Thesis for the Degree of Licentiate of Philosophy

Analysing normative contracts

On the semantic gap between natural and formal languages

John J. Camilleri

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2015

Analysing normative contracts
On the semantic gap between natural and formal languages
John J. Camilleri

Technical Report 141L
ISSN 1652-876X

Research groups: Formal Methods & Language Technology
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Typeset in Palatino and Monaco by the author using XƎTEX
Printed at Chalmers
Gothenburg, Sweden 2015

Abstract

Normative contracts are documents written in natural language, such as English or Swedish, which describe the permissions, obligations, and prohibitions of two or more parties over a set of actions, including descriptions of the penalties which must be payed when the main norms are violated. We encounter such texts frequently in our daily lives in the form of privacy policies, software licenses, and service agreements. The length and dense linguistic style of such contracts often makes them difficult to follow for non-experts, and many people agree to these legally-binding documents without even reading them.

By investigating the processing of normative texts, how they can be modelled formally using a suitable logic, and what kinds of properties can be automatically tested on our models, we hope to produce end-user tools which can take a natural language contract as input, highlight any potentially problematic clauses, and allow a user to easily ask questions about the implications of the contract, getting a meaningful answer in natural language within a reasonable amount of time.

This thesis includes four research articles by the author which investigate the various components that a system such as this would require; from entity recognition and modality extraction on natural language texts, to controlled natural languages and visual diagrams as modelling interfaces, to logical formalisms which can be used for contract representation, to the different kinds of analysis possible and how this can be linked to user questions in natural language.

to Claudia
for never doubting me

Acknowledgements

"At some point during your PhD you will start to believe that you know more than your supervisor — but you will be wrong."

When I first heard this bit of advice from our head of graduate studies, I had no idea how true it would prove to be. That he was also my supervisor was merely incidental. But this quote nicely sums up Gerardo Schneider's style of supervision: always eager to teach, but careful to give you space. I have learnt many things from him, most of which cannot be found in any paper or book. I am very grateful to have such a dedicated, patient and wise supervisor.

I would also like to thank my co-supervisor Koen Claessen, for his special knack for asking questions I've never thought of, and for giving me new energy to solve problems which have tired me out. As if two great supervisors weren't enough, I am also lucky to have worked with Aarne Ranta in one way or another for over six years now. After first meeting in 2009, Aarne apparently saw something in me which I didn't see myself, and he is entirely responsible for me being here today. He has probably had the most major role in my career so far, and it would be no exaggeration to call him a mentor of mine.

The other side of the Gothenburg–Malta connection comes in the form of Mike Rosner and Gordon Pace, my two undergraduate supervisors who were in no small way my gateway into academia, and with whom I am very glad to have met and remained in touch with today.

More generally, I would simply like to thank all the people with whom my research has brought me into contact: my colleagues in the REMU project, the whole GF community, everyone I know through CLT, my friends at Chalmers, and all the administrative staff at our department. All of you together make working here a really great experience. Sorry if you came here looking for your own name!

Finally I must thank my parents for their absolute support, for never questioning the value of education, and for bringing me up to believe that I could achieve anything I wanted.

This work has been carried out within the REMU project — Reliable Multilingual Digital Communication: Methods and Applications. The project is funded by the Swedish Research Council (*Vetenskapsrådet*) under grant number 2012-5746.

Contents

Structure of this thesis

This thesis consists of an introduction followed by four individual research papers, each presented as a separate chapter. While their references have been combined into a single bibliography at the end of the thesis, the papers have otherwise only been edited for formatting purposes, and in general appear in their original form. Thus, it is normal that there is a small element of overlap in the introductions and related works sections of each of the papers.

Chapter 1

Introduction

1.1 Normative contracts

In this thesis we are interested in the modelling and analysis of real-world contracts expressed using the deontic norms of **obligation**, **permission** and **prohibition**. Such documents occur in many forms and in various contexts within our everyday lives, including privacy policies, terms of services, end user license agreements (EULAs), software licenses, service-level agreements, and regulations. What these examples all have in common is that they are predominantly written in **natural language** (NL). The works contained in this thesis only consider contracts written in English, but the methods described here can be applied analogously to documents in other natural languages, such as Swedish, Spanish etc. Our use of a formal interlingua means that a large part of the work is in fact language-agnostic. In considering documents which are authored in NL, we are specifically talking about contracts which are written by and for humans. Note that this excludes machine-readable contracts such as, for example, mobile application permission systems[1] which are defined abstractly first, and then given human-readable verbalisations after.

