$, and thus the maximum depth of the nodes in $TK^N, \varepsilon$ is $h(A^D(\alpha))$ which, by Corollary 3.1.30, is $l(\alpha)$.

**Inductive step:** When $C$ is of the form $C_1 \rightarrow C_2$ the depth of the formula is the maximum of the depths of the two subformulas. The semantics says that we need to check first if $C_1$ holds, which by the inductive hypothesis it means that we need a subtree of depth at most $d(C_1)$. If $C_1$ holds we need to check also $C_2$ which, by the inductive hypothesis, requires also a subtree of depth at most $d(C_2)$. Overall, we need to check a subtree of $TK^N, \varepsilon$ with depth at most $\max(d(C_1), d(C_2))$.

The proof for the formulas $O_C(\alpha)$ and $F_C(\alpha)$ is similar and we treat here only the proof for obligations. The semantics of $O_C(\alpha)$ says that we need to check first the obligation alone which requires nodes of depth at most $h(A^D(\alpha)) = l(\alpha)$ because of the simulation relation. Secondly, we need to check that the reparation $C$ holds at the states corresponding to the leaf nodes of the
Table 3.3: Syntax of the contract language $\mathcal{CL}$.

Based on the above auxiliary results we can prove now the finite model property.

**Theorem 3.1.33 (finite model property)** If a formula has a model then it has a finite model.

**Proof**: We can work equivalently in pointed structures and then we need to prove that if a formula is satisfied in a pointed structure then it is satisfied in a finite pointed structure. Take a formula $C$ of depth $k$ which is satisfiable in the pointed structure $K^N, i$. By Corollary 3.1.27 we know that $C$ is satisfied in the tree-like pointed structure $TK^N, \varepsilon$. Note that the tree might have both infinite depth and infinite branching.

By Lemma 3.1.32 we put a bound on the depth of the tree which is related to the formula. Because we work with deterministic structures and because the set of labels $2^{\mathcal{A}}$ is finite we have a guaranteed finite branching. Therefore the model is finite.

As a corollary of the above theorem we have a (relative) decidability result.

**Corollary 3.1.34 (decidability)**

a. The logic with general obligations as in the semantic Definition 3.1.5 is decidable.

b. The logic with natural obligations is decidable iff the naturalness constraint is decidable.

As a side remark, we have proven the tree model property in Theorem 3.1.26 for general obligations as well as for natural obligations. Therefore, in both cases it is enough to check finite trees for satisfiability, but the difference is that when checking for natural obligations we need to test that the naturalness constraint is satisfied. This test is still an open problem (see end of Section 3.3); we do not know if it is decidable whether an action $\gamma$ can be found satisfying the naturalness constraint.

### 3.2 The Full Contract Logic

The contract logic $\mathcal{CL}$ adds to the deontic modalities from Section 3.1 the dynamic logic modality applied over synchronous actions. The syntax of $\mathcal{CL}$ is given in Table 3.3. The dynamic logic modality $[\cdot]\varphi$ is parameterized by actions $\delta$. The expression $[\delta]\varphi$ is read as: “after the action $\delta$ is performed $C$ must hold”.

\(K^N, i \models \delta | C\) iff \(\forall s \in K^N\) with \((i, s) \in R_{2AB}(\delta)\) then \(K^N, s \models C\).

\(R_{2AB}(\delta) = \{(s, s') | \exists k, \exists \sigma = \sigma_0 \ldots \sigma_k\) a final path in \(A^\delta(\delta)\), \(\exists s_0 \ldots s_k \in K^N\) with \(s_0 = s\) and \(s_k = s'\), and \(\forall 0 \leq i \leq k, \forall (s_i) \in \mathcal{L}([\sigma_i])\), and \(\forall 0 \leq i < k\) with \(\sigma_i \overset{\alpha_i}{\rightarrow} \sigma_{i+1} \in \sigma\) then \((s_i, s_{i+1}) \in R_{2AB}(\alpha_i')\}\)

Table 3.4: Semantics for \(\mathcal{C}L\).

In \(\mathcal{C}L\) we can write conditional obligations, permissions and prohibitions of two different kinds. As an example let us consider conditional obligations. The first kind is represented by \([\delta]O(\alpha)\), which may be read as “after performing \(\delta\), one is obliged to do \(\alpha\)”. The second kind is modeled using the implication operator: \(\mathcal{C} \rightarrow O(\alpha)\), which is read as “If \(\mathcal{C}\) holds then one is obliged to perform \(\alpha\)”.

Propositional dynamic logic (PDL) makes an interplay between the actions and the formulas; i.e., it has formulas as actions (tests) and it has actions defining the formulas (the box modality). The intuition of the test action is that \(\varphi?\) can be performed only if the formula \(\varphi\) holds in the current world. The same, a sequence action \(\varphi?\cdot\alpha\) can be viewed as a guarded action because \(\alpha\) can be performed only if the test \(\varphi?\) succeeds. Note that we use, what is called, poor tests because we do not allow for a modal formula to be a test, but only Boolean tests; i.e., we cannot ask modal questions using the dynamic or the deontic modalities.

There are two differences between the actions \(\delta\) which appear inside the PDL modality \([\cdot]\) and the actions \(\alpha\) which are allowed inside the deontic modalities. We argued before against not having the Kleene \(*\) for the \(\alpha\) actions. Regarding the tests, if we allow deontic test actions, like \(F(\alpha)\) inside the deontic modalities it would break the ought-to-do approach because they introduce the formulas inside the action formalism; i.e., we could write formulas like \(O_C(F(\alpha)?)\). This constitutes a combination of ought-to-do and ought-to-be (for this direction check [dMW96]). Moreover, adding tests inside the deontic modalities does not integrate with our way of giving semantics; we do not know how to mark obligatory (or prohibited) tests, as we do with the actions. Moreover, in case of violation, the reparation \(\mathcal{C}\) is enforced in the same world as the \(O\) and the \(F\), i.e., there is no state change. Therefore, we could reason only with the propositional logic part of \(\mathcal{C}L\). The example \(O_C(F(\alpha)?)\) is read as “It is obligatory (in the current world) that the test \(F(\alpha)\) holds (in the current world), otherwise (if the test does not hold) the reparation \(\mathcal{C}\) should be enforced afterwards”. Compared to deontic actions, tests do not change the world: if a tests succeeds then we remain in the same world and execute the next action, if the test fails then the whole action sequence fails. We can achieve the same by only using the \(\mathcal{C}L\) language as it is. The example above is specified in \(\mathcal{C}L\) as \(F(\alpha) \lor (\neg F(\alpha) \land \mathcal{C})\) which is read as above (we reword it here to match the formula better): “(In the current world) it is forbidden to do \(\alpha\) or it is not forbidden to do \(\alpha\) and the formula \(\mathcal{C}\) holds”. Note the difference of our formula and the formula \(F(\alpha) \lor \mathcal{C}\) which does not capture what we want.

**Definition 3.2.1 (semantics)** The semantics for the dynamic modality of \(\mathcal{C}L\) is given in Table 3.4. The rest of the syntactic constructs of \(\mathcal{C}L\) (i.e., the deontic and propositional operators) have the semantics from Table 3.2.

The expression \([\delta]C\) is evaluated in a state \(i\) of the normative structure \(K^N\), depending
on the automata representation of the dynamic action $\delta$. Essentially, the semantics needs to evaluate the expression $C$ to true in all states $s$ reachable from the initial state $i$ by following the automaton $A^G(\delta)$ of the dynamic action $\delta$. All the states $s$ reached from $i$ are given by the relation described by the dynamic action, i.e., $(i, s) \in R_{2A_B}(\delta)$. The relation $R_{2A_B}(\delta)$ is not as simple to describe as was the case with the deontic actions where we needed to look only at single steps. In the case of dynamic actions we need to look several steps in the structure. We take the approach introduced in [HS83] and use the automata $A^G(\delta)$, as in Definition 2.3.10, interpreting the dynamic actions $\delta$.

The relation $R_{2A_B}(\delta)$ is defined as the set of all pairs of states $(s, s')$ having the property that there is an accepting path $\sigma$ in $A^G(\delta)$ that is matched by a sequence of states in $\mathcal{K}^N$. A sequence of states matches the path $\sigma$ iff all the edges $x_i \xrightarrow{\alpha_s} x_{i+1} \in \sigma$ are matched by the corresponding transitions $(s_i, s_{i+1}) \in R_{2A_B}(\alpha_s)$ (i.e., the indexes have to match as well as the labels $\alpha_s$) and the valuation for the states has to conform with the sets of atoms of the corresponding nodes on the path (i.e., $\mathcal{V}(s_i) \in \mathcal{L}(\alpha_i x_i)$). From this matching sequence of states take the first and the final state as the pair we are looking for. The conformance test $\mathcal{V}(s_i) \in \mathcal{L}(\alpha_i x_i)$ is required to ensure than any test action from $\delta$ is satisfied at the particular state; i.e., if the valuation of the state corresponds to one of the atoms (atoms are encodings of valuations) that are encoded by the automaton $\alpha_i x_i$ of the node $x_i$.

As an example, consider the normative structure of Fig. 3.1 on page 59 and the automaton $A^G(\delta)$ from Fig. 2.10 on page 46 corresponding to the dynamic action $\delta = (p \cdot b)^* + (d \times n) \cdot p \cdot \phi \cdot \cdot p$. We want to check if the formula $[\delta]\phi$ holds in state $s_1$. The automaton $A^G(\delta)$ has the following final paths: $\{(r, t_1, t_2, t_3), (r, t_6), (r, t_4, t_5), (r, t_4, t_5, (t_4, t_5)^*), (r, t_4, t_5, (t_4, t_5, t_4)^*)\}$. Because all our normative structures are reflexive (recall the property (3.2) in Proposition 3.1.6), we calculate $R_{2A_B}(\delta) = \{(s_1, s_1)\}$. At a closer look, the path $(r, t_1, t_2, t_3)$ does not contribute to the $R_{2A_B}(\delta)$ because $\mathcal{V}(s_2) \not\in \mathcal{L}(\alpha_2 x_2)$ (i.e., the automaton $\alpha_2 x_2$ accepts only atoms that make $\phi$ true, but $\mathcal{V}(s_2)$ makes $\phi$ false). The path $(r, t_6)$ is matched by the reflexive loop $(s_1, s_1)$, whereas the rest of the paths $(r, t_4, t_5, (t_4, t_5)^*)$ are matched by sequences of states $s_1, s_2, s_3$ and $s_1, s_2, s_1, (s_4, s_1)^*$. Therefore, $s_1 \models [\delta]\phi$ because $s_1 \models \phi$. Consider now a slight modification $\delta' = (p \cdot b)^* + (d \times n) \cdot p \cdot \phi \cdot \cdot p$ (i.e., the test $\phi$ does not appear). In this case the language $\mathcal{L}(\alpha_2 x_2)$ is the universal language (i.e., all the atoms) and hence, the path $(r, t_1, t_2, t_3)$ contributes to $R_{2A_B}(\delta)$ by adding the pair $(s_1, s_4)$. Because of this $s_1 \not\models [\delta']\phi$ since one of the pairs in $R_{2A_B}(\delta)$ does not respect the condition of the semantics; i.e., $s_4 \not\models \phi$. Consider a subsequent modification $\delta'' = (p \cdot b)^* + (d \times n) \cdot p$ (i.e., the last $p$ action is removed). The automaton $A^G(\delta)$ from Fig. 2.10 on page 46 has the node $t_3$ removed, and thus the final path $(r, t_1, t_2)$ contributed to $R_{2A_B}(\delta)$ with the pair $(s_1, s_3)$. In this case $s_1 \models [\delta'']\phi$ because in both $s_1$ and $s_3$ the formula $\phi$ holds. Consider a last modification of $\delta''$ to $\delta''' = (p \cdot \phi \cdot \cdot b)^* + (d \times n) \cdot p$. In this case all the paths $(r, t_4, t_5)$ cannot contribute to $R_{2A_B}(\delta)$ any more because the valuation $\mathcal{V}(s_4)$ does not make $\phi$ true and hence is not part of $\mathcal{L}(\alpha_4 x_4)$ which contains only those atoms that make $\phi$ true; nevertheless $s_1 \models [\delta''']\phi$.

### 3.2.1 Properties of the $\mathcal{CL}$ logic

The validities and non-validities results for the deontic modalities of Section 3.1.1 hold for $\mathcal{CL}$ also. Adding the dynamic modality does not affect these. Besides, we have extra properties that
deal with the combination of deontic and dynamic modalities.

Denote by any = +\(\alpha_x\in\mathcal{A}_B\) the choice between all the \(\times\)-actions. For all \(\alpha_x\in\mathcal{A}_B^\times\) denote by \(\langle\langle\alpha_x\rangle\rangle\) the formula \(\langle\langle\alpha_x\times\text{any}\rangle\rangle\) and by \([\alpha_x]\) the formula \([\alpha_x\times\text{any}]\). Note that \(\langle\langle\cdot\rangle\rangle\) and \([\cdot]\) are duals in this definition. Extend \([\cdot]\) to all actions \(\alpha\in\mathcal{A}\), as is done in the standard PDL (the definition for \(\langle\langle\cdot\rangle\rangle\) is analogous):

\[
\begin{align*}
[[\alpha + \beta]]& = [[\alpha]]C \land [[\beta]]C \\
[[\alpha \cdot \beta]]& = [[\alpha]][[[\beta]]]C \\
[[\alpha^*]]& = C \land [[\alpha]][[[\alpha^*]]]C
\end{align*}
\]

An important requirement when modelling electronic contracts is that the obligation of a sequence of actions \(O_C(\alpha \cdot \alpha')\) must be equal to the obligation of the first action \(O_C(\alpha)\) and after the first obligation is respected the second obligation must hold \(O_C(\alpha')\). To respect the obligation \(O_C(\alpha)\) means to do any action bigger than \(\alpha\) which is captured with the syntactic construction \([\cdot]\). Note that if \(O_C(\alpha)\) is violated then the reparation \(C\) must be enforced (must hold) and the second obligation is discarded. Violating \(O_C(\alpha)\) means that \(\alpha\) is not executed and thus, by the semantic definition, \([[[\alpha]]]O_C(\alpha')\) holds trivially.

**Proposition 3.2.2** The following statements hold:

\[
\begin{align*}
\models [[\alpha_x]]C &\rightarrow [[\alpha_x \times \alpha'_x]]C \quad (3.27) \\
\models O_C(\alpha \cdot \alpha') &\leftrightarrow O_C(\alpha) \land [[\alpha]]O_C(\alpha') \quad (3.28) \\
\models F_C(\alpha \cdot \alpha') &\leftrightarrow F_\gamma(\alpha) \land [[\alpha]]F_C(\beta) \quad (3.29) \\
\models P(\alpha \cdot \alpha') &\leftrightarrow P(\alpha) \land [\alpha]P(\alpha') \quad (3.30)
\end{align*}
\]

where \(\alpha_x, \alpha'_x\in\mathcal{A}_B^\times\).

**Proof:** The proof of (3.27) is easy as we are concerned only with \(\times\)-actions. Trivially, for an action \(\alpha\) the tree of the action \(\alpha\times\text{any}\) has all edges with labels including \(\alpha\) (even more, the tree has depth 1). Similarly, all the edges of the tree of \(\alpha\times\alpha'_x\times\text{any}\) contain \(\alpha\times\alpha'_x\) and, because \(\alpha_x \subseteq \alpha_x \times \alpha'_x\), they contain also \(\alpha_x\). Because of these, all the transitions in the structure that are relevant for evaluating \([[[\alpha_x \times \alpha'_x]]]C\) are part of the transitions relevant for \([[[\alpha_x]]]C\) and, hence, \(C\) holds in all their ending states. Proof finished as whenever \([[[\alpha_x]]]C\) holds in a state, \([[[\alpha_x \times \alpha'_x]]]C\) holds too.

To prove (3.28) we need a series of results which are easy to check but tedious; we just state these results. To prove the left to right implication it is easy to see that \(O_C(\alpha \cdot \alpha') \rightarrow O_C(\alpha)\) as we discuss further. Trivially, \(A^D(\alpha) \subseteq A^D(\alpha \cdot \alpha')\) from which it is easy to deduce that if \(A^D(\alpha \cdot \alpha') \subseteq K^N\) then \(A^D(\alpha) \subseteq K^N\) (i.e., the first line in the semantics of \(O_C(\alpha)\)). From the same results above it is clear that all the transitions \(s \xrightarrow{\gamma} s' \in K^N\) that are relevant for the semantics of \(O_C(\alpha)\) are among the transitions that are relevant in the semantics of \(O_C(\alpha \cdot \alpha')\) and hence they respect the second condition in the semantics of obligation. A second result easy to verify is that \(K^A_{rem}(\alpha, \gamma) \subseteq K^A_{rem}(\alpha \cdot \alpha', \gamma)\) which implies trivially the third condition in the semantics.
of $O_C(\alpha)$. Related to this result is that $A^D(\overline{\alpha}) \subseteq A^D(\overline{\alpha \cdot \alpha'})$ which means that $\text{leafs}(A^D(\overline{\alpha})) \subseteq \text{leafs}(A^D(\overline{\alpha \cdot \alpha'}))$ which makes the last requirement in the semantics of $O_C(\alpha)$ trivially true.

To finish the left to right implication we prove $O_C(\alpha \cdot \alpha') \rightarrow [[\alpha]]O_C(\alpha')$. Recall that the construction of $A^D(\alpha \cdot \alpha')$ first constructs $A^D(\alpha)$ and $A^D(\alpha')$ and then just attaches the whole $A^D(\alpha')$ to all the leafs of $A^D(\alpha)$ (i.e., replaces each leaf with the root of $A^D(\alpha')$). The action $\alpha$ is a deontic action and, hence, its interpretation is a tree and the semantics of $[[\cdot]]$ follows all the transitions in $K^N$ that are bigger than the edges of this tree. Therefore, these are all the transitions that participate in the $A^D(\alpha \cdot \alpha')_S, K^N$; actually in the second condition of the simulation relation. This simple observation gives all the rest of the proof; it implies that all the states where we have to evaluate $O_C(\alpha')$ are part of $A^D(\alpha \cdot \alpha')_S, K^N$, actually $A^D(\alpha')_S, K^N$ where $s$ is related to the leafs of $A^D(\alpha)$. The second and third conditions in the semantics of $O_C(\alpha')$ in the states $s$ follow similarly. The last condition holds because $A^D(\alpha') \subseteq A^D(\alpha \cdot \alpha')$.

The proof of the right to left implication follows a similar tedious argument but the main intuition is as follows. To get the semantics of $O_C(\alpha \cdot \alpha')$ we need to achieve two main goals: (1) to walk on the $K^N$ structure according to $A^D(\alpha \cdot \alpha')$ and to find all the appropriate $\circ$ markers, (2) $C$ must hold at the appropriate states. Walking on $K^N$ goes well and finds all the necessary markers until reaching the leaf nodes of the first part of the tree, i.e., of $A^D(\alpha)$, because it comes from the semantics of $O_C(\alpha)$. Nevertheless, we can continue because all these states are the same as the states reached through $[[\alpha]]$. Therefore, because of the semantics of $O_C(\alpha')$ from all these states we can continue until we reach the leaf nodes of the big tree $A^D(\alpha \cdot \alpha')$. For the second part it is easy to see that all the states of $K^N$ reached because of the tree $A^D(\overline{\alpha \cdot \alpha'})$ are the same as the states reached because of the tree $A^D(\overline{\alpha \cdot \alpha'})$ together with those reach through making first $\alpha$ and then following $A^D(\alpha')$.

The proof of (3.30) follows similar arguments as above. Remark that $[\cdot]$ is used instead, because we have strong simulation in the case of permissions.

The proof of (3.29) is similar only that it reasons about partial simulations. To remark is that the first prohibition has an empty reparation. From a practical point of view the prohibition is irrelevant because it does not impose any restrictions. Technically, the reparation $C$ holds (is enforced) only in the states corresponding to the leafs of the tree $A^D(\alpha \cdot \alpha')$, therefore, there is no information about what holds at the leafs of $A^D(\alpha)$.  \qed

**Theorem 3.2.3 (decidability of poor tests synchronous PDL)** Propositional dynamic logic with synchronous actions and poor tests is decidable in EXPTIME.

**Proof:** The PDL version with automata inside the dynamic modality from [HS83] is proven decidable with a method that builds finite models based on a variant of Fischer-Ladner closure and using Hintikka-like sets. This proof can be easily adapted to our automata over guarded synchronous strings and to our dynamic modality over synchronous actions. Therefore, if we consider only the propositional part and the dynamic modality of $\mathcal{CL}$ we have a decidable extension of PDL which can talk about synchronous actions.  \qed

Unfortunately, we could not give a proof of decidability for the deontic modalities using a Fischer-Ladner closure method. Therefore, we cannot combine the proof based on finite tree
models for the deontic modalities with a proof based on Fischer-Ladner closure for the dynamic modality to obtain the decidability of the full CL.

On the other hand the PDL logic does not have the finite tree model property because of the Kleene *$. This applies to our dynamic modality over synchronous actions too. On the other hand we show in Theorem 3.2.4 that the dynamic modality over synchronous actions has the tree model property. This together with Theorem 3.1.26 give the tree model property for full CL. From the tree model one just needs to find the right selection method to obtain a bounded tree model property (i.e., bounded branching) as was done for the modal $\mu$-calculus [KP83, SE84]. Using this, one can prove decidability by a standard translation into the SnS logic which is decidable [Rab69].

**Theorem 3.2.4 (tree model for CL)** For a pointed structure $< K^N, i >$ we have:

$$TK^N, x \models [\delta]C \iff K^N, \rho(x) \models [\delta]C$$  \hspace{1cm} (3.31)

**Proof:** The proof follows the semantics of $[\cdot]$ modality from Table 3.4 on page 81 and uses an argument similar to what we did in the proof of Theorem 3.1.26 for the last part of the case for obligations. This means that we use structural induction and use the following induction assumption: $TK^N, x' \models C$ iff $K^N, \rho(x') \models C$ for any $x' \in TK^N$.

For the left to right implication we use *reductio ad absurdum* and assume that $\exists s \in K^N$ s.t. $(\rho(x), s) \in R^A_{2A_B}(\delta)$ for which $K^N, s \not\models C$. (This is the negation of the semantics of $[\delta]C$ cf. Table 3.4 on page 81.) Having $(\rho(x), s) \in R^A_{2A_B}(\delta)$ it means that $\exists \sigma = \sigma_0 \ldots \sigma_k$ a final path in $A^G(\delta)$ and $\exists s_0 \ldots s_k \in K^N$ s.t. $s_0 = \rho(x), s_k = s$, $\forall 0 \leq i \leq k$, $\forall \sigma_i \in L(\sigma_i)$, and for any $\sigma_i \xrightarrow{\alpha_i} \sigma_{i+1} \in \sigma$ we have $(s_i, s_{i+1}) \in R^A_{2A_B}(\alpha_i)$. By Lemma 3.1.23 it means that for $(\rho(x), s_1) \in R^A_{2A_B}(\alpha_0)$ we find $(x, x\alpha_0s_1) \in R^T_{2A_B}(\alpha_0)$; and the same for all $i$, ending with the transition $(x\alpha^0_{s_0} \ldots s_{k-1}, x\alpha^0_{s_0} \ldots \alpha^k_{s_k}) \in R^T_{2A_B}(\alpha^k-1)$. Because the valuation functions $\forall$ and $\forall^T$ agree on all propositional constants $\phi$ it means that $\forall^T(x\alpha^0_{s_0} \ldots s_i) \in L(\sigma_i)$. In this way we have found for the final path $\sigma$ the sequence of states $x, x\alpha^0_{s_1}, \ldots, x\alpha^0_{s_k}$ in the tree model that satisfy the conditions for $(x, x\alpha^0_{s_0} \ldots \alpha^k_{s_k}) \in R^T_{2A_B}(\delta)$ and, by the left part of the implication, we have $TK^N, x\alpha^0_{s_0} \ldots \alpha^k_{s_k} \models C$. We use the inductive hypothesis to obtain $K^N, \rho(x\alpha^0_{s_0} \ldots \alpha^k_{s_k}) \models C$ which is the same as $K^N, s_k \models C$; but $s_k = s$ and hence we get the contradiction.

The right to left implication follows analogous arguments. \hspace{1cm} \boxed{}$

### 3.2.2 Paradoxes and Puzzles

The following propositions show that the most important paradoxes of deontic logic are avoided in CL, either because they are not expressible in the language or because they are excluded by the semantics.

**Ross’s Paradox** [Ros41] in natural language it is expressed as:

| a. | “It is obligatory that one mails the letter”; |

---

*Note: This is a simplified transcription and may not capture the exact syntax or context of the original text.*
b. “It is obligatory that one mails the letter or one burns the letter”.

In SDL these are expressed as:

a. \( O(p) \),
b. \( O(p \lor q) \).

The problem is that in SDL one can make the inference \( O(p) \to O(p \lor q) \).

**Proposition 3.2.5** Ross’s paradox does not hold in CL.

**Proof:** Basically, Ross’s paradox says that it is counter intuitive to have \( O(a) \to O(a + b) \) (e.g., “Obligation to drink implies obligation to drink or to kill”). In CL this inference is not possible as witnessed by Proposition 3.1.21(3.18).

**The Good Samaritan Paradox** [Pri58] in natural language it is expressed as:

a. “It ought to be the case that Jones helps Smith who has been robbed”;
b. “It ought to be the case that Smith has been robbed”;
c. and the natural inference “Jones helps Smith who has been robbed if and only if Jones helps Smith and Smith has been robbed”.

In SDL the first two are expressed as:

a. \( O(p \land q) \);
b. \( O(q) \).

The problem is that in SDL one can derive that \( O(p \land q) \to O(q) \) which is counter intuitive in the natural language.

**Proposition 3.2.6** The Good Samaritan paradox can not be expressed in CL.

**Proof:** The Good Samaritan paradox uses *ought-to-be* and is more delicate to transform it into our *ought-to-do* approach. The transformation looks like: \( \varphi \to O(a) \) which means that “If Smith has been robbed (i.e., \( \varphi \)) then is obligatory that John helps Smith (i.e., \( O(a) \))”. We can not express in CL obligations over conjunction of two actions that are not performed concurrently as this paradox is expressed in SDL. Also, with our representation of the paradox we cannot deduce \( \varphi \); i.e., that Smith has been robbed.

**The Free Choice Permission Paradox** [Ros41] in natural language it is expressed as:

a. “You may either sleep on the sofa or sleep on the bed”;
b. “You may sleep on the sofa and you may sleep on the bed”.

In SDL this is:
a. \( P(p \lor q) \),
b. \( P(p) \land P(q) \).

The natural intuition tells that \( P(p \lor q) \rightarrow P(p) \land P(q) \). In SDL this leads to \( P(p) \rightarrow P(p \lor q) \) which is \( P(p) \rightarrow P(p) \land P(q) \), so \( P(p) \rightarrow P(q) \). As an example: “If one is permitted something, then one is permitted anything”.

**Proposition 3.2.7** The Free Choice Permission paradox does not exist in CL.

**Proof:** The Free Choice Permission paradox basically says that from having one permission we may infer that we have any permission. That is: \( P(a) \rightarrow P(a + b) \) or \( P(a) \rightarrow P(a) \land P(b) \). Neither of the two implications hold in our approach. The second one is obvious. The first one is ruled out by Proposition 3.1.21-(3.23) which is a consequence of Proposition 3.1.20-(3.13). \( \square \)

**Sartre’s Dilemma [McN06]** in natural language is expressed as:

a. It is obligatory to meet Jones now (as promised to Jones);
b. It is obligatory to not meet Jones now (as promised to Smith).

In SDL this is:

a. \( O(p) \),
b. \( O(\neg p) \).

The problem is that in the natural language the two obligations are intuitive and often happen, where the logical formulas are inconsistent when put together (in conjunction) in SDL.

**Proposition 3.2.8** Sartre’s Dilemma is not expressible in our approach.

**Proof:** Sartre’s dilemma can be rewritten in contracts terminology as: Obliged to meet John and Forbidden to meet John. This is written in \( \mathcal{CL} \) as \( O(a) \land F(a) \) which is a well formed formula. But this results in a contradiction because of Corollary 3.1.19-(3.8). \( \square \)

**Chisholm’s Paradox [Chi63]** in natural language is expressed as:

a. John ought to go to the party;
b. If John goes to the party then he ought to tell them he is coming;
c. If John does not go to the party then he ought not to tell them he is coming;
d. John does not go to the party.

In SDL these are expressed as:

a. \( O(p) \),
b. \( O(p \rightarrow q) \),
c. \( \neg p \rightarrow O(\neg q) \),

d. \( \neg p \).

The problem is that in SDL one can infer \( O(q) \land O(\neg q) \) which is due to statement 2.

**Proposition 3.2.9** The Chisholm’s paradox is avoided in \( \mathcal{CL} \).

**Proof:** The propositions of the Chisholm’s paradox are expressed in \( \mathcal{CL} \) as: a. \( O(a) \), b. \( [a]O(b) \), c. \( [\overline{a}]O(\overline{b}) \). Note first that formulas a. and c. give the CTD formula \( O_C(a) \) of \( \mathcal{CL} \) where \( C = O(\overline{b}) \). The problem in SDL was that one may infer both \( O(b) \) and \( O(\overline{b}) \) holding in the same world. This is not our case because \( O(b) \) holds only after doing action \( a \), where \( O(\overline{b}) \) holds only after doing the contradictory action \( \overline{a} \). Therefore, we can not have in the same world both \( O(b) \) and \( O(\overline{b}) \).

---

**More discussions about \(*\) inside deontic modalities**

One may argue for using the \(*\) not standalone (as we discussed in the introduction) but in a more elaborated action. A first try would be to have \( a^+ \) or for the same arguments \( b \cdot a^* \) or any finite prefix followed by the repetition \( a^* \). In any of these cases we reduce to the same argument as for the \( a^* \) alone. This is because the obligation of a sequence of actions is the same as the sequence of obligations. Therefore, \( O(b \cdot a^*) \) is equivalent to the obligation of \( b \) and after executing \( b \) the obligation of \( a^* \) which gives the same paradox as presented in the introduction.

---

**3.3 Final Remarks**

In this chapter we have presented a formal language for writing contracts, and have provided a formal semantics over, what we call, normative structures. We showed how \( \mathcal{CL} \) avoids most of the classical paradoxes, and enjoys some desired properties for contracts. Given that our application domain is that of electronic contracts, we have also given arguments for restricting syntactically and semantically certain uses of (and relations between) obligations, permissions and prohibitions, usually considered in philosophical and logical discussions. We have shown important results of tree model property for \( \mathcal{CL} \) and decidability results.

---

**3.3.1 Related Work**

There are currently several different approaches aiming at defining a formal language for contracts. Some works concentrate on the definition of contract taxonomies [Aag01, BJP99, TP05], while others look for formalizations based on logics (e.g., classical [DKR04], modal [DM01], deontic [GR06, PDK05] and defeasible logic [Gov05, SG05]). Other formalizations are based on models of computation (e.g., FSMs [MJSSW04] and Petri Nets [Das00]). None of the above has reached enough maturity as to be considered the solution to the problems of formal definition of contracts. Some provide a good framework for monitoring but lack a formal semantics and a reasoning system; others have nice proof systems and model theory, but not mechanism.
for monitoring or negotiation; many of the deontic-based approaches put too much emphasis on the logical properties and neglect the practical side, including monitoring. None of them captures all the intuitive properties of e-contracts we have described, while avoiding the most important paradoxes.

Our work is closely related to those based on logic, and in particular to [BWM01] where an interesting characterization of obligation, permission and prohibition is done following an *ought-to-do* approach based on a deontic logic of regular actions. Their idea is to use $\mu$-calculus as a basis and then define obligation, permission and prohibition over regular expressions on actions. The main differences w.r.t. our approach are the following. (a) There is no notion of *contract language*, only characterization of obligation, permission and prohibition in the logic. (b) The only deontic primitive is permission over atomic actions; obligation is defined as an infinite conjunction of negation of permission over actions not in the scope of the negation. We avoid this infinite conjunction by defining permission, prohibition, and obligation as primitives. (c) All the deontic operators are defined over regular actions, including the Kleene star. We consider it is not natural to have starred actions under the deontic notions, we have thus dropped it. (d) Obligation on the choice of actions is not compositional; it is compositional in our case. (e) There is no conjunction over actions, i.e., it is not possible to express concurrent actions, which is the case in our approach. (g) Negation on actions (meaning “not performing an action”) is defined as a complement of the (infinite) set of actions. In our case the complement operator over the $^*\!$-free/deontic actions returns a finite action (with a finite representation tree). (h) CTDs cannot be defined unless an extension of the approach of [BWM01] is considered. In our setting both CTDs and CTPs are primitives. (i) The semantics of obligation, permission and prohibition is given in terms of properties over traces, instead of over an extension of the Kripke structure as in our case.

The idea of marking states in an action-based logic for giving semantics to the deontic notions was first presented in [Mey88], where the special propositional constant $V$ was added to denote an “undesirable state-of-affairs” in the current state. We use a marking function which marks states with obligation markers $\circ$ or prohibition markers $\bullet$, deviating from other approaches (e.g., [Bro03, Mey88]).

For a nice overview of the history, problems and different approaches on deontic logic see [vW99]. The chapter of McNamara in the Handbook of the History of Logic contains a general description of the topic, mainly the different paradoxes arising under SDL [McN06]. For a discussion on CTDs see [PS96] and references therein.

### 3.3.2 Open problems

There are several problems (some more challenging than the other) that have not been considered in this chapter. One problem is to extend $CL$ by adding real-time, so to be able to express and reason about contracts with deadlines. Other immediate extension is the syntactic distinction between subjects, proper actions and objects. This would permit to make queries (and model check properties) for instance about all the rights and obligations of a given subject (partner in the contract), or determine under which conditions somebody is obliged/forbidden to perform something.

We have not considered in this chapter the problem of negotiation. We believe this is an
important feature of a contract language which must be taken into account. For the $\mathcal{CL}$ logic an important technical tool is a proof system, preferably in the style of tableaux. This is hard to obtain as the semantics does not offer a decomposition view, i.e., depending on the subformulas (we do not know what are the subformulas of an obligation of a complex action, as is done in PDL for the box operator). In the next section the restricted semantics on traces manages to reveal such subformula decomposition, but for the full semantics of this chapter we were not able to find such.