We refer to this somewhat broad class of documents as **normative contracts** (or **texts**). It should be noted that the term *contract* has different interpretations in different fields. In software engineering, *design by contract* refers to the specification of software in terms of assertions and pre- and post-conditions. The term *contract*, of course, is standardly defined in legal fields as an agreement between parties which is protected by law. In fact legal systems generally contain entire sections specific to contract law. A *contract* in the world of financial trading is yet another

[1] Android app permissions: `https://support.google.com/googleplay/answer/6014972?hl=en`, accessed 2015-08-11

concept: a promise of payment between parties, whose value changes over time and which can be traded as an asset.

While these concepts may be related in a general sense, for the remainder of this thesis we will use the term *contract* to refer to the class of normative texts described at the beginning of this section (unless otherwise stated).

1.2 Motivation

To better identify what kinds of contracts we are interested in and what we want to do with them, let us consider some motivation.

1.2.1 The *"biggest lie on the web"*

Anyone who uses computers and the internet will have come across license agreements and privacy policies which they must agree to before using a piece of software or service. These documents tend to be written in an esoteric legal style which most people do not have the expertise or patience to understand. Yet the majority of users agree to such documents anyway without ever reading them, even though they understand that they are entering into a legally binding agreement. This common habit — which has been called the *"biggest lie on the web"*[2] — can have a number of negative outcomes, including the erosion of respect for contracts and the potential for exploitation of users.

In an experiment organised by the security company **F-Secure** [29], a free public Wi-Fi hotspot was set up in London with a somewhat unusual terms of service. These terms contained a so-called *"Herod Clause"*, stating that users of the hotspot agreed to give up their eldest child to the service provider *"for the duration of eternity."* In the short period the terms and conditions were live, six people signed up. Of course such a clause would not stand up in court, and the experiment only set out to prove a point.

Awareness about personal information and online privacy is becoming more widespread. Terms of service documents play a central role in this, and some projects already exist which try to make such them easier to understand by users. For example, **Terms of Service; Didn't Read (ToS;DR)**[3] is a user initiative which rates and labels the terms and privacy policies of major websites, giving them classifications on a simple scale from *very good* to *very bad*. This rating is

[2] `http://biggestlie.com/`, accessed 2015-07-20
[3] `https://tosdr.org/`, accessed 2015-06-15

done manually by the community, and the classifications themselves are meant to be clear and easily accessible.

In a similar way, the **Privacy Icons**[4] project attempts to make internet users more aware of how their private information may be used by grading the privacy policies of various websites. They define a few general privacy criteria such as data retention, location information, and SSL support — each of which is represented by an individual icon. A browser plugin then gives colour-codes to each icon when visiting a particular site, to indicate how well it ranks in each area. The icons evolved from a Mozilla-led working group that included privacy organisations such as the Electronic Frontier Foundation, Center for Democracy and Technology, and W3C.

CommonTerms[5] is yet another project with similar goals, which started out as an effort to identify the most common terms from the policies of 20 popular websites. It has since been active in raising awareness of the problem, and in trying to connect together the various different groups dealing with this issue in one way or another. They are responsible for the *Biggest Lie*[6] campaign and for starting the OpenNotice[7] group.

In the domain of software licenses, the **Choose a License**[8] website aims to help developers choose an appropriate Open-Source Software (OSS) license for their project. They do this by summarising the most popular open-source licenses in use today in terms of what each license requires, permits, and forbids — allowing for quick and easy comparison between the various options available.

The different copyright licenses available from the **Creative Commons (CC)**[9] also use icons to indicate their main features. But their approach goes a little deeper, with all CC licenses incorporating a three-layer design. Firstly, each license begins as a traditional legal tool, in the kind of language and text formats that lawyers work with. Secondly, the licenses are also available in a non-technical human-readable format called the *Commons Deed*, summarising and expressing some of the most important terms and conditions. Thirdly, each license also comes in a machine-readable version — a summary of the key freedoms and obligations written into a format that software systems, search engines, and other kinds of technology can process. This is done using the **CC Rights Expression Language (CC REL)**[10].

The problem of understanding legal texts in the context of the web is becoming more well known, and technological approaches to solving it are becoming more common. At the same

[4] `https://disconnect.me/icons`, accessed 2015-07-20
[5] `http://www.commonterms.net/`, accessed 2015-07-20
[6] `http://www.biggestlie.com/`, accessed 2015-08-11
[7] `http://opennotice.org/`, accessed 2015-07-24
[8] `http://choosealicense.com/`, accessed 2015-06-15
[9] `https://creativecommons.org/licenses/`, accessed 2015-07-20
[10] `https://wiki.creativecommons.org/wiki/CC_REL`, accessed 2015-07-20

time, interest in how computers can be used in area of law and contracts is also growing from within the legal field. In a discussion of the effects that machine intelligence might have on the delivery of legal services, McGinnis and Pearce [52] believe that computers will take an increasingly larger role and eventually replace humans in certain tasks (e.g. legal search, generation of documents, and predictive analytics). They predict that as lawyers continue to embrace computaional tools in their work, such technologies will also become more available to non-lawyers, leading ultimately to *"the end of [their] monopoly"* over providing legal services. Surden [78] notes that the benefits of *computable contracting* could include reduced transaction costs associated with contracting process, new potential for analysis and prediction, and the possibility of autonomous *computer-to-computer* contracting.

1.2.2 Approaches to classification

Most of the projects mentioned in the previous section are concerned with classifying and rating normative documents according to various criteria. The simplest way of achieving this is of course manually: with a group of people first deciding on a set of criteria, reading the texts, and then classifying them. Doing this for a single terms of service document would probably require a couple of man-hours, and one person alone likely cannot even do this for all the terms and conditions they've ever signed.

Delegating this manual work to a community is one way of tackling this labour-intensive task, which is what **ToS;DR** is essentially doing. Given the right community, this approach may be feasible for classifying small numbers of prominent documents, such as the privacy policies of the most popular sites on the internet. But of course, the motivation for automating this task is great. A software tool that could process normative texts and automatically classify them according to some criteria would benefit everyone, from the writers of the policies to the users that must decide whether to accept them or not.

On its surface, this can be seen as a natural language processing (NLP) task of *classification*. Such tasks usually make good cases for the application of common machine learning techniques, given the availability of a suitable corpus for use as training data. There is in fact an active research community in the area artificial intelligence (AI) in the legal domain [9], and applying machine learning and other AI techniques to legal texts is by no means new [75, 57, 37]. Taking research such as this into the commercial world, a startup company out of Carnegie Mellon University called **LegalSifter**[11] specifically uses machine learning techniques to sell contract analysis services. Their software claims to help customers to better understand legal documents through

[11] `http://www.legalsifter.com`, accessed 2015-07-24

summarisation, sorting and advanced search tools.

1.2.3 Beyond classification

Automatic classification is a useful task which can help users to determine at a glance whether a particular normative contract meets some requirements or not. But there exist many other properties of contracts that we may wish to be able to determine automatically. Prisacariu [67] lists a number of interesting contract-processing tasks that would benefit from automation. The various questions we may have about a contract could be grouped into the following categories:

- **Summarisation** — can we summarise a long contract into a shorter outline?
- **Visualisation** — can we represent a natural language contract in other informative graphical formats?
- **Comparison** — how does one version of a contract compare with a previous version? What is the effective difference between two similar contracts?
- **Conflict detection** — does a contract contain the potential for conflicting obligations, making it impossible to satisfy?
- **Compatibility** — do two separate contracts conform with each other, such that I can satisfy both without violating either?
- **Simulation** — under a given contract, what would my obligations be after performing a certain action at a given time?
- **Property testing** — does a contract contain any loopholes? Is it possible for a party to escape its obligations without penalty?
- **Negotiation** — can the terms of a contract be negotiated and changed iteratively until both parties are satisfied with it?
- **Translation** — can a contract be translated from one natural language to another in a syntactically correct and meaning-preserving way?

All these kinds of tasks can be thought of generally as *contract analysis*. The works covered in this thesis focus specifically on the tasks of *conflict detection*, *visualisation*, *simulation*, and *property testing*. In the following section we describe our general approach to performing these types of analysis.

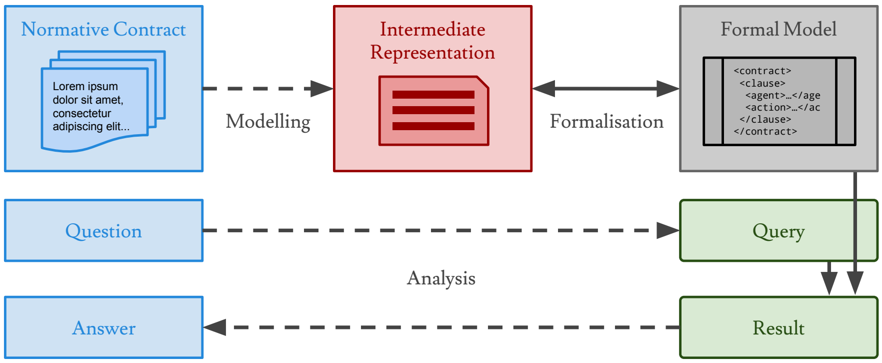


Figure 1.1: General overview of the method we use in our approach. Dotted lines represent tasks which currently require manual effort by the user. Solid lines are fully automatic.