Concerning actions, we got inspiration from the works on dynamic logics [Pra76]. We would like to deepen the study of the action algebra to make the distinction between the intuitive meaning of conjunction under obligation, permission and prohibition. Further investigation is also needed to characterize negation on actions, both for capturing and distinguishing the ideas of “not doing something” and “doing something different than a given action”, which are not differentiated in our current approach.
Chapter 4

Run-time Monitoring of Electronic Contracts

Electronic inter-organizational relationships are governed by contracts regulating their interaction. It is necessary to run-time monitor the contracts, as to guarantee their fulfillment as well as the enforcement of penalties in case of violations. The work presented in this chapter shows how to obtain a run-time monitor for a contract written in $CL$. We first give a trace semantics for $CL$ which formalizes the notion of a trace fulfills a contract. We show how to obtain, for a given contract, an alternating Büchi automaton which accepts exactly the traces that fulfill the contract. This automaton is the basis for obtaining a deterministic finite state machine which acts as a run-time monitor for $CL$ contracts.

4.1 Preliminaries

The main contribution of this chapter is an automatic procedure for obtaining a run-time monitor for contracts, directly extracted from the $CL$ specification. We give a trace semantics for the expressions of $CL$ in Section 4.2. This captures the fact that a trace respects (does not violate) a contract clause (expression of $CL$). In Section 4.3.1 we show how to construct for a contract an alternating Büchi automaton which recognizes exactly all the traces respecting the contract. The automaton is used in Section 4.3.2 for constructing the monitor as a Moore machine (for monitoring the contract).

We recall in Table 4.1 the syntax of $CL$ that we use for run-time monitoring. Because in this chapter we are interested in monitoring the actions in a contract, we give here a slightly different version of $CL$ where we have dropped the assertions (the propositional constants $\phi \in \Phi_B$) from the syntax of $CL$, keeping only the modalities over actions. The formal semantics in terms of traces is given later in Section 4.2 (see Table 4.2).

Though in this chapter we concentrate on theoretical aspects, we show the feasibility of our approach on the following small didactic example. The example states one contract clause which we use throughout the chapter to exemplify some of the main concepts we define. (This is a paraphrasing combining clauses 7.2 and 7.3 of the contract example in Appendix B or from the introductory section 1.2.)
\[
\begin{align*}
\mathcal{C} & := O_C(\alpha) \mid P(\alpha) \mid F_C(\alpha) \mid C \rightarrow C \mid [\delta]C \mid \bot \\
\mathcal{C} & := C \lor C \mid C \land C \mid C \oplus C \quad \text{(derived Boolean operations)} \\
\alpha & := a \mid 0 \mid 1 \mid \alpha \times \alpha \mid \alpha \cdot \alpha \mid \alpha + \alpha \\
\delta & := a \mid 0 \mid 1 \mid \delta \times \delta \mid \delta \cdot \delta \mid \delta + \delta \mid \delta^* \mid \phi ? \\
\varphi & := \phi \mid 0 \mid 1 \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \neg \varphi
\end{align*}
\]

Table 4.1: Syntax of the contract language CL used for monitoring.

Example 4.1.1 “If the Client exceeds the bandwidth limit then (s)he must pay \[\text{price}\] immediately, or (s)he must delay the payment and notify the Provider by sending an e-mail. If in breach of the above (s)he must pay double.”

In Example 4.1.1 the basic actions are \(A_B = \{e, p, n, d\}\) (standing for “extend bandwidth limit”, “pay”, “notify by email”, and “delay”). An example of a concurrent action is \(d \times n \in A_B^\times\), or even \(e \times p \times d \times n\).

From Example 4.1.1 consider the complement \(p + d \times n\) of the action “pay or delay and notify by e-mail”. Any action \(\gamma\) which does not "contain" neither \(p\) nor \(d \times n\) (i.e. \(p \not\sqsubseteq \gamma\) and \(d \times n \not\sqsubseteq \gamma\)) is part of the complement of \(p + d \times n\); e.g. \(p \times e \not\sqsubseteq p + d \times n\), but \(d, e, d \times e \in p + d \times n\).

Example 4.1.1 in CL syntax: The transition from the conventional contract given in the introduction to a formal representation is manual. The following is the CL expression that we obtain.

\[[e]O_{O.(p \cdot p)}(p + d \times n)\]

In short the expression is read as: After executing the action “exceed bandwidth limit” there is the obligation of choosing between either “paying” or at the same time “delay the payment” and “notify by e-mail”. The CL expression also states the reparation \(O_{\perp}(p \cdot p)\) in case the obligation above is violated, which itself is an obligation of doing twice in a row the action of paying. Note that this second obligation has no reparation attached, therefore if it is violated then the whole contract is violated. Note also that we translate “pay double” into the CL sequential composition of the same action \(p\) of paying.

### 4.2 Semantics on Respecting Traces

The present section is devoted to presenting a semantics for CL with the goal of monitoring electronic contracts. For this we are interested in identifying the respecting and violating traces of actions. We follow the many works in the literature which have a presentation based on traces e.g. [Pra79]. We first give brief definitions used throughout the rest of this chapter.

**Definition 4.2.1 (traces)** Consider a trace denoted \(\sigma = a_0, a_1, \ldots\) as an ordered sequence of concurrent actions. Formally a trace is a map \(\sigma : \mathbb{N} \rightarrow A_B^\times\) from natural numbers (denoting positions) to concurrent actions from \(A_B^\times\). Take \(m_\sigma \in \mathbb{N} \cup \infty\) to be the length of a trace.\(^1\) A (infinite) trace which from some position \(m_\sigma\) onwards has only action 1 is considered finite. We use \(\varepsilon\) to denote the empty trace. We denote by \(\sigma(i)\) the element of \(\sigma\) at position \(i\), by \(\sigma(i..j)\)

\(^1\)When \(\sigma\) is obvious from the context we use just \(m\) instead of \(m_\sigma\).
a finite subtrace, and by \( \sigma(i..) \) the infinite subtrace starting at position \( i \) in \( \sigma \). The concatenation of two traces \( \sigma' \) and \( \sigma'' \) is denoted \( \sigma' \sigma'' \) and is defined iff the trace \( \sigma' \) is finite: \( \sigma' \sigma''(i) = \sigma'(i) \) if \( i < m_o \), and \( \sigma' \sigma''(i) = \sigma''(i - m_o) \) for \( i \geq m_o \) (e.g. \( \sigma(0) \) is the first action of a trace, \( \sigma = \sigma(0..i) \sigma' \) where \( \sigma' = \sigma(i+1..) \)).

For the technical developments of this chapter we give below an alternative definition of trees in terms of sets of strings over \( \mathbb{N} \).

**Definition 4.2.2 (trees as subsets of \( \mathbb{N}^\omega \))** A tree is a prefix-closed subset \( T \subset \mathbb{N}^\omega \) s.t. if \( x c \in T \) with \( x \in T \) a tree node and \( c \in \mathbb{N} \) then \( x c' \in T \), \( \forall c' < c \). We call \( xc \) the successors of \( x \). A node with no successors is called leaf. The root of the tree is the empty string \( \varepsilon \). A \( \Sigma \)-labeled tree is a pair \( (T, V) \) where the valuation function \( V(x) \in \Sigma \) assigns to each node an element of the alphabet \( \Sigma \). A path \( \tau \) in a tree \( T \) is a set \( \tau \subseteq T \) s.t. \( \varepsilon \in \tau \) and if \( x \in \tau \) then either \( x \) is a leaf (in which case \( \tau \) is called a full path) or \( \exists c \in \mathbb{N} \) unique and \( xc \in \tau \). We denote a path by \( x_0, x_1, \ldots \). A path of a \( \Sigma \)-labeled tree \( \tau = x_0, x_1, \ldots \) defines an (in)finite word \( \alpha = V(x_0), V(x_1), \ldots \) over \( \Sigma \). We denote by \( |x| \) the depth of the node \( x \) in the tree.

We know from Section 2.2.5 that deontic actions can be interpreted as trees. For example
the action \( a + b \) is interpreted as the tree \( A^D(a + b) = (\{\varepsilon, \varepsilon 0, \varepsilon 1\}, V) \) where \( V(\varepsilon 0) = a \) and \( V(\varepsilon 1) = b \). Intuitively, \( + \) provides the branching in the tree, and \( \cdot \) provides the parent-child relation on each branch. The node labels from \( A^D_\alpha \) encode the concurrency operator \( \times \). A trace \( \sigma \) is said to be contained in a \( A^D_\alpha \)-labeled tree \( (T, V) \) iff \( \exists \tau \subseteq T \) a path and \( \sigma(0) = V(\varepsilon c) \), with \( \varepsilon c \in \tau \), and if \( \sigma(i) = V(x) \) then \( \sigma(i + 1) = V(x c) \) with \( x, x c \in \tau \) (where in all the above and in all the rest we consider \( c \in \mathbb{N} \)). (Denote this by \( \sigma \in T \) as shorthand for \( \sigma \in (T, V) \)). Naturally, any trace \( \sigma \) which is contained in a tree \( A^D(\alpha) \) of an action \( \alpha \) is finite as the trees which interpret deontic actions are of finite depth. We consider here the set of all traces which are full paths in the tree \( T \) and denote it by \( |T| = \{\sigma | \sigma \text{ a full path in } T\} \).

From Section 2.3.3 we also know that there is a one-to-one (i.e., the completeness result) relation between the actions and their tree interpretations. Therefore we can consider actions of \( A^D \) as the associated set of traces of the tree returned by \( A^D \).

We have given the negation of actions as a derived operator \( \overline{\cdot} : A \rightarrow A \) in Definition 2.2.17. The negation operator takes an action in canonical form (as we know from Theorem 2.2.10 that for each action there exists an equivalent canonical form) and returns another action also in canonical form, cf. Proposition 2.2.18. It may be intuitive that the set of traces associated to an action \( \alpha \) is \( |A^D(\alpha)| \), but it is not so intuitive what is the set of traces for a negation of an action \( \overline{\alpha} \). The following result shows explicitly the set of traces which correspond to the action negation. This helps in understanding better the semantic definition of \( [\overline{\alpha}]C \). It also helps in the proof of Proposition 4.2.5.

**Proposition 4.2.3 (characterizing action negation with traces)** The set of traces that are full paths of the tree interpreting the negation of an arbitrary *-free action \( \alpha = +_{i \in I} \alpha_i \times \cdot \alpha_i \) in its canonical form (i.e., the set \( |A^D(\overline{\alpha})| \)) is equal to the following set of traces:

\[
|A^D(\overline{\alpha})| = \{\sigma | \sigma = (0)\varepsilon \land \forall i \in I, \alpha_i \not\subseteq \sigma(0)\} \cup \\
\{\sigma | \sigma \subseteq (0)\sigma(1..) \land \exists i \in I, s.t. \alpha_i \neq 1 \land \alpha_i = \sigma(0) \land \\
\sigma(1..) \in |A^D(+_{j \in I'} \alpha_j)| with I' = \{j \in I | \alpha_j \not\subseteq \sigma(0)\}\}.
\]
\[
\sigma \models C_1 \land C_2 \text{ if } \sigma \models C_1 \text{ and } \sigma \models C_2.
\]
\[
\sigma \models C_1 \lor C_2 \text{ if } \sigma \models C_1 \text{ or } \sigma \models C_2.
\]
\[
\sigma \models C_1 + C_2 \text{ if } (\sigma \models C_1 \text{ and } \sigma \not\models C_2) \text{ or } (\sigma \not\models C_1 \text{ and } \sigma \models C_2).
\]
\[
\sigma \models [\alpha_x]\mathcal{C} \text{ if } \alpha_x \subseteq \sigma(0) \text{ and } \sigma(1..) \models \mathcal{C}, \text{ or } \alpha_x \not\subseteq \sigma(0).
\]
\[
\sigma \models [\beta \cdot \beta']\mathcal{C} \text{ if } \sigma \models [\beta]\mathcal{C} \text{ and } \sigma \models [\beta']\mathcal{C}.
\]
\[
\sigma \models [\beta + \beta']\mathcal{C} \text{ if } \sigma \models [\beta]\mathcal{C} \text{ and } \sigma \models [\beta']\mathcal{C}.
\]
\[
\sigma \models C_1 ? C_2 \text{ if } \sigma \not\models C_1, \text{ or if } \sigma \models C_1 \text{ and } \sigma \models C_2.
\]
\[
\sigma \models O_C(\alpha_x) \text{ if } \alpha_x \subseteq \sigma(0), \text{ or if } \sigma(1..) \models \mathcal{C}.
\]
\[
\sigma \models O_C(\alpha_x) \alpha' \text{ if } \sigma \models O_C(\alpha) \text{ and } \sigma \models [\alpha]O_C(\alpha').
\]
\[
\sigma \models O_C(\alpha + \alpha') \text{ if } \sigma \models O_\bot(\alpha) \text{ or } \sigma \models O_\bot(\alpha') \text{ or } \sigma \models [\alpha + \alpha']\mathcal{C}.
\]
\[
\sigma \models F_C(\alpha_x) \text{ if } \alpha_x \not\subseteq \sigma(0), \text{ or if } \alpha_x \subseteq \sigma(0) \text{ and } \sigma(1..) \models \mathcal{C}.
\]
\[
\sigma \models F_C(\alpha_x) \alpha' \text{ if } \sigma \models F_\bot(\alpha) \text{ or } \sigma \models [\alpha]F_C(\alpha').
\]
\[
\sigma \models F_C(\alpha + \alpha') \text{ if } \sigma \models F_C(\alpha) \text{ and } \sigma \models F_C(\alpha').
\]
\[
\sigma \models [\alpha_x]\mathcal{C} \text{ if } \alpha_x \not\subseteq \sigma(0) \text{ and } \sigma(1..) \models \mathcal{C}, \text{ or if } \alpha_x \subseteq \sigma(0).
\]
\[
\sigma \models [\alpha + \alpha']\mathcal{C} \text{ if } \sigma \models [\alpha]\mathcal{C} \text{ and } \sigma \models [\alpha']\mathcal{C}.
\]
\[
\sigma \models [\alpha + \alpha']\mathcal{C} \text{ if } \sigma \models [\alpha]\mathcal{C} \text{ or } \sigma \models [\alpha']\mathcal{C}.
\]

Table 4.2: Trace semantics of \(\mathcal{C}\).

**Proof:** The definition of the set of traces is inductive. This is because the definition of the canonical form of actions is inductive and therefore also the action negation. The tree \(A^D(\pi)\) is the same as \(A^D(\gamma_{i\in I} \alpha_x \alpha^i)\) which is \(A^D(\gamma_{i\in I} \beta_x + \gamma_{j\in J} \gamma_{i\in I} \alpha^i)\) as in Definition 2.2.17. The first part of the action, i.e. \(\gamma_{i\in I} \beta_x\) gives the full paths of the tree of length 1 (i.e. on the first level). It is simple to observe that these paths are captured by the first set of traces \(\{\sigma \mid \sigma = (\sigma(0)e \land \forall i \in I, \alpha_x \subseteq \sigma(0)\}\}. These are traces with one element \(\sigma(0)\) (i.e. ending in the empty trace \(\epsilon\)) and they respect the same condition like in the definition of \(\Pi\).

Note that when \(I\) is a singleton and we have only one \(\alpha_x\) then the condition \(\alpha_x \not\subseteq \sigma(0)\) becomes \(\sigma(0) = \sigma'(0) \cup \sigma''(0) \land \sigma'(0) \subseteq \alpha_x \land \sigma''(0) \subseteq \alpha_x\). We read this condition as: the first element of \(\sigma\) (which we recall is a set of basic actions) has some actions, i.e. \(\sigma'(0)\), among those, but not all, of the basic actions of \(\alpha_x\) and the other basic actions, i.e. \(\sigma''(0)\) are different than those of \(\alpha_x\); i.e. are among \(\varepsilon \cup \alpha_x\).

The other full paths of length greater than 1 of the tree are given by the second part of the action, i.e. \(\gamma_{j\in J} \alpha_x \alpha^j\). All these paths are captured by the second set of traces which are of length at least 2. All branches of \(\gamma_{i\in I} \alpha_x \alpha^i\) where \(\alpha^i = 1\) disappear. This is because when negating \(\overline{\alpha} = I = 0\) that branch ends in \(0\) which propagates upwards by \(\alpha \cdot 0 = 0\) and disappears eventually by \(\alpha + 0 = \alpha\). Therefore we are looking only at traces s.t. \(\alpha^i \neq 1\). From these we take the traces which start with a set of basic actions that include the action \(\alpha_x\) (i.e. \(\alpha_x \subseteq \sigma(0)\)) and are followed by a trace (i.e. \(\sigma(1..)\)) which is part of the traces of the negation of the smaller compound action \(\gamma_{i\in I} \alpha^i\). \(\square\)
Definition 4.2.4 (trace semantics of $C\mathcal{L}$) We give in Table 4.2 a recursive definition of the satisfaction relation $\models$ over pairs $(\sigma, C)$ of traces and contracts; it is usually written $\sigma \models C$ and it is read as “trace $\sigma$ respects the contract (clause) $C$”. We write $\sigma \not\models C$ instead of $(\sigma, C) \notin \models$ and read it as “$\sigma$ violates $C$.” We say that a formula $C$ is valid and denote it by $\models C$ iff $\forall \sigma, \sigma \models C$. A formula is not valid, denoted $\not\models C$ iff $\exists \sigma$ s.t. $\sigma \not\models C$.