1.3 Method

Any text in natural language may contain ambiguities or be otherwise unclear, and contracts are no exception. This can be due to context sensitivity, under-specified terminology, or convoluted linguistic style. As a result, it is not easy to apply the kinds of automatic analysis we are interested directly on natural language.

Instead, the approach we adopt is to first represent normative contracts using a **formal language** (FL) — that is, a language having a formal, well-defined syntax and semantics. A formal language has a limited but clearly-defined expressivity, and can be designed specifically to be precise and unambiguous. An expression in FL which represents a phrase or text in NL is called a **model**. By working with formal models, analysis can then become a well-defined task which can be automated. The design or choice of an underlying formalism is not easy, though. On the one hand, such a formalism should be expressive enough to facilitate the kinds of analysis desired. On the other hand, it should also be defined at a level of abstraction which is close enough to the target domain so as to allow for efficient modelling. In other words, the more abstract the formalism is, the harder it will be to take a NL sentence and find a formal representation for it.

Even with a formalism which satisfies these criteria, constructing a model to accurately represent a NL text — the **modelling** process — is not necessarily trivial. This task requires a certain level of expertise, not only at the syntactic level (the use of the formal language), but also in its semantics (the implied meanings of each construct and their composition).

Furthermore, performing the analysis itself generally also requires a particular level of competence. This includes the construction of queries or properties, running them against the model, and understanding the results obtained. This is also the case for so-called *push-button technologies* such as model checking, where one still needs to write the properties in a logic and interpret the counter-examples, which are usually given as long formulas representing the trace leading to the problematic case.

The ideal situation for an end user would be to have the benefit of both worlds; the simplicity and familiarity of NL combined with the power of formal methods, but without having to be involved in the technical details of the latter. This sums up the general goal of all the works contained in this thesis. This method of analysis contracts can be coursely split into the following tasks, summarised in Figure 1.1:

1. **Modelling** — providing tools for processing NL, and interfaces for building and working with formal models (possibly via an intermediate representation).

2. **Formalisation** — design or choice of a logical formalism to use for modelling contracts.

3. **Analysis** — processing contract models to detect conflicts, answer queries, or test if a property holds.

1.3.1 Modelling

Broadly, *modelling* is the process of going from a representation in some original format — in this case NL — into a corresponding representation in a target format — in this case our formalism. We tend to refer to modelling as the *front-end* of the system, and it is what the domain expert or end user will spend considerable time on. It is important to note that a model can only be an *interpretation* of an original text. The meaning of a model depends on the semantics of the formalism, and it may or may not match the original or intended meaning. In general, modelling is a process that can introduce errors or change meaning. Furthermore, if a normative text is inherently ambiguous, building a model of it in an unambiguous formalism can force the modeller to choose one out of multiple possible interpretations. The modelling process itself often uncovers such ambiguities.

Given that our goal is build a system for end users, we are of course interested in making the modelling process as user-friendly as possible. Thus we tend to provide an intermediate representation between the user and the underlying formalism (as shown in Figure 1.1). The idea is that while models in this intermediate representation are directly translatable into models in the formal language, they also have some properties which make them easier to handle for end users. To further explain this idea, we summarise the two main intermediate representations considered in this thesis: **controlled natural languages** and visualisation as **structured diagrams**.

Controlled Natural Language (CNL)

A *controlled natural language* [81] is a deliberately constrained or restricted version of a natural language, typically for the purposes of performing automatic processing of some kind. The syntax of a CNL is formally defined and generally simpler that of its parent NL, while its vocabulary is also reduced (at least in the case of structural words). These limitations make it possible to have a precise semantics for the CNL, which would be difficult or impossible for unrestricted language. For a survey of different kinds of CNLs, see Kuhn [49].

By designing a CNL which is close to a NL which the users understand, and using this as an interface for a purely logical language, users who are not experts in the underlying formalism can both read and write formal contract models, while at the same time not needing to deal with the issues that arise when unrestricted NL is used.

Both of the CNLs discussed in this work have been implemented using the Grammatical Framework (GF) [72], which is a programming language and platform for interlingua-based multilingual grammars. A distinct advantage of using GF is that it is built with translation in mind, making it a natural choice for building tools which convert statements in CNL into another representation, such as the concrete syntax of the formal language. As an example, consider the following sentence taken from the case study in Chapter 2:

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

Using the CNL defined in that chapter, this clause would be written as:

```
if {the flight} leaves {in two hours} then both
  - {the ground crew} must open {the check-in desk}
  - {the ground crew} must request {the passenger manifest}
```

This verseion of the clause contains some extra markup and is not entirely natural, but it is straightforward to follow for anyone who understands English.

The introduction of a CNL means that when creating models, users are restricted in what they can express. In other words, many expressions in NL will not be valid in the CNL. We thus come to the issue of how to help the user to know what is valid in the CNL and what is not. The simplest solution is to present some kind of manual for them to read, essentially teaching them the language's grammar. A better solution is to construct a suitable user interface (UI) which interactively helps the user construct valid CNL statements and provides helpful error messages. This idea is covered in this thesis (see Chapter 3) but also discussed further in Future Work (Section 1.6).

Visualisation

As an alternative to text-based representations of contract models, we also consider the idea of having a visual representations in the form of structured tree-like diagrams. For this we use the *Contract-Oriented (C-O) Diagram* representation introduced by Martínez et al. [51]. The motivation behind these diagrams is to help users clearly see the hierarchical and sequential dependencies that exist between the different clauses in a contract.

The main idea is that each box (Figure 1.2) represents a single norm, specifying a deontic modality (obligation, permission, or prohibition) over an agent and action. Each box may also have conditional expressions, timing restrictions, and reparation information. These boxes are then combined through operators for conjunction, sequence and choice, to build a complete model which is visualised as a graph structure (see example in Figure 1.3). Our work in Chapter 3 covers a web-based editor for working with *C-O Diagrams* and converting them into contract models.

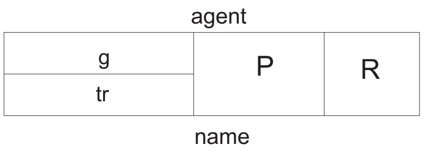


Figure 1.2: *C-O Diagram*: basic box structure where g = guard, tr = timing restriction, P = propositional content, R = reparation. [51]

1.3.2 Formalism

The formal languages used in this thesis for modelling contracts are based on deontic logic, containing at their core operators for **obligation**, **permission**, and **prohibition** of agents over actions. Deontic logics in general are known to suffer from a number of paradoxes, which mainly stem from problems or limitations attributed to Standard Deontic Logics (SDLs) [55, 53]. The formalisms covered here are thus not SDLs, but more restricted systems which take inspiration from deontic logic.

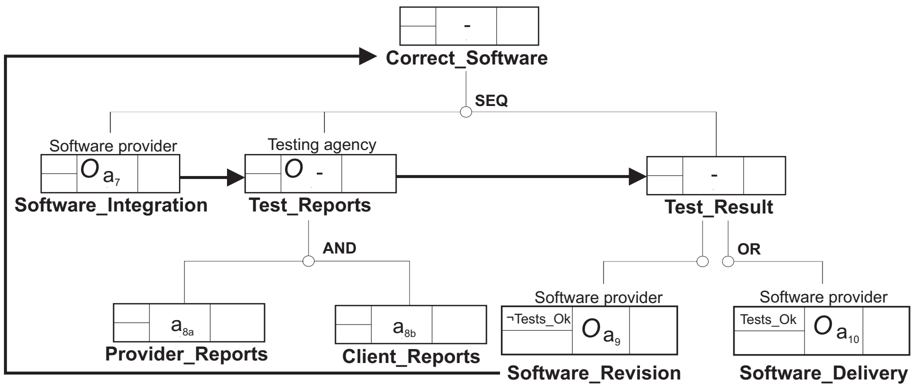


Figure 1.3: Full example of a *C-O Diagram*, modelling a software development process. [51]

In Chapter 2 we the use the $\mathcal{CL}$ language [69] as a logic for specifying normative contracts. $\mathcal{CL}$ is an *action-based* logic, meaning that modalities are applied to *actions* (e.g. "the customer must sign the agreement") as opposed to *state-of-affairs* (e.g. "the customer agreement must be signed"). Complex actions can be expressed using operators for choice, sequence, concurrency and repetition. Clauses in $\mathcal{CL}$ can also have **reparations** — sub-clauses which are applied as a penalty when the primary obligation or prohibition is violated. These are respectively referred to as *contrary-to-duties* (CTDs) and *contrary-to-prohibitions* (CTPs), and play a central role in how contracts are defined and used. $\mathcal{CL}$ combines deontic logic with propositional dynamic logic (PDL), and avoids many of the major paradoxes of SDL by applying to actions rather than states — the so-called *ought-to-do* approach [68, 70]. More details about $\mathcal{CL}$ can be found in Section 2.2.1.