Since some of the cases from the definition of $\models$ are typical for modal (dynamic) logics, like the first lines corresponding to the semantics for the derived operators $\land, \lor, \odot$, we only comment on those that are particular to our logic. A trace $\sigma$ respects an obligation $O_C(\alpha_x)$ if either of the two complementary conditions (see Proposition 4.2.5) is satisfied. The first condition deals with the obligation itself: the trace $\sigma$ respects the obligation $O(\alpha_x)$ if the first action of the trace includes $\alpha_x$. Otherwise, in case the obligation is violated, the only way to fulfill the contract is by respecting the reparation $C$; i.e. $\sigma(1..) \models C$. Respecting an obligation of a choice action $O_C(\alpha_1 + \alpha_2)$ means that it must be executed one of the actions $\alpha_1$ or $\alpha_2$ completely; i.e. obligation needs to consider only one of the choices. If none of these is entirely executed then a violation occurs (thus the negation of the action is needed) so the reparation $C$ must be respected. An important requirement when modelling electronic contracts is that the obligation of a sequence of actions $O_C(\alpha \cdot \alpha')$ must be equal to the obligation of the first action $O_C(\alpha)$ and after the first obligation is respected the second obligation must hold $[\alpha]O_C(\alpha')$. Note that if $O_C(\alpha)$ is violated then it is required that the second obligation is discarded, and the reparation $C$ must hold. Violating $O_C(\alpha)$ means that $\alpha$ is not executed and thus, by the semantic definition, $[\alpha]O_C(\alpha')$ holds regardless of $O_C(\alpha')$.

The proof of Proposition 4.2.5 provides for a more clear understanding of why the two conditions in the semantical definition of $O_C(\alpha_x)$ are complementary and therefore why the second condition does not need the explicit specification of the negation of the concurrent action. The same result is also useful in giving the completeness of the semantical definitions of $[\alpha_x]C$, $[\alpha_x]C$, or $F_C(\alpha_x)$. That is, we get as an immediate corollary that the conditions on traces cover all the possible traces. For convenience we define an enclosing relation over traces $\sigma \supseteq \sigma'$ iff $\forall i \in \mathbb{N}, \sigma'(i) \subseteq \sigma(i)$. Note that this definition requires that $m_\sigma \geq m_{\sigma'}$.

Proposition 4.2.5 For an arbitrary action $\alpha$, any infinite trace $\sigma$ is either starting with a trace bigger w.r.t. $\supseteq$ then a complete path of $A^D(\alpha)$ or it starts with a trace bigger than a complete path of $A^D(\overline{\alpha})$.

Proof: The proof is by reductio ad absurdum. If the trace $\sigma \supseteq \sigma A^D(\alpha) \sigma'$ starts with a full path of the tree $A^D(\alpha)$ the proof is finished. Suppose it is not the case that $\sigma \supseteq \sigma A^D(\alpha) \sigma'$. This means that $\exists i \leq h(A^D(\alpha))$ s.t. $\sigma(0..i - 1) \supseteq \sigma A^D(\alpha)(0..i - 1)$ and for all possible actions $\sigma A^D(\alpha)(i)$ of extending the trace $\sigma A^D(\alpha)(0..i - 1)$ in the tree $A^D(\alpha)$ it is the case that $\sigma(i) \not\supseteq \sigma A^D(\alpha)(i)$. Consider the characterization of the negation of Proposition 4.2.3. It is easy to see that the trace $\sigma A^D(\alpha)(0..i - 1) \sigma(i)$ is a full path of the tree $A^D(\overline{\alpha})$ interpreting the negation of $\alpha$ because the first part is a trace of the action $\alpha$ and the last step of the trace respects the condition in the first set of traces of Proposition 4.2.3. More explicitly, in Proposition 4.2.3 $\forall i \in I, \alpha^i$ means that for all branches... each action on the branch must not be less than the current element of the trace. This is the same as the argument needed above.

Footnote 2: Violation of an obligatory action is encoded by the action negation.
Considering that $\sigma^{A^P(a)}(0..i - 1)\sigma(i)$ is a full path of $A^D(\pi)$ and that $\sigma(0..i - 1) \supseteq \sigma^{A^P(a)}(0..i - 1)$ we finish the proof as the trace $\sigma$ is starting with the trace $\sigma(0..i) \supseteq \sigma^{A^P(\pi)}$ which is greater than a full path of the tree interpreting the negation of $\alpha$.

The semantics of Table 4.2 is defined s.t. it captures some intuitive properties one finds in legal contracts; we list them in Proposition 4.2.6.

**Proposition 4.2.6 (properties on traces)**

$\forall C(a) \land \forall C(b) \Leftrightarrow \forall C(a \times b)$ (1)

$\neg O(a + b) \Rightarrow O(a \times b)$ (2)

$\neg O(a + b) \Rightarrow O(a)$ (3)

$F(a + b) \Leftrightarrow F(a) \land F(b)$ (4)

$F(a \cdot b) \Leftrightarrow F(a) \lor [a]F(b)$ (5)

$\neg F(a \times b) \Rightarrow F(a)$ (6)

$[a_x]C \Rightarrow [a_x \times a'_x]C$ (7)

$[\beta]C_1 \land [\beta']C_2 \Rightarrow [\beta \times \beta']C_1 \land C_2$ (8)

**Proof:** The proofs of these properties is routine. The method is the classical one for validity of implication where we need to look at all and only the models which satisfy the formula on the left of the implication and make sure that the models satisfies also the formula on the right.

For property (1), we must prove two implications. We deal first with the $\Rightarrow$ one. Take a trajectory $\sigma$ s.t. it satisfies the formula on the left, i.e. $\sigma \models O_C(a)$ and $\sigma \models O_C(b)$. We are in the simple case when we consider basic actions $a$ and $b$. We look at the semantics of obligation. If it is the case that $\sigma(1..) \models \forall C$ then it is clear that $\sigma \models O(a \times b)$. Otherwise we have the case when both $\sigma(0) \supseteq a$ and $\sigma(0) \supseteq b$. It implies that $\sigma(0) \supseteq a \times b$ which means that $\sigma \models O(a \times b)$. The second implication $\Leftarrow$ is simpler and uses the same judgement.

Properties (2), (4), or (6) are similar.

For property (3) we need to give a counterexample. Clearly a trace starting with $\sigma(0) = a$ satisfies $O(a + b)$ but does not satisfy $O(a \times b)$. We find similar counterexamples for properties like (5) or (7).

For properties (8) and (9) we again need to show that for all $\sigma$ which respect the formula on the left of the arrow they also respect the formula on the right.

From [HKT00] we know how to encode LTL over finite traces only with the dynamic $\gamma$ modality and the Kleene $*$; e.g. “always obliged to do $\alpha$” is encoded as $[(+\gamma \in A_n^\alpha)^*]O(\alpha)$. The action $+\gamma \in A_n^\alpha$ is read as “choice between any concurrent action” and we denote it by any.

**Example 4.1.1 as traces:** Consider the expression on page 92 which encodes in $CL$ the contract clause of Example 4.1.1 from the introduction. We give here few examples of traces of actions which respect the contract clause:

- $\sigma = e, p$ — (“exceed bandwidth limit” and then “pay”) which respects the contract because it respects the top level obligation;

- $\sigma = e, d, p, p$ — (“exceed bandwidth limit” and then “delay payment” after which “pay” twice in a row) which even if it violates the top level obligation because it does not notify by e-mail at the same time when “delaying payment”, it still respects the reparation by paying twice;

- $\sigma = p, p, p$ — (“pay” three times in a row) because every trace which does not start with the action $e$ respects the contract.

Examples of traces which violate the clause are:
• \( \sigma = e, e, e \) – (constantly “exceeding the bandwidth limit”) which violates both the first obligation and the second one by not paying;

• \( \sigma = e, d, d \) – (after “exceeding the bandwidth limit” it constantly “delays the payment”) which again violates both obligations.

4.3 Monitoring CL Specifications of Contracts

4.3.1 Satisfiability checking for CL using alternating automata

Automata theoretic approach to satisfiability of temporal logics was introduced in [VW86] and has been extensively used and developed since (see [KVW00] for a more recent overview of the field and a particularly detailed presentation of alternating tree automata and the automata approach to branching time logics). We recall first basic theory of automata on infinite objects.

We follow the presentation of Vardi [Var95, Var97] and try to use the same terminology and notation. Given an alphabet \( \Sigma \), a word over \( \Sigma \) is a sequence \( a_0, a_1 ... \) of symbols from \( \Sigma \). The set of infinite words is denoted by \( \Sigma^\omega \).

We denote by \( B^+(X) \) the set of positive Boolean formulas \( \theta \) (i.e. containing only \( \land \) and \( \lor \), and not the \( \neg \)) over the set \( X \) together with the formulas \( \text{true} \) and \( \text{false} \). For example \( \theta = (s_1 \lor s_2) \land (s_3 \lor s_4) \) where \( s_i \in X \). A subset \( Y \subseteq X \) is said to satisfy a formula \( \theta \) iff the truth assignment which assigns \( \text{true} \) only to the elements of \( Y \) assigns \( \text{true} \) also to \( \theta \). In the example, the set \( \{s_1, s_3\} \) satisfies \( \theta \); but this set is not unique.

Alternating automata [CKS81] combine existential choice of nondeterministic finite automata (i.e. disjunction) with the universal choice (i.e. conjunction) of \( \forall \)-automata [MP87] (where a run of the automaton says that from a state the automaton must move to all the next states given by the transition function, making a copy of itself for each next state). For example the transition \( \rho(s, a) = \{s_1, s_2, s_3\} \) of a NFA (which takes a state \( s \) and a symbol \( a \) and returns a set of states to which the automaton can move by reading the symbol \( a \) in the state \( s \)) can be equivalently viewed as the Boolean formula \( \theta = s_1 \lor s_2 \lor s_3 \). For a \( \forall \)-automaton the same transition is encoded by the \( \theta = s_1 \land s_2 \land s_3 \).

**Definition 4.3.1 (alternating automata)** An alternating Büchi automaton [MSS88] is a tuple \( A = (S, \Sigma, s_0, \rho, F) \), where \( S \) is a finite nonempty set of states, \( \Sigma \) is a finite nonempty alphabet, \( s_0 \in S \) is the initial state, and \( F \subseteq S \) is the set of accepting states. The automaton can move from one state when it reads a symbol from \( \Sigma \) according to the transition function \( \rho : S \times \Sigma \rightarrow B^+(S) \).

For example \( \rho(s_0, a) = (s_1 \lor s_2) \land (s_3 \lor s_4) \) means that the automaton moves from \( s_0 \) when reading \( a \) to state \( s_1 \) or \( s_2 \) and at the same time to state \( s_3 \) or \( s_4 \). Intuitively the automaton chooses for each transition \( \rho(s, a) = \theta \) one set \( S' \subseteq S \) which satisfies \( \theta \) and spawns a copy of itself for each state \( s_i \in S' \) which should test the acceptance of the remaining word from that state \( s_i \). Because the alternating automaton moves to all the states of a (nondeterministically chosen) satisfying set of \( \theta \), a run of the automaton is a tree of states.
Definition 4.3.2 (runs of alternating automata) Formally, a run of the alternating automaton on an input word \( \alpha = a_0, a_1, \ldots \) is an \( S \)-labeled tree \((T, V)\) (i.e. the nodes of the tree are labeled by state names of the automaton) such that \( V(\varepsilon) = s_0 \) and the following hold:

for a node \( x \) with \(|x| = i \) s.t. \( V(x) = s \) and \( \rho(s, a_i) = \theta \) then \( x \) has \( k \) children \( \{x_1, \ldots, x_k\} \) which is the number of states in the chosen satisfying set of states of \( \theta \), say \( \{s_1, \ldots, s_k\} \), and the children are labeled by the states in the satisfying set; i.e. \( \{V(x_1) = s_1, \ldots, V(x_k) = s_k\} \).

For example, if \( \rho(s_0, a) = (s_1 \lor s_2) \land (s_3 \lor s_4) \) then the nodes of the run tree at the first level have one label among \( s_1 \) or \( s_2 \) and one label among \( s_3 \) or \( s_4 \). When \( \rho(V(x), a) = \text{true} \), then \( x \) need not have any children; i.e. the branch reaching \( x \) is finite and ends in \( x \). A run tree of an alternating Büchi automaton is accepting if every infinite branch of the tree includes infinitely many nodes labeled by accepting states of \( F \). Note that the run tree may also have finite branches in the cases when the transition function returns \( \text{true} \).

Complementation of alternating automata is straight forward. It involves constructing a dual of a Boolean formula \( \theta \) of \( B^+(S) \). The dual \( \overline{\theta} \) is obtained from \( \theta \) by switching \( \lor \) and \( \land \), and by switching \( \text{true} \) and \( \text{false} \). For an automaton \( A = (S, \Sigma, s_0, \rho, F) \) we define the negated automaton \( \overline{A} = (S, \Sigma, s_0, \overline{\rho}, F) \) where \( \overline{\rho}(s, a) = \overline{\rho(s, a)} \). The language accepted by \( \overline{A} \) is the complement of the language accepted by \( A \) [CKS81].

Fischer-Ladner closure for \( CL \): For constructing the alternating automaton for a \( CL \) expression we need the Fischer-Ladner closure [FL77] for our \( CL \) logic. We follow the presentation in [HKT00] and use similar terminology. We define a function \( FL : CL \rightarrow 2^{CL} \) which for each expression \( C \) of the logic \( CL \) returns the set of its subexpressions. \( FL \) was introduced for process logics in order to deal with the compound actions. For avoiding circularity in the definition of \( FL \) an auxiliary function is needed \( FL^{\square} : \{[\beta]C \mid \beta \text{ an action}\} \rightarrow 2^{CL} \) (see [HKT00]). The functions \( FL \) and \( FL^{\square} \) are defined inductively in Table 4.3.

The following result gives the dimension (i.e. cardinality) of the Fischer-Ladner closure \( FL(\cdot) \) in terms of the dimension of a formula. It proves to be also linear as in the case of propositional dynamic logic [HKT00]. Naturally, the dimension of a formula (denoted \(|C|\)) is the number of symbols it contains; e.g. for an action \(|a + b| = 3\) where \( a, b \in A_B \), and for a formula \(|O_C(a)| = |a| + |C|\). We use the same notation for the dimension of the closure \(|FL(C)|\).

Theorem 4.3.3 (dimension of the Fischer-Ladner closure)

a. For any \( CL \) formula \( C \) is the case that \(|FL(C)| \leq |C|\).

b. For any \( CL \) formula \([\beta]C \) is the case that \(|FL^{\square}([\beta]C)| \leq |\beta|\).

Proof: The proof is by using simultaneous induction on the structure of the formula. The proof of \( b. \) is the same as in PDL [HKT00]. For the proof of \( a. \) we need to deal with the special constructions that \( CL \) introduces. We will not treat the Boolean operators \( \land, \lor, \text{or} \Theta \).

The basic case for \( \top \) and \( \bot \) is clear:

\(|FL(\bot)| = 1 = |\bot|\)
\( FL(\top) \triangleq \{ \top \} \)
\( FL(\bot) \triangleq \{ \bot \} \)
\( FL(P(\alpha)) \triangleq \{ P(\alpha) \} \)
\( FL(C_1 \land C_2) \triangleq \{ C_1 \land C_2 \} \cup FL(C_1) \cup FL(C_2) \)
\( FL(C_1 \lor C_2) \triangleq \{ C_1 \lor C_2 \} \cup FL(C_1) \cup FL(C_2) \)
\( FL(C_1 \oplus C_2) \triangleq \{ C_1 \oplus C_2 \} \cup FL(C_1) \cup FL(C_2) \)
\( FL([\beta]C) \triangleq FL^\Box([\beta]C) \cup FL(C) \)
\( FL^\Box([\beta]C) \triangleq \{ [\beta]C \} \)
\( FL^\Box([\beta \cdot \beta']C) \triangleq \{ [\beta \cdot \beta']C \} \cup FL^\Box([\beta]C) \cup FL^\Box([\beta']C) \)
\( FL^\Box([\beta + \beta']C) \triangleq \{ [\beta + \beta']C \} \cup FL^\Box([\beta]C) \cup FL^\Box([\beta']C) \)
\( FL^\Box([\beta^*]C) \triangleq \{ [\beta^*]C \} \cup FL^\Box([\beta]C) \)
\( FL^\Box([C_1 ?]C_2) \triangleq \{ [C_1 ?]C_2 \} \cup FL(C_1) \)
\( FL(O_C(\alpha_x)) \triangleq \{ O_C(\alpha_x) \} \cup FL(C) \)
\( FL(O_C(\alpha \cdot \alpha')) \triangleq \{ O_C(\alpha \cdot \alpha') \} \cup FL(O_C(\alpha)) \cup FL([\alpha]O_C(\alpha')) \)
\( FL(O_C(\alpha + \alpha')) \triangleq \{ O_C(\alpha + \alpha') \} \cup FL(O_\perp(\alpha)) \cup FL(O_\perp(\alpha')) \cup FL([\alpha + \alpha']C) \)
\( FL(P(\alpha)) \triangleq \{ P(\alpha) \} \)
\( FL(F_C(\alpha_x)) \triangleq \{ F_C(\alpha_x) \} \cup FL(C) \)
\( FL(F_C(\alpha \cdot \alpha')) \triangleq \{ F_C(\alpha \cdot \alpha') \} \cup FL(F_\perp(\alpha)) \cup FL(F_C(\alpha')) \)
\( FL(F_C(\alpha + \alpha')) \triangleq \{ F_C(\alpha + \alpha') \} \cup FL(F_C(\alpha)) \cup FL(F_C(\alpha')) \)
\( FL([\rightarrow]C) \triangleq FL^\Box([\rightarrow]C) \cup FL(C) \)
\( FL^\Box([\rightarrow]C) \triangleq \{ [\rightarrow]C \} \)
\( FL^\Box([\leftarrow \cdot \leftarrow]C) \triangleq \{ [\leftarrow \cdot \leftarrow]C \} \cup FL^\Box([\leftarrow]C) \cup FL^\Box([\leftarrow]C) \)
\( FL^\Box([\leftarrow + \leftarrow]C) \triangleq \{ [\leftarrow + \leftarrow]C \} \cup FL^\Box([\leftarrow]C) \cup FL^\Box([\leftarrow]C) \)

Table 4.3: Computing the Fisher-Ladner Closure
From [HKT00] we have:

\[
|FL([\beta]\mathcal{C})| \leq FL^\square([\beta]\mathcal{C}) + |FL(\mathcal{C})|
\]
\[
\leq |\beta| + |\mathcal{C}| \text{ by induction hypothesis a and b}
\]
\[
= |[\beta]\mathcal{C}|.
\]

The proof for the other \(CL\) constructs is particular to our logic but it is similar to what is done for PDL in [HKT00].

\[
|FL(OC(\alpha_\times))| \leq 1 + |FL(O\mathcal{C}(\alpha))|
\]
\[
\leq 1 + |\alpha| + |\mathcal{C}| |\alpha'| \text{ by induction hypothesis a}
\]
\[
= |OC(\alpha_\times)|.
\]

Note that the reparation \(\mathcal{C}\) is considered only once as when making the union of \(FL(O\mathcal{C}(\alpha)) \cup FL(O\mathcal{C}(\alpha'))\) the elements of \(FL(\mathcal{C})\) will appear only once. In general the subformulas of the reparation \(\mathcal{C}\) will appear only once no matter how we have to decompose the obligations.

\[
|FL(O\mathcal{C}(\alpha + \alpha'))| \leq 1 + |FL(O\mathcal{C}(\alpha))| + |FL(O\mathcal{C}(\alpha'))| + |FL(\mathcal{C})|
\]
\[
\leq 1 + |\alpha| + |\mathcal{C}| + |\alpha'| \text{ by induction hypothesis a}
\]
\[
= |OC(\alpha + \alpha')|.
\]

The proof for \(FL(P(\alpha))\) and \(FL(F\mathcal{C}(\alpha_\times))\) are similar.

\[
|FL(F\mathcal{C}(\alpha_\times\cdot\alpha'))| \leq 1 + |FL(F\mathcal{C}(\alpha))| + |FL(F\mathcal{C}(\alpha'))|
\]
\[
\leq 1 + |\alpha| + |\alpha'| + |\mathcal{C}| \text{ by induction hypothesis a}
\]
\[
= |FC(\alpha_\times\cdot\alpha')|.
\]

\[
|FL(F\mathcal{C}(\alpha + \alpha'))| \leq 1 + |FL(F\mathcal{C}(\alpha))| + |FL(F\mathcal{C}(\alpha'))|
\]
\[
\leq 1 + |\alpha| + |\alpha'| + |\mathcal{C}| \text{ by induction hypothesis a}
\]
\[
= |FC(\alpha + \alpha')|.
\]

This ends the proof. \(\square\)

**Theorem 4.3.4 (automaton construction)** Given a \(CL\) expression \(\mathcal{C}\), one can build an alternating Büchi automaton \(A^N(\mathcal{C})\) which will accept all and only the traces \(\sigma\) respecting the contract expression.

**Proof:** Take an expression \(\mathcal{C}\) of \(CL\), we construct the alternating Büchi automaton \(A^N(\mathcal{C}) = (S, \Sigma, s_0, \rho, F)\) as follows. The alphabet \(\Sigma = \mathcal{A}^\times_B\) consists of the finite set of concurrent actions; that is basic actions \(a, b, \ldots\) and actions composed only by using the concurrent composition operator, like \(a \times b\). Therefore the automaton accepts traces as defined in Section 4.2. The set
\[\rho(\bot, \gamma) \triangleq \text{false} \quad \rho(T, \gamma) \triangleq \text{true} \quad \rho(P(\alpha), \gamma) \triangleq \text{true}\]

\[\rho(C_1 \land C_2, \gamma) \triangleq \rho(C_1, \gamma) \land \rho(C_2, \gamma)\]

\[\rho(C_1 \lor C_2, \gamma) \triangleq \rho(C_1, \gamma) \lor \rho(C_2, \gamma)\]

\[\rho(C_1 \lor C_2, \gamma) \triangleq (\rho(C_1, \gamma) \land \rho(C_2, \gamma)) \lor (\rho(C_1, \gamma) \land \rho(C_2, \gamma))\]

\[\rho(O_C(\alpha), \gamma) \triangleq \text{if } \alpha_x \subseteq \gamma \text{ then true else } C\]

\[\rho(O_C(\alpha \cdot \alpha'), \gamma) \triangleq \rho(O_C(\alpha), \gamma) \land \rho([\alpha]O_C(\alpha'), \gamma)\]

\[\rho(O_C(\alpha + \alpha'), \gamma) \triangleq \rho(O_{\bot}(\alpha), \gamma) \lor \rho(O_{\bot}(\alpha'), \gamma) \lor C\]

\[\rho(F_C(\alpha), \gamma) \triangleq \text{if } \alpha_x \not\subseteq \gamma \text{ then true else } C\]

\[\rho(F_C(\alpha \cdot \alpha'), \gamma) \triangleq \rho(F_{\bot}(\alpha), \gamma) \lor F_C(\alpha')\]

\[\rho(F_C(\alpha + \alpha'), \gamma) \triangleq \rho(F_C(\alpha), \gamma) \land \rho(F_C(\alpha'), \gamma)\]

\[\rho([\alpha_x]C, \gamma) \triangleq \text{if } \alpha_x \not\subseteq \gamma \text{ then C else true}\]

\[\rho([\beta \cdot \beta']C, \gamma) \triangleq \rho([\beta][\beta']C, \gamma)\]

\[\rho([\beta + \beta']C, \gamma) \triangleq \rho([\beta]C, \gamma) \land \rho([\beta']C, \gamma)\]

\[\rho([\beta']C, \gamma) \triangleq \rho(C, \gamma) \land \rho([\beta][\beta']C, \gamma)\]

\[\rho([C_1?]C_2, \gamma) \triangleq \rho(C_1, \gamma) \lor (\rho(C_1, \gamma) \land \rho(C_2, \gamma))\]

Table 4.4: Transition Function of Alternating Büchi Automaton

of states \(S = FL(C) \cup FL(\overline{C})\) contains the subexpressions of the start expression \(C\) and their negations, where \(FL(\overline{C}) \triangleq \{\neg C' \mid C' \in FL(C)\}\). Recall that in \(CL\) the negation \(-C\) is \(C \rightarrow \bot\). The initial state \(s_0\) is the expression \(C\) itself. The set of final states \(F\) contains all the expressions of the type \([\beta^*]C\).

The transition function \(\rho : S \times A_B^c \rightarrow B^+(S)\) is defined in Table 4.4 and is based on the dualizing construction we have seen before, only that the dual of a state \(\overline{C}\) is the state \([C']\bot\) containing the negation of the expression. It is easy to see that if a run tree has an infinite path then this path goes infinitely often through a state of the form \([\beta^*]C\), thus explaining the \(F\) set. By looking at the definition of \(\rho\) we see that the expression \([\beta^*]C\) is the only expression which requires repeated evaluation of itself at a later point in the run. This causes the infinite unwinding in the run tree.

The rest of the proof shows the correctness of the automaton construction.

**Soundness:** Given an accepting run tree \((T, \mathcal{V})\) of \(A_N(C)\) over a trace \(\sigma\) we prove that \(\forall x \in T\) a node of the run tree with depth \(|x| = i, i > 0\), labeled by \(\mathcal{V}(x) = C_x\) a state of the automaton represented by a formula \(C_x \in FL(C) \cup FL(\overline{C})\), it is the case that \(\sigma(i..) \models C_x\). This implies that also \(\sigma(0..) \models \mathcal{V}(\epsilon) = C\), which means that if the automaton \(A_N(C)\) accepts a trace \(\sigma\) then the trace respects the initial contract \(C\).

We use induction on the structure of the formula \(C_x\). A formula \(C'\) is said to be a subformula of \(C_x\) iff \(C' \in FL(C_x)\). The induction method says that we have to prove the property (i.e. the soundness property) for the formula \(C_x\) by having as hypothesis that the property holds for all subformulas of \(C\).
For the truth formula $T$ it is trivial as $T$ is respected by any trace and thus by $\sigma(i..)$. For the other nonrecursive formulas the proof is simple by looking at the definition of the respecting relation $|=\ $between traces and formulas, and at the construction of the transition relation $\rho$ of the automaton. We take a case for each formula construction:

a. if $C_x = C' \land C''$ and we are at depth $|x| = i$ it means that the transition relation is $\rho(C' \land C'', \sigma(i)) = (\rho(C', \sigma(i)) \land \rho(C'', \sigma(i)))$. We should understand the transition relation as follows: because we are in an accepting run tree on $\sigma$ it means that the automaton from state $C' \land C''$ accepts $\sigma(i)$ iff the automaton accepts $\sigma(i)$ from both states $C'$ and $C''$. We can apply the induction hypothesis on the subformulas $C'$ and $C''$ because we know now that there is an accepting run from states $C'$ and $C''$ on the remaining trace $\sigma(i..)$. This means that we get both $\sigma(i..) \models C'$ and $\sigma(i..) \models C''$ which by the semantics it means that $\sigma(i..) \models C' \land C''$; i.e. the conclusion.

b. for $C_x = C' \lor C''$ proof is similar as for $\land$.

c. if $C_x = C' \oplus C''$ the proof follows the same arguments as before. Consider we are at depth $|x| = i$ it means that the transition relation is $\rho(C' \oplus C'', \sigma(i)) = (\rho(C', \sigma(i)) \land \rho(C'', \sigma(i))) \lor ((\rho(C', \sigma(i)) \land \rho(C'', \sigma(i)))$). The intuition for the transition relation is clear by now; if we know that the automaton accepts $\sigma(i)$ from state $C' \oplus C''$ then we know that the automaton accepts $\sigma(i)$ from state $C'$ but it does not accept $\sigma(i)$ from state $C''$ (or the other disjunction branch). By inductive reasoning we are ensured that $C'$ holds in $\sigma$ from point $i$ on, and we are ensured that $C''$ fails from this point on. By the semantics of $\oplus$ we get that $\sigma(i..) \models C' \oplus C''$. We make the same reasoning for the other disjunction choice.

d. if $C_x = O_{C'}(\alpha_x)$ and we are at depth $|x| = i$ it means that the transition relation can be of two kinds: first $\rho(O_{C'}(\alpha_x), \sigma(i)) = \text{true}$ which, because the run is accepting, it means that $\alpha_x \subseteq \sigma(i)$ or $\alpha_x = \sigma(i)$ which from the definition of the respecting relation we conclude that $\sigma(i..) \models O_{C'}(\alpha_x)$. The transition relation can also be $\rho(O_{C'}(\alpha_x), \sigma(i)) = C'$ which because the run is accepting and from the induction hypothesis we conclude that $\sigma(i+1..) \models C'$. By following the semantic definition we conclude that $\sigma(i..) \models O_{C'}(\alpha_x)$.

e. if $C_x = O_{C'}(\alpha \cdot \alpha')$ because the run is accepting the rest of the trace $\sigma(i..)$, then by the transition relation it means that the automaton also accepts $\sigma(i..)$ from both the state $O_{C'}(\alpha)$ and from the state $[\alpha]O_{C'}(\alpha')$. By the semantics we have that $\sigma(i..) \models O_{C'}(\alpha \cdot \alpha')$.

f. All remaining cases are similar with the exception of the recursion formula $[\beta^*]C$. The transition relation says that the automaton must accept $\sigma(i..)$ from state $C$ and also from the state $[\beta][\beta^*]C$. Note that the accepting run $(T, \mathcal{V})$ over $\sigma(i..)$ has finite branching whenever $\beta \not\subseteq \sigma(i)$. If $(T, \mathcal{V})$ has an infinite branch than this branch contains $[\beta^*]C$ infinitely many times. Because of this and the induction hypothesis we have that $\sigma(i..) \models [\beta][\beta^*]C$ and from the transition relation and the induction hypothesis again we have that $\sigma(i..) \models C$. Now from the semantics of $[\beta^*]C$ we finish the proof $\sigma(i..) \models [\beta^*]C$.

Completeness: Given a trace $\sigma$ s.t. $\sigma \models C$ we prove that the constructed automaton $A^N(C)$ accepts $\sigma$ (i.e. there exists a run tree $(T, \mathcal{V})$ of $A^N(C)$ over the trace $\sigma$).
The proof proceeds by constructing a run tree of $A^N(C)$ which maintains the following invariant: $\forall x \in T$ with $V(x) = C_x$ then $\sigma(|x|.) \models C_x$. The run has to start in the initial state of the automaton so $V(\varepsilon) = C$ and since we know from the hypothesis that $\sigma \models C$ the invariant is satisfied for $\varepsilon$. It is easy to see by the semantics of $\mathcal{CL}$ and by following the definition of $\rho$ over the composing subformulas of $C$ that the run can always proceed s.t. for a node $x$ for which the invariant holds the successor nodes all satisfy the invariant.

We give a few examples: consider the formulas for which the transition function $\rho$ takes value $true$; like $\rho(OC(\alpha_\leq), \sigma(0))$ with $\alpha_\leq \subseteq \sigma(0)$. It is clear from the semantics that $\sigma \models O_C(\alpha_\leq)$ and the run tree is accepting since it goes into a $true$ transition. For more complicated formulas like $O_C(\alpha' \cdot \alpha'')$ because of the semantics we have that $\sigma \models O_C(\alpha' \cdot \alpha'')$ implies $\sigma \models O_C(\alpha')$ and $\sigma \models [\alpha']O_C(\alpha'')$. By semantics again, the second relation gives $\sigma(1..) \models O_C(\alpha'')$. The run tree must proceed according to the $\rho$ function and thus it can advance one step into the tree by two means: either by respecting the first obligation and thus one of the successors of $\varepsilon$ must be labeled by state $O_C(\alpha'')$, but this satisfies the invariant. The other is by not satisfying the obligation and thus ending up in a state labeled by the reparation $C$. □

**Example 4.1.1 as alternating automata:** We shall now briefly show how for the $\mathcal{CL}$ expression $C = [e]O_{\perp(p\cdot p)}(p + d \times n)$ of page 92 we construct an alternating automaton which accepts all the traces (like the ones we have seen on page 96) that satisfy $C$ and none others. The Fischer-Ladner closure of $C$ generates the following set of subexpressions:

\[
FL(C) = \{C, O_{\perp(p\cdot p)}(p + d \times n), O_{\perp}(p), \bot, O_{\perp}(d \times n), O_{\perp}(p \cdot p), [p]O_{\perp}(p)\}
\]

The set $A^*_B$ of concurrent actions is the set $\{e, p, n, d\}^\times$ of basic actions closed under the constructor $\times$. The alternating automaton is:

\[
A^N(C) = (FL(C) \cup FL(C), \{e, p, n, d\}^\times, C, \rho, \emptyset)
\]

Note that there is no expression of the form $[\beta^n]C$ in $FL$ because we have no recursion in our original contract clause from Example 4.1.1, therefore the set of final states is empty. This means that the automaton is accepting all run trees which end in a state where the transition function returns $true$ on the input symbol.\(^3\)

The transition function $\rho$ is defined in table below where $C_1 = O_{\perp(p\cdot p)}(p + d \times n)$:

<table>
<thead>
<tr>
<th>$\rho(state, action)$</th>
<th>$e$</th>
<th>$p$</th>
<th>$d$</th>
<th>$e \times d$</th>
<th>$e \times p$</th>
<th>$d \times n$</th>
<th>$e \times d \times n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>$C_1$</td>
<td>$true$</td>
<td>$true$</td>
<td>$C_1$</td>
<td>$true$</td>
<td>$C_1$</td>
<td>$true$</td>
</tr>
<tr>
<td>$C_1$</td>
<td>$O_{\perp(p\cdot p)}$</td>
<td>$true$</td>
<td>$O_{\perp(p\cdot p)}$</td>
<td>$true$</td>
<td>$true$</td>
<td>$true$</td>
<td></td>
</tr>
<tr>
<td>$O_{\perp}(p)$</td>
<td>$\bot$</td>
<td>$true$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$true$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$O_{\perp}(d \times n)$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$true$</td>
<td>$true$</td>
<td></td>
</tr>
<tr>
<td>$O_{\perp}(p \cdot p)$</td>
<td>$\bot$</td>
<td>$O_{\perp}(p)$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$O_{\perp}(p)$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$[p]O_{\perp}(p)$</td>
<td>$true$</td>
<td>$O_{\perp}(p)$</td>
<td>$true$</td>
<td>$true$</td>
<td>$O_{\perp}(p)$</td>
<td>$true$</td>
<td>$true$</td>
</tr>
</tbody>
</table>

Computing the values in the table above is easy; e.g.:  

\(^3\)Note that for this particular example we do not see the power of alternating automata. More, the alternating Büchi automata behaves like a NFA.
Run-time Monitoring of Electronic Contracts

Because from the state $\bot$ nothing can be accepted (as it generates only false) we have written in the table only $O \bot (p \cdot p)$. There are $2^4$ labels in the alphabet of $A^N(C)$ and we have exemplified only some of the more interesting ones. Moreover, none of the states from $FL$ (i.e. $[C_1?] \bot$, the complemented expressions) are reachable nor do they contribute to the computation of any transition to a reachable state (like e.g. $O \bot (d \times n)$ contributes to the computation of $\rho(C_1, e)$), so we have not included them in the table. The line for state $\bot$ is omitted as it generates only false.