The remainder of the works included in this thesis (Chapters 4, 3 and 5) use the *C-O Diagram* [26] language as a formalism. Apart from the visual representation discussed in the previous section, *C-O Diagrams* also have a formal syntax and can thus be considered a logic. As with $\mathcal{CL}$, the *C-O Diagram* language is based on the three main deontic modalities defined over complex actions. In addition, clauses themselves can combined with refinement operators for conjunction, sequence, and choice. The biggest novelty with *C-O Diagrams* is their handling of time using clocks, taken from the theory of Timed Automata (TA) [1]. Each *C-O Diagram* has a set of real-valued variables which increment at the same rate, representing the passage of time in the wall-clock sense. The value of a clock may be reset to zero, and any clock can be used to create a timing restriction on a clause. This makes it possible to express time windows and clause expirations, both in absolute time and relative to other clauses. *C-O Diagrams* also differ from $\mathcal{CL}$ in that agents are separated from actions, the condition expression language is a lot more expressive, and name-based referencing is used for reparations (as opposed to inline nesting). The *C-O Diagram* language is covered in more depth in Section 4.2.

1.3.3 Analysis

The point of modelling contracts in a formal language is to be able to perform some kind of analysis on them. In the case of $\mathcal{CL}$ contracts, we check these for deontic conflicts using the CLAN analysis tool [31]. Conflicts in this context can arise when there exists an obligation or permission together with a prohibition on the same action, or when two mutually exclusive actions are both permitted and/or obliged at the same time. The tool will search the entire possibility space of a contract, and if such a conflict is found it will return a counter-example in the form of a trace which leads to the conflicted state. This kind of analysis does not require any query or property as input. The AnaCon framework described in Chapter 2 automatically verbalises any counter-examples produced in the analysis using the same CNL for defining the contract model.

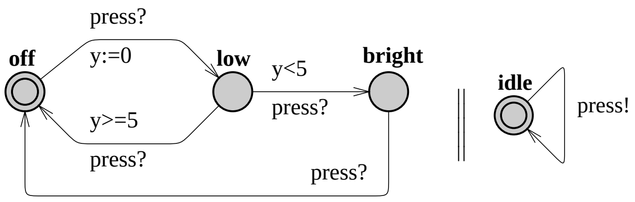


Figure 1.4: Example of an NTA, modelling the state transitions of a simple lamp (left) and the user operating it (right). [8]

The kind of analysis performed on *C-O Diagrams* is considerably more general than direct conflict detection. Díaz et al. [26] introduce a translation from *C-O Diagram* models into **networks of timed automata** (NTA) [10] (see Figure 1.4 for an example). NTAs are amenable to model checking using the using the Uppaal [8] tool, which allows properties written a subset of timed computation tree logic (TCTL) to be validated against the system. This allows us to test the following kinds of properties:

1. **Reachability** — is a certain scenario possible, given the constraints in a contract?
2. **Safety** — can we guarantee that an undesirable situation is always avoided?
3. **Liveness** — will a certain outcome always be eventually reached?

An important issue with this method of performing analysis is that the underlying NTA system, while semantically equivalent to the original *C-O Diagram* model, is at a much lower level of abstraction. While a *C-O Diagram* is defined in terms of clauses and refinement, NTAs are defined in terms of locations and edges. This means that a high-level query about *C-O Diagram* must first be converted into a lower-level property in TCTL in order to be tested against the NTA. This translation issue is not covered in the present work, but is one of the primary directions of future work (see Section 1.6).

1.4 Related work

Each paper in this collection includes its own related work section which lists works that are more closely related to the specific topic. We present here some more generally related work on the application of computational solutions to the legal domain.

Working with similar deontic formalisms, Pace and Schapachnik [62] present a method of modelling contracts which regulate two-party systems as automata. They are particularly interested in the relationship between permissions and obligations, interpreting them as a form of synchronisation between parties. For example, *"John is permitted to withdraw cash"* is both a permission for John as well as an obligation on the other party — the bank — to provide the withdrawal facility and honour it. This effectively treats permission as a first-class deontic modality. On top of this, the authors define a notion on *contract strength* which can be used to compare the relative strictness of two contracts.

Flood and Goodenough [32] also explore the idea of representing financial contracts as finite-state automata, where the automaton's locations represent the states that a financial relationship can be in (such as *default* or *delinquency*), and its transitions are labelled with events (such as *payment arrives* or *due date passes*). They then use standard automaton-based techniques to determine whether a contract is internally coherent and whether it is complete relative to a particular event alphabet. After illustrating their approach with a simple loan agreement case study, they go on to suggest how financial contracting could be conceived computationally in general and explore some of the wider implications that grow from viewing contracts as a system of computation.

Peyton Jones and Eber [65] introduce a functional combinator language for working with complex financial contracts — the kind which are traded in financial derivitive markets — together with an implementation[12] of it as a domain-specific language (DSL) embedded in Haskell. These kinds of contracts are somewhat different from the normative contracts we are concerned with in that they do not feature the deontic modalities, and are thought of as having an inherent financial value which varies over time. For recent work continuing this along these lines, see [6].