In Fig. 4.1 we picture all the reachable states of the automaton $A^N(C)$. For brevity, we have not represented all the transitions; e.g. from state $C$ to state $C_1$ there should be a transition for each label which includes $e$ (like $e \times d, e \times p$, or $e \times d \times n$).

For example, the automaton accepts the trace $e, e \times d, p, p$, because starting from the state $C$, the state $C_1$ is reached after $e$, then the state labelled $O \bot (p \cdot p)$ after $e \times d$, then $O \bot (p)$ after $p$. From this state, the transition function results in true on the symbol $p$, and the automaton accepts the trace.

Conversely, the automaton rejects the trace $e, e \times d, p, e \times d$, since, from the state labelled $O \bot (p)$ the state labelled $\bot$ is reached while reading $e \times d$. From then on the transition function will never result in true and the Büchi acceptance set is empty, which means that any infinite extension of the run tree will not be accepted either.

4.3.2 Constructing the monitor

We use the method of [BLS06] and we consequently use a 3-valued semantics approach to run-time monitoring. The monitor will generate a sequence of observations, denoted $[\sigma \models C]$, for a finite trace $\sigma$ by:

$$[\sigma \models C] = \begin{cases} 
\text{tt} & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma \sigma' \models C \\
\text{ff} & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma \sigma' \not\models C \\
? & \text{otherwise}
\end{cases}$$

The method of [BLS06] uses the $NBA(C)$ together with the automaton for the negated expression $NBA(\neg C)$, and returns a Moore machine associated with an expression $C$ which for each state it outputs a symbol of $\{\text{tt, ff, ?}\}$ if what it has seen until that state respectively respects the expression $C$, violates $C$, or it does not know yet.
We can obtain a nondeterministic Büchi automaton \(NBA(C)\) from our alternating automaton \(A^N(C)\) s.t. both automata accept the same trace language. The method is standard [Var95] and it constructs an automaton exponentially larger than the input automaton \(A^N(C)\), therefore \(NBA(C)\) is exponential in the size of the expression.

The method of [BLS06] is the following: take the \(NBA(C)\) for which we know that \([\sigma \models C] \neq \mathsf{ff}\) if there exists a state reachable by reading \(\sigma\) and from where the language accepted by \(NBA(C)\) is not empty. Similarly for \([\sigma \models C] \neq \mathsf{tt}\) when taking the complement of \(NBA(C)\) (or equivalently we can take the \(NBA(\neg C)\) of the negated formula which is \([C'?{\bot}]\)). Construct a function \(F : S \rightarrow \{\top, \bot\}\) which for each state \(s\) of the \(NBA(C)\) returns \(\top\) iff \(\mathcal{L}(NBA(C), s) \neq \emptyset\) (i.e. the language accepted by \(NBA(C)\) from state \(s\) is not empty), and \(\bot\) otherwise. Using \(F\) one can construct a nondeterministic finite automaton \(NFA(C)\) accepting finite traces s.t. \(\sigma \in \mathcal{L}(NFA(C))\) iff \([\sigma \models C] \neq \mathsf{ff}\). This is the same NBA only that the set of final states contains all the states mapped by \(F\) to \(\top\). Similarly construct a \(NFA(\neg C)\) from \(NBA(\neg C)\). One uses classical techniques to determinize the two NFAs. Using the two obtained DFAs one constructs the monitor as a finite state machine which at each state outputs \(\{\mathsf{tt}, \mathsf{ff}, ?\}\) if the input read until that state respectively satisfies the contract clause \(C\), violates it, or it cannot be decided. The monitor is the product of the two \(DFA(\mathcal{C})\) and \(DFA(\neg \mathcal{C})\).

The method of [BLS06] omits one important requirement that we need for monitoring electronic contracts. We need that the monitor can read (and move to a new state) each possible action from the input alphabet. When doing the product of the two DFAs, if one of them does not have a transition for one of the symbols then this is lost for the monitor too. The solution is simple; we add to each DFA a dummy state which is not accepting and which collects all the missing transitions (with the missing labels).

The whole method is proven correct; i.e. \([\sigma \models C] = \lambda(\rho(s_0, \sigma))\) the semantics of \(C\) on the finite trace \(\sigma\) (\([\sigma \models C]\)) is exactly the output of the Moore machine \((\lambda : S \rightarrow \{\mathsf{tt}, \mathsf{ff}, ?\}\) is the output function) from the state reached by reading \(\sigma\) from the starting state \(s_0\). The monitor generated is proven to have size double-exponential in the size of the expression; one exponent coming from the Büchi automaton and the other from determinizing the NBAs [BLS06]. It is known that minimization techniques give good results for such problems and on-the-fly generation of the state space and of the transition relation make the automata manageable. For \(C\mathcal{L}\) we get the same size of the final monitor even if we go in the beginning through alternating automata.

Example 4.1.1 as run-time monitor: Consider the alternating automaton \(A^N(C)\) constructed before. The resulting monitor for this automaton is sketched in Fig. 4.2. Consider the traces showed on page 96 for the expression \(\mathcal{C}\): for a respecting trace \(e, p\) the monitor outputs \(?, \mathsf{tt}\); for the rejecting trace \(e, e, e\) the monitor outputs \(?, ?, ?, \mathsf{ff}\).

### 4.4 Final Remarks and Open Problems

The work reported here may be viewed from different angles. On one hand we use alternating automata which has recently gained popularity [Var97, KVV00] in the temporal logics community. We apply these to a rather unconventional logic \(C\mathcal{L}\) [PS07a], a process logic (PDL [FL77]) extended with deontic logic modalities [vW51]. On another hand we presented the
formal language $\mathcal{CL}$ with a trace semantics, and showed how we specify electronic contracts using it. Though $\mathcal{CL}$ has been originally designed as a language for specifying electronic contracts in the context of service-oriented architectures, it has been argued that its use may be extremely useful in component-based development systems [OSS07]. Due to the contrary-to-duties and contrary-to-prohibitions, $\mathcal{CL}$ is also suitable to specify properties, and reason about, fault-tolerant systems, similar to what is presented in [CM07].

From a practical point of view we presented here a first fully automated method of extracting a run-time monitor for a contract formally specified using the $\mathcal{CL}$ logic.

Note that our main objective is not to enforce a contract, but only to monitor it, that is to observe that the contract is indeed satisfied. In our opinion monitoring contracts is more reasonable than enforcing them, since in our contract such agreements are supposed to be written between different parties who have already agreed on their content. In other words, a contract must also contain what are the actions to be performed in case of violation of certain clauses.

The trace semantics presented in this paper is intended for monitoring purposes, and not to explain the language $\mathcal{CL}$. Thus, from the trace semantics point of view $[\alpha_\nearrow]C$ is equivalent to $F_C(\alpha_\nearrow)$; whereas this is not the case in the branching semantics of $\mathcal{CL}$ from Section 3.2. Another particular issue concerns the $\oplus$ operator which needs some additional explanation. Our trace semantics interprets it as an exclusive or, but when combined with obligations we might get some counter-intuitive specifications. For instance, according to the trace semantics there is no model for $O(a) \oplus O(a)$ though one may expect to get $O(a)$ instead under an interpretation of $\oplus$ as a choice operator. Moreover, the trace semantics will exclude traces starting with $a \times b$ and those starting with $a, a$ and $b, b$ for the $\mathcal{CL}$ expression $\sigma = O_{O(b)}(a) \oplus O_{O(a)}(b)$.

4.4.1 Relating the full and the trace semantics - open problem

The contents of this section are labeled as open problem because many of them do not have full proofs, hence are just conjectures. One may like to further investigate the statements that have no proof.

A folk technique called linearization takes (in our case) a pointed normative structure and returns all the (in)finite traces that start in the designated state $i$ of the pointed structure. Denote
Proposition 4.4.1 For any trace $\sigma$ we can find a normative structure $K^N$ and a state $i$ s.t. $\sigma \in \parallel K^N, i\parallel$.

Some of the following results hold for a restricted syntax of $\mathcal{CL}$ which is given in Table 4.5. These syntactic restrictions are enough for doing runtime monitoring. The more general syntax we had in Section 4.2 contained general negation of clauses and tests inside the dynamic modalities. These are not realistic for doing runtime monitoring.

The following lemma intuitively says that if a formula is satisfied in a pointed normative structure then all the traces that start from that point satisfy the same formula in the trace semantics. Basically this gives the conformance of the trace semantics with the full semantics of $\mathcal{CL}$.

Lemma 4.4.2 For a formula $C$ built using the restricted syntax of Table 4.5 we have:

\[
\text{for arbitrary } K^N \text{ and } i \in K^N, \text{ if } K^N, i \models C \text{ then } \forall \sigma \in \parallel K^N, i\parallel : \sigma \models C.
\]

Proof: The proof uses structural induction and for some cases also the proof principle reductio ad absurdum.

Take $C$ to be $O_{c_1}(\alpha_x)$. Suppose that $\exists \sigma \in \parallel K^N, i\parallel \text{ s.t. } \sigma \not\models O_{c_1}(\alpha_x)$. By the trace semantics this means that $\sigma = \sigma(0)\sigma'$ and $\alpha_x \not\subseteq \sigma(0)$ and $\sigma' \not\models C_1$. Then, $\sigma(0) \in K^N, w \models \alpha_x$ and by the branching semantics we have that $K^N, w \models C_1$ where $w$ is the state reached by $\sigma(0)$. Moreover, $\sigma' \in \parallel K^N, w\parallel$. By the inductive hypothesis we know that $\forall \sigma' \in \parallel K^N, w\parallel$ then $\sigma' \models C_1$ which is a contradiction with the initial supposition.

Take $C$ to be $O_{c_1}(\alpha_1 + \alpha_2)$. From the branching semantics of $K^N, i \models O_{c_1}(\alpha_1 + \alpha_2)$ we know that the normative structure $K^N$ simulates the tree $A^D(\alpha_1 + \alpha_2)$ from state $i$. This means that all the traces in the maximal simulating structure include a trace in the action $\alpha_1 + \alpha_2$. Moreover, at the end state of all the traces in the non-simulating reminder structure the reparation $C_1$ holds. We know that any infinite trace of $K^N$ starts with a trace contained in one of these two structures. If $\sigma$ starts with a trace which is part of the maximal simulating structure then $\sigma \models O_{\bot}(\alpha_1)$ or $\sigma \models O_{\bot}(\alpha_2)$. Otherwise, if $\sigma$ starts with a trace contained in the non-simulating reminder structure then $\sigma \models [\alpha_1 + \alpha_2]C_1$. This is because the non-simulating reminder contains the negation action, and by the inductive hypothesis because $C_1$ holds at the end states it means that the remaining trace $\sigma' \models C_1$.

Take $C$ to be $O_{c_1}(\alpha_1 \cdot \alpha_2)$. The proof is similar to the case before.
Take $C$ to be $C_1 \land C_2$. From the branching semantics we have that $K^N, i \models C_1$ and $K^N, i \models C_2$. By the inductive hypothesis we get that $\forall \sigma \in \|K^N, i\|, \sigma \models C_1$ and the same $\sigma \models C_2$ which by the trace semantics it means that $\forall \sigma \in \|K^N, i\|, \sigma \models C_1 \land C_2$.

When $C$ is $C_1 \lor C_2$ the proof is similar as for the case before. Here it is enough to use the inductive hypothesis only once for one of the subformulas, say for $C_1$.

The proofs for when $C$ is $\bot$ or $\top$ are trivial.

The following corollary relates the satisfiability on normative structures with satisfiability only on traces. Intuitively, all the traces that come from the full semantics on normative structures are good traces in terms of the trace semantics; precisely, they are traces accepted by the trace semantics. Nevertheless, there may be more traces accepted by the trace semantics. This means that the trace semantics may be more loose, more liberal.

**Corollary 4.4.3**

$$\bigcup_{K^N, i = C} \|K^N, i\| \subseteq \{\sigma \mid \sigma \models C\}$$

The following lemma states the relation between the satisfiability in the trace semantics and the satisfiability in the branching semantics. It basically says that if a contract clause is respected by some trace of actions then the contract is satisfiable in the branching semantics.

**Lemma 4.4.4** With the restricted syntax of Table 4.5 we have:

$$\text{if } \exists \sigma \text{ s.t. } \sigma \models C \text{ then } \exists K^N, \exists i \in K^N \text{ s.t. } K^N, i \models C.$$  

**Proof:** The proof is based on Lemma 4.4.2 and uses the proof principle *reductio ad absurdum*.

Suppose that $\not \exists K^N, \not \exists i \text{ s.t. } K^N, i \models C$. This is the same as saying $\forall K^N, \forall i \text{ then } K^N, i \not \models C$. By the syntactic definition of the propositional $\neg$ operation we have $\forall K^N, \forall i \text{ then } K^N, i \models \neg C$. (This is the validity of the $\neg C$ formula.) From Lemma 4.4.2 we know that for a given normative structure and a state we have that $\forall \sigma \in \|K^N, i\| \text{ then } \sigma \models \neg C$. Together with a consequence of Proposition 4.4.1, i.e. $\forall \sigma \text{ then } \sigma \in \bigcup_{K^N, i} \|K^N, i\|$, we deduce that $\forall \sigma, \sigma \not \models \neg C$. This means that $\forall \sigma, \sigma \not \models C$. This is a contradiction with the hypothesis of the lemma and thus we conclude the proof.

**Lemma 4.4.5** With the restricted syntax of Table 4.5 we have:

$$\text{if } \sigma \models C \text{ then } \exists K^N, \exists i \in K^N \text{ s.t. } K^N, i \models C \text{ and } \sigma \in \|K^N, i\|.$$  

**Proof:** The proof of the lemma uses induction on the structure of the contract clause $C$.

- When $C = \top$ the proof is trivial. Any trace respects $C$; take an arbitrary trace $\sigma$. Any normative structure makes $\top$ true in any world. Therefore, just take an arbitrary normative structure $K^N$ and an arbitrary point $i$ such that $\sigma \in \|K^N, i\|$; this is clearly possible.

- Take $C$ to be $O_{c_1}(\alpha_x)$. The $\sigma$ which respects $C$ can be of two kinds:
a. $\sigma = \sigma(0)\sigma'$ where $\alpha_x \in \sigma(0)$ and $\sigma'$ is any trace. We construct the normative structure of Figure 4.3(i). Following the branching semantics it is clear that $O_\mathcal{C}(\alpha_x)$ holds at state $s_1$ because all the markers $o_a$ with $a \in \alpha_x$ corresponding to $\alpha_x$ mark the state $s_2$. Moreover, the trace $\sigma$ is part of the traces starting in state $s_1$. This is because we have labeled the first transition in the normative structure by $\sigma(0)$ the first element of $\sigma$, and from state $s_2$ any (infinite) trace is part of the normative structure. Note that we could have given a more simple structure which from $s_2$ accepts only the trace $\sigma'$ and not any trace.

b. $\sigma = \sigma(0)\sigma'$ where $\alpha_x \not\in \sigma(0)$ and $\sigma'$ respects $\mathcal{C}_1$. (Note that from Proposition ??? we know that these two cases cover all the possible traces.) From the inductive hypothesis we know that $\exists K^N_1$ s.t. $K^N_1, w \models \mathcal{C}_1$ and $\sigma' \in \|K^N_1, w\|$. Thus we construct the normative structure $K^N$ pictured in Figure 4.3(ii). One can easily see the trace $\sigma(0)\sigma'$ in $K^N$, starting with the transition $(s_1, \sigma(0), w)$ and then continuing with the trace $\sigma'$ which is part of $K^N_1$ starting in $w$. By the definition of the branching semantics $O_{\mathcal{C}_1}(\alpha_x)$ holds in $s_1$ because state $s_2$ is marked by all the necessary markers and all the transitions in the reminder structure (which is only one transition, i.e. $(s_1, \sigma(0), w)$) have the reparation $\mathcal{C}_1$ holding in the end state (i.e. $w$ in our case).