[12] `https://web.archive.org/web/20130814194431/http://contracts.scheming.org`, accessed 2015-09-09

In his PhD thesis, Wyner [82] presents the **Abstract Contract Calculator** — a Haskell program for representing the contractual notions of an agent's obligations, permissions, and prohibitions over abstract complex actions. Actions are treated as transition functions between states-of-affairs. The tool is designed as an abstract, flexible framework in which alternative definitions of the deontic concepts can be expressed and exercised. The program is however not a logic, and its high level of abstraction and lack of temporal operators means it is limited in its application to processing concrete contracts. In particularly, a focus of the work was on avoiding deontic paradoxes by using a dynamic language with rich markers for fulfillment and violation.

There is also considerable work in the representation of contracts as knowledge bases or *ontologies*, rather than using logic-like languages. The **LegalRuleML** project [5] embodies one of the strongest efforts in this area. LegalRuleML is a rule interchange format for the legal domain, allowing implementers to structure the contents of the legal texts in a machine-readable format. The format aims to enable modelling and reasoning that allow users to evaluate and compare legal arguments which have been constructed using the rule representation tools provided as part of the project. It is part of the larger RuleML initiative [74], set up within the OASIS consortium for open standards. LegalRuleML takes inspiration from another XML-based ontology language — the **Legal Knowledge Interchange Format (LKIF)**, developed as part of the ESTRELLA project [28] — but is considered to be more powerful than it, especially in the expression of temporal aspects [5].

A similar project with a broader scope is the **CEN MetaLex** language [12] — an open XML interchange format for legal and legislative resources. Its goals include enabling public administrations to link legal information between various levels of authority and different countries and languages, allowing companies to connect to and use legal content in their applications, and improving transparency and accessibility of legal content for citizens and businesses.

The **Semantics of Business Vocabulary and Business Rules (SBVR)** [60] uses a CNL to provide a fixed vocabulary and syntactic rules for expressing of terminology, facts, and rules for business documents. The goal is to allow natural and accessible descriptions of the conceptual structure and operational controls of a business, yet which at the same time can be represented in predicate logic and converted to machine-executable form. It also includes an associated XML Metadata Interchange (XMI) format, which supports the exchange of documents across businesses. SBVR is geared towards business rules, and not specifically at the kinds of normative texts in which we are interested.

Related to this work is also the area of *argumentation theory* — the study of how conclusions can be reached through logical reasoning. **Carneades** [36] is both a mathematical model of argumentation as well as a software toolbox for argument evaluation, construction and visualisation.

The software provides support for constructing, evaluating and visualising arguments using formal representations of facts, concepts, defeasible rules and argumentation schemes. Any number of argumentation schemes may be used together, making it an open architecture for hybrid reasoning. Carneades has been used in **Elterngeld**, a system which makes automatic decisions on child benefit claims in Germany. The system determines whether the statements put forward to support a particular claim are justified based on the tenets of the law, by coding them in a machine-readable format and then comparing them with elements of the law to score the claim [42].

When it comes to combining controlled language with visual representations, it is worth mentioning the work of Haralambous et al. [39] who introduce the idea of a **controlled hybrid language (CHL)** that allows information sharing and interaction between a CNL (specified by a context-free grammar) and a controlled visual language (specified by a symbol-relation grammar). Their INAUT system is used to represent French nautical charts together with their companion texts in a combined and complimentary way, effectively achieving the kind of multimodal presentation of structured data which we envision for normative contracts. The kind of charts in their system are cartographic maps, and use a visual formalism which is considerably different to *C-O Diagrams*.

1.5 Outline and contributions

The main body of work in this thesis is covered in the following research papers, each of which is presented as a separate chapter:

Paper I Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. A framework for conflict analysis of normative texts written in controlled natural language. *Logic and Algebraic Programming*, 82(5-7):216–240, 2013. doi: 10.1016/j.jlap.2013.03.002.

Paper II John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. A CNL for Contract-Oriented Diagrams. In *Workshop on Controlled Natural Language (CNL 2014)*, volume 8625 of *LNCS*, pages 135–146. Springer, 2014.

Paper III John J. Camilleri, Filippo del Tedesco, and Gerardo Schneider. Modelling and analysis of normative texts. 2015. (Under submission).

Paper IV John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider. Extracting formal models from normative texts. 2015. (Under submission).