- Take $\mathcal{C}$ to be $F_{\mathcal{C}_1}(\alpha_x)$. Similar as before, the traces are of two kinds (as given in the trace semantics):

  a. $\sigma = \sigma(0)\sigma'$ where $\alpha_x \not\in \sigma(0)$ and $\sigma'$ is any trace. We construct the normative structure of Figure 4.3(iii). The transition $(s_1, \sigma(0), s_2)$ is labeled by the first element of the trace and the concurrent action $\alpha_x$ is not contained in this label. The trace $\sigma$ is clearly contained in $\|K^N, s_1\|$. By the branching semantics $K^N, s_1 \models F_{\mathcal{C}_1}(\alpha_x)$ because there is no transition in $K^N$ which simulates $\alpha_x$ and thus the partial simulation condition is trivially respected and the reparation $\mathcal{C}_1$ needs not be checked in any state. Note that there is no need of $\bullet$ markers because if an action is not present among the labels of the normative structure then this action is by default considered forbidden.

  b. $\sigma = \sigma(0)\sigma'$ where $\alpha_x \in \sigma(0)$ and $\sigma'$ respects $\mathcal{C}_1$. For this we construct the normative structure of Figure 4.3(iv). By the branching semantics $K^N, s_1 \models F_{\mathcal{C}_1}(\alpha_x)$ because $\alpha_x \in \sigma(0)$ and $K^N, s_2 \models \mathcal{C}_1$ by the inductive hypothesis. Moreover, state $s_2$ is marked by all markers corresponding to $\sigma(0)$. By the inductive hypothesis we know that $\sigma' \in \|K^N, s_2\|$
and thus $\sigma(0)\sigma' \in \|K^N, s_1\|$. 

\[ \square \]

**Corollary 4.4.6**

\[
\{ \sigma \mid \sigma \models C \} \subseteq \bigcup_{K^N, i \models C} \|K^N, i\|
\]

### 4.4.2 Related work

For run-time verification our use of alternating automata on infinite traces of actions is a rather new approach. This is combined with the method of [BLS06] that uses a three value (i.e. true, false, inconclusive) semantics view for run-time monitoring of LTL specifications. We know of the following two works that use alternating automata for run-time monitoring: in [FS04] LTL on infinite traces is used for specifications and alternating Büchi automata are constructed for LTL to recognize finite traces. The paper presents several algorithms which work on alternating automata to check for word inclusion. In [SB06] LTL has semantics on finite traces and nondeterministic alternating finite automata are used to recognize these traces. A determinization algorithm for alternating automata is given which can be extended to our alternating Büchi automata.

We have taken the approach of giving semantics to $\mathcal{C}L$ on infinite traces of actions which is closer to [FS04] but we want a deterministic finite state machine which at each state checks the finite input trace and outputs an answer telling if the contract has been violated. For this reason we found the method of [BLS06] most appealing. On the other hand a close look at the semantics of $\mathcal{C}L$ from Section 4.2 reveals the nice feature of this semantics which behaves the same for finite traces as for infinite traces. This coupled with the definition of alternating automata from Section 4.3.1 which accepts both infinite and finite traces gives the opportunity to investigate the use of alternating finite automata from [SB06] on the finite trace semantics. This may generate a monitor which is only single-exponential in size.
Chapter 5

Conclusions

By now we have presented an action-based dynamic deontic logic for representing and reasoning about electronic/legal contracts, that we called $\mathcal{CL}$ to stand for contract logic. The main objective has been to develop the theoretical foundations of $\mathcal{CL}$. The goal with $\mathcal{CL}$ was to have a logic (that is more than a formal language) for writing and talking about electronic contracts. The challenge for $\mathcal{CL}$ was to incorporate (in one single logic) natural notions and properties that were found scattered over several papers in the existing literature. The notions that $\mathcal{CL}$ incorporates have been discussed all over the thesis and have been summarized in the introduction chapter; the properties for contracts that $\mathcal{CL}$ respects have been presented as propositions (with proofs) in Chapter 3.

One final desire with this research was to show the practical applications of the logical theory of $\mathcal{CL}$. One such application was investigated and presented in Chapter 4 as a methodology for doing run-time monitoring of electronic contracts. This application is not trivial and uses powerful theoretical techniques on top of $\mathcal{CL}$. The practical applications of $\mathcal{CL}$ are multiple and some of them (the more theoretically inclined ones) are presented in the Section 5.1 of this concluding chapter.

During this search for the right $\mathcal{CL}$ language a very simple and powerful theory was developed in the form of synchronous Kleene algebra and its extension with tests. This algebraic theory is the technical basis of $\mathcal{CL}$ as it is necessary for defining the semantics of $\mathcal{CL}$. But synchronous Kleene algebra is interesting on its own as it is the extension of regular programs with concurrency in the simple form of synchrony. SKA is nicely behaved and has low decidability complexity.

In this respect we have first investigated at length the properties of the synchronous actions that are the heart of $\mathcal{CL}$. The resulting algebraic formalism is called synchronous Kleene algebra (with tests) and is interesting in itself, outside the $\mathcal{CL}$ language. For $\mathcal{CL}$ we have used the completeness result of the algebra and the representation results to give the semantics which depends on the structure of the actions. Outside $\mathcal{CL}$, Kleene algebras are used in relation with sequential programming languages and Hoare-style reasoning about programs. Hence, the synchronous Kleene algebra that we developed in Chapter 2 is the algebraic formalism behind programming languages that have a notion of concurrency that can be modelled as synchrony. As we argued in the introductory Section 2.1, the synchrony model, as introduced by R.Milner, is expressive enough to capture a great deal of applications.

From a theoretical point of view, the technical results of Chapter 2 are as follows:
a. We have introduced two algebraic formalisms for synchronous actions:

a.1. *Synchronous Kleene algebra* and

a.2. *Synchronous Kleene algebra with tests*.

b. For both of these formalisms we have presented standard models as, respectively, *sets of synchronous strings* and *sets of guarded synchronous strings*.

c. For both formalisms we have given representation theorems (in the style of Kleene’s representation theorem) in terms of automata that accept, respectively, *synchronous strings* and *guarded synchronous strings*.

d. We have proven completeness results for both formalisms.

e. Both formalisms are decidable and with low complexities.

f. We have isolated and investigated the class of \( \ast \)-free synchronous actions that are used inside the deontic modalities of \( \mathcal{CL} \).

g. For the \( \ast \)-free actions a number of theoretical results have been shown among which is the representation of these as some special rooted trees, which are used in the semantics of the deontic modalities of \( \mathcal{CL} \).

Based on this strong theoretical foundation for the synchronous actions we could define the semantics (and syntax) of the \( \mathcal{CL} \) logic in Chapter 3. This chapter represents our second main theoretical contribution. All the motivations for developing \( \mathcal{CL} \) as it is, have been given in the introductory Section 1.3. In Chapter 3 we give the formal theoretical definitions and results for \( \mathcal{CL} \). We define formally all the concepts that have been put forward as motivation and we formulate and prove the properties that Section 1.3 asked from the \( \mathcal{CL} \) logic.

More precisely, the theoretical discourse of Chapter 3 can be summarized as follows:

a. We have defined the syntax of \( \mathcal{CL} \) which includes:

a.1. *Deontic modalities* defined to include syntactically the reparations for the obligations and for the prohibitions. In this way, contrary-to-duties and contrary-to-prohibitions can be represented as first-class-citizens in \( \mathcal{CL} \).

a.2. The deontic modalities are applied exclusively over \( \ast \)-free synchronous actions (which we call deontic actions in the context of \( \mathcal{CL} \)).

a.3. *Dynamic modality*, adopted from the propositional dynamic logic, applied over general synchronous actions but where we allow only, what is called, *poor tests*.

a.4. This language is expressive enough to encode *temporal notions* like “always” and “eventually”, which we use in some of our examples.

b. We have defined a full semantics for \( \mathcal{CL} \) in terms of Kripke-like structures that we call *normative structures*. 
c. We have shown that $CL$ has the tree model property. We did this first for parts of $CL$, hence we have also separate tree model results for the deontic sublanguage of $CL$ and for the dynamic sublanguage (over synchronous actions). The tree model property is important in its own as many logics that have this property are amenable to, what is known as, the automata theoretic approach to logic [Var96].

d. We have shown decidability results for $CL$ and sublanguages, based on the tree model property.

e. We have proven quite a number of properties for $CL$, some of which have been mentioned in the motivation section. We have shown both validities and non-validities for $CL$.

f. We have shown why $CL$ avoids some the classic paradoxes of standard deontic logic.

After all these theoretical results for $CL$ we can be confident in our logical formalism for representing contracts. In Chapter 4 we show how $CL$ can be used to represent an example from a legal contract that we used in another paper as a case study for doing model checking of contracts written in $CL$ [PPS07]. Actually we show how the theory of $CL$ can be used in practice to monitor at run-time that the contract specification is not violated.

More precisely, Chapter 4 includes:

a. A trace semantics for $CL$ to be used for run-time monitoring. This is less powerful semantics than the full semantics based on normative structures that we presented in Chapter 3.

b. We have shown how to use the theory of alternating automata (which involved a Fischer-Ladner construction) to resolve the satisfiability problem for $CL$ w.r.t. the trace semantics.

c. We have shown how to construct, from a corresponding alternating automaton, for a $CL$ clause a deterministic finite machine which acts as a monitor. That means that the Moore machine that we build listens and signals as soon as a violation of the contract appears.

d. All these sum up to an automated technique for generating a monitor machine from a $CL$ contract clause.

## 5.1 Research directions and open problems

In this last section I want to discuss possible future research directions that could be taken starting from the work that I have just presented. Each chapter contains a last subsection with open problems stemming from the work presented there. Here I plan to give more general views and some other challenges that have not been mentioned until now. I also recall some problems.

Concerning verification of contracts through model checking or proof theory techniques based on $CL$, an encoding of a variant of the logic presented here into NuSMV has been presented by the author in [PPS07]. With the development of the explicit semantics for $CL$ based on Kripke-like structures from Chapter 3, we are now in the position of developing a specific model checker for $CL$.

Besides the development of a model checker, future work may include the development of a proof system. The proof system for $CL$ may take either the form of a Hilbert-style axiomatic
system or a Tableaux-like proof system. Such a proof system may give new insights into developing a *compositional semantics* for $\mathcal{CL}$. This compositional semantics may be defined in a small-step style (like we defined the simulation relation between normative structures and trees or like we defined the negation operation on actions). By small-step semantics I mean defining the $\mathcal{CL}$ semantics not by looking at all the tree of the action but by looking at small actions first and then continuing to the rest of the action, similar to what is done for PDL.

Based on a proof system, or a model checking technique for that matter, we can easily develop techniques for *negotiation of contracts*. At a first glance this could be done as follows. First specify a set of properties for each party (agent) involved in the contract. These properties may, most desirably, be written in $\mathcal{CL}$ too. The second step concerns only one party (the initiator) writing a (template) contract in $\mathcal{CL}$ and sending it to the other party. This, in turn, checks (using either model checking or proof theory, when the properties are also written in $\mathcal{CL}$) to see if all its properties are satisfied on the received contract. If some properties fail then the party modifies the contract accordingly and sends back a new contract to the initiator party. The counter-examples generated from a model checker could prove very useful for this phase of the negotiation. This same procedure of checking properties, changing the contract, and sending it back is to be repeated until no more properties fail. This is the point when an agreement has been reached and a contract has been defined between the parties. From this point the contract can be started (enacted) and the run-time monitoring can begin. The challenges of negotiation are to find how to decide when and how to stop the negotiation procedure.

Regarding the expressiveness of $\mathcal{CL}$ one can investigate *extensions with quantities like time and amounts* of $\mathcal{CL}$. Such extensions are desirable in the legal domain, where many contracts involve notions of time and amounts of money. Examples abound: deadlines like “within seven days” or “no longer than 3 hours” or “at the end of the week”; amounts of money most often appear like prices for goods, for which arithmetic operations might be specified like “double” or “the same as”. Most probably quantities should be added at the level of the actions algebra. There, actions may have durations, and special actions might be defined for which special operations may be allowed, like arithmetic operations or comparisons in the case of time. In this respect there are algebraic formalisms that include time or more complex datastructures and one can start looking in the timed process algebras, like the mCRL2 algebra, or timed $\pi$-calculi.

More applications of $\mathcal{CL}$ can be investigated. One first application can be in the area of *Internet agents*, where $\mathcal{CL}$ contracts can be used to regulate the behavior and the goals of the agents. The reasoning system behind $\mathcal{CL}$ would give the reasoning power of the agents.

A second application can be automation of the analysis of legal contracts. Here $\mathcal{CL}$ can be more directly used as it is tailored to notions from legal contracts. One application that we already presented in the thesis is the detection of conflicts in contracts. But before anything can be done, a controlled natural language framework has to be used to translate English like written contracts in $\mathcal{CL}$. From here, questions can be answered in the form of yes/no properties, or even quantitative or temporal inquiries can be done.

Maybe more important is the fact that a formal representation of the ambiguous English written legal contract is obtained. From this formalization the semantics (i.e., the normative structure) reveals much of the structure of the contract, like the choice or the sequences of actions involved. Through some graphical tool, the informations obtained from $\mathcal{CL}$ can be valuable to the user and displayed in a graphical human intelligible way. A diagrammatic display
language can be readily defined on top of $\mathcal{L}$.

The study of the use of $\mathcal{L}$ as a semantic framework for other languages for Internet services lacking formal semantics can be very useful. Here I think of semantics for the BPEL language.

Contracts can be seen as specifications of the behavior of an object or component in a distributed object-oriented environment. In this sense one could investigate contracts as types. For example, in the Creol object-oriented language [JO07], objects are typed by interfaces. Behavior interfaces are a very simple form of contract (over histories of actions). $\mathcal{L}$-like contracts would be more powerful behavior specifications as types of objects. These ideas can go further and be applied to more complex components [OSS07].
Bibliography


Author’s Selected Bibliography


Appendix A

Rewriting theory and convergence for 
\* -free actions

We recall first classical notions and notations for term rewriting systems and equational unification from [BN98].

An order-sorted signature, denoted \( \Sigma \), is a finite set of functional symbols with a finite partially ordered set of sorts \((S, \leq)\). In our setting consider the signature of the \* -free actions algebra (denoted \( \mathcal{A}^D \) here) which has three binary function symbols (+,\cdot,\times), two special constants 1, 0 (i.e. a constant is a function symbol of arity 0), and another finite number of special constants \( \mathcal{A}_B \) which we called basic actions. The ordered sorts are \( \mathcal{A}_B \leq \mathcal{A}_B^* \leq \mathcal{A} \) where \( \mathcal{A}_B, \mathcal{A}_B^* \in S \) are the three elements (sorts) of the set of sorts \( S \). We assume an \( S \)-sorted family \( V = \{ V_s \}_{s \in S} \) of disjoint sets of variables. \( T(\Sigma, V) \) is the set of terms, and \( T(\Sigma) \) is the set of ground terms. \( V(t) \) with \( t \in T(\Sigma, V) \) a term represents the set of variables occurring in \( t \). The set of ground terms corresponds to the carrier set \( \mathcal{A} \) of the algebra \( \mathcal{A}^D \). We will be working in this section on the term algebra \( T_{\mathcal{A}^D} \). The set of positions of a term \( t \) is denoted \( Pos(t) \) and the set of nonvariable positions \( Pos_{\Sigma}(t) \). The subterm of \( t \) at position \( p \) is denoted \( t|_p \) and the replacing of the subterm at position \( p \) of the term \( t \) with a new term \( u \) is denoted \( t[u]_p \). A substitution \( \varsigma \) is a sorted mapping from a finite subset \( Dom(\varsigma) \) of \( V \) into \( T(\Sigma, V) \). The set of variables introduced by \( \varsigma \) is denoted \( Ran(\varsigma) \). Substitutions can be extended homomorphically from variables to the whole set of terms. A substitution applied to a term is denoted \( t_\varsigma \). Composition of two substitutions is denoted \( \varsigma\varsigma' \) and when applied to a term \( t_\varsigma \varsigma' \) it can also be understood as first applying \( \varsigma \) and afterwards applying \( \varsigma' \).

An identity (or equation) is a pair of terms \( t = t' \) with \( t, t' \in T(\Sigma, V) \). Intuitively, an identity holds in the term algebra iff the equality of the two terms is true for all the ways of replacing the variables. The equational theory induced by a set of identities \( E \) is the relation \( =_E = \{(t, t') \in T(\Sigma, V) \times T(\Sigma, V) \mid E \models t = t'\} \) where the symbol \( \models \) represents the semantics consequence relation. The axioms of the \( \mathcal{A}^D \) algebra is a set of identities, and the relation \( =_{\mathcal{A}^D} \) is a congruence over the set of actions from \( \mathcal{A} \).

A rewrite rule is a directed identity \( l \rightarrow r \) satisfying \( l \not\in V \) and \( V(r) \subseteq V(l) \). A variant of a rule \( l \rightarrow r \) is \( l_\varsigma \rightarrow r_\varsigma \) where \( \varsigma \) is a variable renaming substitution (i.e. a substitution which assigns to each variable from \( Dom(\varsigma) \) a fresh variable not from \( Dom(\varsigma) \)). A term rewrite system \( TRS \) is a set of rewrite rules. A rewrite relation \( \sim_R \) associated with a TRS \( R \) is defined as: \( t \sim_R t' \) ifff exists a variant \( l \rightarrow r \) of a rewrite rule from \( R \), \( \exists \varsigma \in Pos_{\Sigma}(t) \), and \( \exists \varsigma \) a substitution
s.t. \( t'_p = l_\zeta \) and \( t' = t[r_\zeta]_p \). The transitive-reflexive closure of \( \sim_R \) is denoted \( \sim_R^* \). We denote by \( =_R \) the transitive-reflexive-symmetric closure of \( \sim_R \). The same as \( =_E \) we have that \( =_R \) is a congruence. A derivation of a term \( t \) is a sequence \( t \sim_R t_1 \sim_R t_2 \sim_R \ldots \) of rewrite rules (sometimes also called rewrite steps). A TRS is said to be terminating iff there is no infinite derivation from any term. A TRS is confluent iff for any terms \( t, t_1, t_2 \) with \( t \sim_R t_1 \) and \( t \sim_R t_2 \) then \( \exists u \) s.t. \( t_1 \sim_R^* u \) and \( t_2 \sim_R^* u \). A TRS which is both terminating and confluent is called convergent (or complete).

We can associate to an equational system \( E \) an equivalent TRS \( R \) by directing in an arbitrary way the identities from \( E \). It is easy to see that the two systems are equivalent (i.e. \( t =_E t' \) iff \( t =_R t' \)) for any choice of direction of the identities. More important is to find a orientation s.t. \( R \) is convergent. Some identities are inherently nonterminating; for example the identity \( f(t, t') = f(t', t) \) which defines the commutativity of the functional symbol \( f \). For this purpose it is common to define a rewrite relation modulo a set of identities. The relation \( \sim_{R/E} \) is the composition of the two relations \( =_E \circ \sim_R \circ =_E \) where \( R \) is a set of rewrite rules and \( E \) is the set of “problematic” identities. Alternatively, \( t \sim_{R/E} t' \) iff \( \exists s, s' \in T(\Sigma, V) \) s.t. \( t =_E s, t' =_E s' \) and \( s \sim_R s' \). The notions of termination, confluence, and convergence are defined naturally as for the \( \sim_R \) relation. We consider an order-sorted rewrite theory to be \( R = (\Sigma, E, R) \) which has associated a rewrite relation \( \sim_{R/E} \). The theory is said to be terminating (respectively confluent or convergent) iff the rewrite relation is terminating (respectively confluent or convergent).

For the equational system of the \({}^*\)-free actions from Table 2.2 we obtain the equivalent order-sorted rewrite theory \( T_{A^D} = (\Sigma, R_{A^D}, E_{AC+\times}) \). The signature \( \Sigma \) is the same signature of the algebra \( A^D \). The rules of the TRS \( R_{A^D} \) are given in Table A.1. The set of identities \( E = \{A+, C+, A\times, C\times\} \) is the four axioms which define the associativity and commutativity of the functional operators \( + \) and \( \times \). The associated rewrite relation is done modulo AC (associativity and commutativity). These kind of rewriting as been well investigated and good algorithms are known for it. Note that the rewrite rule (12) is applied only to terms of sort \( A_B \) (i.e. only to basic actions denoted \( a \)). The rule (14) is applied to terms of sort \( A_B^\alpha \) which we denote by \( a_\alpha \). Terms of general sort \( A \) are denoted with \( \alpha, \beta, \gamma \).

To prove termination of the \( R_{A^D} \) we use the dependency pairs termination criterion [AG00] which is implemented in CiMEusing the polynomial interpretations method in order to generate the required orderings [CMTU05]. We present now how the dependency pairs termination criterion applies to our term rewriting system \( R_{A^D} \).

Let \( R_{A^D} \) be a set of rewrite rules. The set of defined symbols is \( D_{R_{A^D}} = \{ \text{root}(l) \mid l \rightarrow r \in R_{A^D} \} \); and the set of constructor symbols is \( C_{R_{A^D}} = \Sigma \setminus D_{R_{A^D}} \) (\( \text{root}(t) \) returns the symbol to the root of term \( t \)). For our concrete example \( D_{R_{A^D}} = \{+, \times\} \) and \( C_{R_{A^D}} = \{1, 0, \} \cup A_B \). A dependency pair is a pair of terms \( \langle f(s_1, \ldots, s_n), g(t_1, \ldots, t_m) \rangle \) s.t. \( \exists f(s_1, \ldots, s_n) \rightarrow C[g(t_1, \ldots, t_m)] \in R_{A^D} \) a rewrite rule where \( C[\cdot] \) is a context and \( f, g \in D_{R_{A^D}} \) are defined symbols. The set of dependency pairs is finite if the TRS is finite (which is in our case).

**Theorem A.0.1 ([AG00])** A TRS \( R_{A^D} \) is terminating iff there exists a well-founded weakly monotonic quasi-ordering \( \geq \) where both \( \geq \) and \( > \) are closed under substitution, s.t.

- \( l \geq r \) for all \( l \rightarrow r \in R_{A^D} \);
- \( s > t \) for all dependency pairs \( \langle s, t \rangle \).
(1) \( \alpha + 0 \rightarrow \alpha \)
(2) \( \alpha + \alpha \rightarrow \alpha \)
(3) \( (\alpha \cdot \beta) \cdot \gamma \rightarrow \alpha \cdot (\beta \cdot \gamma) \)
(4) \( \alpha \cdot 1 \rightarrow \alpha \)
(5) \( 1 \cdot \alpha \rightarrow \alpha \)
(6) \( \alpha \cdot 0 \rightarrow 0 \)
(7) \( 0 \cdot \alpha \rightarrow 0 \)
(8) \( \alpha \cdot (\beta + \gamma) \rightarrow \alpha \cdot \beta + \alpha \cdot \gamma \)
(9) \( (\alpha + \beta) \cdot \gamma \rightarrow \alpha \cdot \gamma + \beta \cdot \gamma \)
(10) \( \alpha \times 1 \rightarrow \alpha \)
(11) \( \alpha \times 0 \rightarrow 0 \)
(12) \( a \times a \rightarrow a \) for \( a \in \mathcal{A}_B \)
(13) \( \alpha \times (\beta + \gamma) \rightarrow \alpha \times \beta + \alpha \times \gamma \)
(14) \( (\alpha \times \beta, \gamma) \times (\beta \cdot \gamma) \rightarrow (\alpha \times \beta, \gamma) \cdot (\alpha \times \beta) \)

Table A.1: A TRS equivalent modulo AC to \( \mathcal{A}_D \) and which is convergent modulo AC.

Finding the required quasi-ordering can be done automatically and rather efficiently using dependency graphs and polynomial interpretations [AG00]. The method has been implemented in the CiME tool [CMTU05].

For our \( \mathcal{A}_D \) algebra the TRS \( \mathcal{R}_{AD} \) has a finite number of rules and thus a finite number of dependency pairs, therefore the method above can be applied. We have proven the term rewriting system \( \mathcal{R}_{AD} \) of Table A.1 to be terminating using CiME (see the implementation details below). Thus, we have that the rewriting relation \( \leadsto_R \) is terminating. Note also that the equivalence classes generated by the four identifies in \( E \) of associativity and commutativity are finite. This and the fact that \( \leadsto_R \) is terminating implies that \( \leadsto_{R/E} \) is terminating.

**Theorem A.0.2 (termination of \( \mathcal{T}_{AD} \))** The rewriting relation \( \leadsto_{R_{AD}/E_{AC+\times}} \) given by the order-sorted rewrite theory \( \mathcal{T}_{AD} = (\Sigma, R_{AD}, E_{AC+\times}) \) is terminating.

**Proof:** The proof is done using both the tool CiME and AProVE and is based on the discussion above. See details below. \( \square \)

Once the system is known to be terminating, proving confluence is done using the critical pairs method for proving local confluence. See below how we use CiME to prove confluence.

**Theorem A.0.3 (confluence of \( \mathcal{T}_{AD} \))** The rewriting relation \( \leadsto_{R_{AD}/E_{AC+\times}} \) given by the order-sorted rewrite theory \( \mathcal{T}_{AD} = (\Sigma, R_{AD}, E_{AC+\times}) \) is local confluent. Because it is also terminating (by Theorem A.0.2) it implies that the rewriting relation is confluent.

**Proof:** The proof was done using both the tool CiME and AProVE. \( \square \)

CiME is a tool for checking termination and confluence of term rewriting systems. It can do a lot more, like completion or unification. The latest version of CiME is 2.02 and incorporates
Rewriting theory and convergence for \(*\)-free actions

The input syntax is natural for a term rewriting systems. We have specified the term rewriting system from Table A.1 in CiME.\(^2\)

We need to define first the signature of our TRS and a set of variables to work with. CiME does not support order-sorting (like Maude does f.ex.). But we are lucky because our sorts are finite

```language=plaintext
let F = signature "
0,1,a,b : constant;
+ : AC;
. : infix binary;
& : AC;
";

let X = vars "x y z";

let R = TRS F X "
x + 0 -> x;
x + x -> x;
x . 1 -> x;
1 . x -> x;
x . 0 -> 0;
0 . x -> 0;
(x . y) . z -> x . (y . z);
(x . y) + (x . z) -> x . (y + z);
(x + y) . z -> (x . z) + (y . z);
x & 1 -> x;
x & 0 -> 0;
a & a -> a;
b & b -> b;
x & (y + z) -> (x & y) + (x & z);
(a . x) & (a . y) -> (a & a) . (x & y);
(a . x) & (b . y) -> (a & b) . (x & y);
(a . x) & (a & b . y) -> (a & a & b) . (x & y);
(b . x) & (b . y) -> (b & b) . (x & y);
(b . x) & (a & b . y) -> (b & a & b) . (x & y);
(a & b . x) & (a & b . y) -> (a & a & b) . (x & y);
a & (a . x) -> (a & a) . x;
a & (b . x) -> (a & b) . x;
a & (a & b . x) -> (a & a & b) . x;
b & (a . x) -> (b & a) . x;
b & (b . x) -> (b & b) . x;
b & (a & b . x) -> (b & a & b) . x;
a & b & (a . x) -> (a & b & a) . x;
a & b & (b . x) -> (a & b & b) . x;
";
```

---

\(^1\)The CiMEweb site is: http://cime.lri.fr/

\(^2\)See input specification files for the CiMEmtool on the COSoDIS project homepage: http://www.ifi.uio.no/cosodis/software.shtml
Appendix B

Example of a full legal contract

Here is an example of a legal contract between an Internet provider and a client.

This deed of Agreement is made between:
1. [name], from now on referred to as Provider and
2. the Client.

INTRODUCTION
3. The Provider is obliged to provide the Internet Services as stipulated in this Agreement.
4. If any provision or part provision of this Agreement is held invalid, unenforceable or illegal for any reason, the such provision or part provision which shall be deleted and the thus obtained Agreement shall remain in full force.
5. DEFINITIONS
5.1. In this Agreement unless the context otherwise requires:
   a) Agreement means this agreement, its recitals, clauses and any schedule of this agreement and any subsequent Services Request;
   b) Clause means a clause of this Agreement;
   c) Equipment specific to Client are materials created by the Provider for the Client during the running of this Agreement for provision of the Internet Service, which are specific to Client and not capable of being reused by the Provider.
      i) Equipment is Client Specific if and only if it is specifically marked as Equipment Specific to Client;
   d) Personal data means any data that identifies a person and which should be and remain confidential throughout this Agreement.
   e) Party means a party to this Agreement and its successors, trustees and permitted assigns;
   f) The Provider has as constitutive administrative officers the following:
      i) Client Relations Department for relations with the Client
      ii) Executions Department for managing Equipment and overall maintenance.
   g) Working hours is any period from 08:00 to 18:00 of each day of the week except the days of the weekend.
   h) Development Work means web site content development and integration services work that is to be provided to Client;
   i) Internet Services means Internet services, including, without limitation, ADSL and dial up connections, web site hosting, and Development Work.
   j) Internet traffic may be measured by both Client and Provider by means of Equipment and may take the two values high and normal.
k) Money references are references to US currency

l) **Force Majeure** means a circumstance beyond the reasonable control of the **Provider**, which results in **Provider** being unable to perform on time an obligation under this Agreement.

**OPERATIVE PART**

6. GENERAL

6.1. Internet Services shall be supplied by the **Provider** or by any other tertiary party employed by the **Provider**. Should any such third party not be able to provide the Internet Services within a reasonable time then **Provider** may procure the services of a similar organization at a similar cost.

6.2. When the Internet Services is requested periodically by the **Client** (eg per semester) then such Internet Services shall automatically be supplied again for a similar period unless **Client** advises otherwise the **Provider** in writing 30 days prior to the expiration of such a period.

7. **CLIENT’S RESPONSIBILITIES AND DUTIES**

7.1. The **Client** shall not:
   a) supply false information to the Client Relations Department of the **Provider**.
   b) interfere with the network or disrupt any other user, service or equipment;
   c) use Internet Services for any illegal, unauthorized or dangerous purpose;
   d) publish any material for which Client is not the Intellectual Property Right owner or is so authorized to publish; and
   e) transfer, assign, sell to or share with any other person any right under this Agreement.

7.2. Whenever the Internet Traffic is **high** then the **Client** must pay \[price\] immediately, or the **Client** must notify the **Provider** by sending an e-mail specifying that he will pay later.

7.3. If the **Client** delays the payment as stipulated in 7.2, after notification he must immediately lower the Internet traffic to the **normal** level, and pay later twice \((2 \times \[price\])\).

7.4. If the **Client** does not lower the Internet traffic immediately, then the **Client** will have to pay \(3 \times \[price\]\).

7.5. The **Client** shall, as soon as the Internet Service becomes operative, submit within seven (7) days the Personal Data Form from his account on the **Provider**’s web page to the Client Relations Department of the **Provider**.

7.6. The **Client** is responsible for, and without limitation, backup, any data owned by **Client** that is stored on the **Provider** system.

7.7. The **Client** must keep its password and user account details confidential and not disclose same to any other party. Should any such disclosure occur the **Client** shall report same to the **Provider** in writing as soon as possible.

8. **CLIENT’S RIGHTS**

8.1. The **Client** may choose to pay either:
   a) each month;
   b) each three (3) months;
   c) each six (6) months;

9. **PROVIDER’S SERVICE**

9.1. **Provider** shall use its best endeavors to ensure that the Internet Services are available at all times.

9.2. As part of the Service offered by the **Provider** the **Client** has the right to an e-mail and an user account.

9.3. **Provider** is obliged to offer with no limitation and within a period of seven (7) days a password and any other Equipment Specific to Client, necessary for the correct usage of the e-mail account, upon receiving of all the necessary data about the client from the Client Relations Department of the **Provider**.

9.4. Each month the **Client** pays the **bill** the **Provider** is obliged to send a Report of Internet Usage to
10. PROVIDER’S DUTIES

10.1. The Provider takes the obligation to return the personal data of the client to the original status upon termination of the present Agreement, and afterwards to delete and not use for any purpose any whole or part of it.

10.2. The Provider guarantees that the Client Relations Department, as part of his administrative organization, will be responsive to requests from the Client or any other Department of the Provider, or the Provider itself within a period less than two (2) hours during working hours or the day after.

11. PROVIDER’S RIGHTS

11.1. The Provider takes the right to alter, delete, or use the personal data of the Client only for statistics, monitoring and internal usage in the confidence of the Provider.

11.2. Provider may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:

a) delete any material found on its equipment and/or refuse to publish any material which is, in Provider’s opinion, unauthorized; illegal or possibly illegal; unlawful; obscene; infringes any Intellectual Property Right of any third party; defamatory; excessive in volume; uncollected for an excessive period; in an unauthorized area; dangerous; or in breach of Adroit’s Publishing Policy which may be updated and changed from time to time;

b) Suspend Internet Services immediately if Client is in breach of Clause 7.1;

c) Increase the Fees on 30 days written notice. In the event of a periodic Services Request the charges may change each renewal of such an order.

12. LIMITED WARRANTY

12.1. Client acknowledges that the Provider does not and cannot monitor or control the content and information accessed via the Internet and shall not hold the Provider responsible in any way for any content or information accessed via the Internet.

12.2. Provider shall supply Internet Services with all due care and skill.

12.3. Provider shall resupply any Internet Services which are not supplied in accordance with Clause 12.2 provided that Client notifies the Provider of same within a reasonable time of the supply of Internet Services.

12.4. Provider cannot and does not warrant that Internet Services will be available 24 hours a day or that any fault will be corrected within a specific time frame.

13. TERMINATION

13.1. Without limiting the generality of any other Clause in this Agreement the Client may terminate this Agreement immediately without any notice and being vindicated of any of the Clause of the present Agreement if:

a) the Provider does not provide the Internet Service for seven (7) days consecutively.

13.2. The Provider is forbidden to terminate the present Agreement without previous written notification by normal post and by e-mail.

13.3. The Provider may terminate the present Agreement if:

a) any payment due from Client to Provider pursuant to this Agreement remains unpaid for a period of 14 days;

b) Client breaches any Clause of this Agreement and such breach is not remedied within 14 days of written notice by Provider;

13.4. If this Agreement or any Services Request is terminated then the Provider may in its sole discretion:

a) retain all moneys paid, which is agreed to be a genuine estimate of part of Provider’s loss and damage suffered;
b) charge a reasonable sum for Internet Services performed in respect of which no sum has been previously charged;
c) be regarded as discharged from any further obligations under this Agreement.

14. DISCLAIMER OF WARRANTIES

omitted

15. LIMITATION OF LIABILITY

omitted

16. GOVERNING LAW

16.1. The Provider and the present Agreement are governed by and construed according to the Law Regulating Internet Services and to the Law of the State.

a) The Law of the State stipulates that any ISP Provider is obliged, upon request to seize any activity until further notice from the State representatives.

16.2. The parties irrevocably submit to the exclusive jurisdiction of the Courts of the State and any Courts hearing appeals from such Courts.