In **Paper I** (Chapter 2) we start by taking the $\mathcal{CL}$ language and the accompanying conflict analysis tool CLAN. The work presents a CNL for $\mathcal{CL}$, and joins these components together to produce a basic tool for writing contracts in CNL and checking them for conflicts. My contributions here include implementing the AnaCon tool, which uses a GF grammar to parse CNL contracts into expressions in $\mathcal{CL}$, passes these expressions into the CLAN tool, and renders any potential counter-examples back into CNL using the same grammar. I also worked on applying to the tool to the two case studies described in the article.

Paper II (Chapter 3) also presents a CNL for contracts implemented using GF. In this work however we use the *C-O Diagram* formalism, which amongst other things includes the ability to express timing constraints on clauses and comes with a visual representation. Apart from the design of the CNL itself, this work also includes working implementations of a CNL editor, a visual editor for working with the diagrams, and a common interchange format between them. My role here was the complete design of the CNL, along with the web-based CNL editing tool and back-end conversion tools.

In **Paper III** (Chapter 4) we look further into the language of *C-O Diagrams* and the kinds of analysis possible on them. This includes some of our own modifications to the original formalism, the definition of a trace semantics, and a complete working implementation of the translation from *C-O Diagrams* into NTA. We apply this method to a small case study, and demonstrate our methods for syntactic and semantic analysis of contract models. My work here included the changes to the formalism, the work on the trace semantics, the entire implementation of the translation back-end, and the work on the case study.

Finally, **Paper IV** (Chapter 5) addresses the front-end task of modelling, investigating the use of NLP methods for parsing normative texts and building partial models in the *C-O Diagram* formalism. The idea behind this tool is to provide a first-pass processing phase which could automatically extract some information from a contract and reduce some of manual work required in modelling. My contribution in this work included consultation on the system's heuristic rules, the design of the experiments and carrying out the evaluation.

1.6 Reflections & future work

While each remaining chapter contains a short summary and a list of possible improvements, we present here some overall reflections about the current limitations of our work, together with some ideas for future work.

1.6.1 Structure of *C-O Diagrams*

Hierarchy. The design of *C-O Diagrams* promotes a structure which is highly hierarchical, where each clause is refined into sub-clauses in a tree-like fashion. Our experience of working with normative contracts such as terms of services is that these documents tend to in fact have very little depth. For the most part, these documents are written as a list of top-level clauses. One plausible explanation for this is that natural language does not nest particularly well; humans are more used to writing and expressing ourselves in a linear fashion, which is why the normative texts we have seen lack any deep structure. This kind of flat structure is supported by *C-O Diagrams* in the formal sense, but the tree-like visual representation of the formalism feels a little too hierarchical for models which are mostly flat.

Timing constraints. *C-O Diagrams* come with some powerful temporal features, where every clause can be given an activation window as well as a completion window, which may be expressed absolutely or relative to the completion of another clause. However, it seems to be the case that timing constraints on clauses are in fact quite rare in the kinds of normative contracts we have looked it. This is not to say that the ability to specify timing restrictions is an unimportant feature. However it must be recognised that the majority of normative clauses have no temporal specification at all, and are just implied to hold persistently for as long as the contract is in force. This should be realised in the form of reasonable defaults which take effect when temporal constraints are omitted — which seems to be true in the majority of cases.

Clocks. The idea of having multiple clocks which run simultaneously but which can be reset individually comes directly from the theory of timed automata. While it is powerful enough to allow expression of both absolute and relative timing constraints, doing so requires that the modeller has a thorough understanding of how and when these clocks are reset, and that they are able to encode their timing restrictions using clock conditions. We consider this to be the wrong level of abstraction for a user who is building contract models based on normative texts in NL. Rather, these kinds of low-level clock conditions should be replaced by more high-level temporal operators which are based on how we express time in NL, such as *since, until, between* and so on.

1.6.2 Design choices

Persistence. Should a normative clause apply continually or just once? From a natural language perspective, the answers seems to differ between the three modalities. Assuming that we are only considering norms over actions (not states), we are inclined to treat obligation as a one-time specification. For example, if you are obliged to pay a bill, then after paying it your obligation has been fulfilled and you should not have to pay it again (note that repetitive obligations, such as paying the rent every month, are a different matter). Prohibition on the other hand suggests persistence — if you are forbidden from killing, then this surely must continue to hold even if actually do kill someone. This tends to apply to permission too — for example, permission to drive a car. However it is not hard to think of an example where a permission should hold only once — such as the permission to vote. The question then arises as to whether persistence should be built into the formalism, whereby each norm can be explicitly marked as persistent, or whether we can avoid this and achieve the same thing with the right defaults and proper encoding. For example, a permission to vote exactly once could simply be guarded with a *"not yet voted"* condition.

Default modality. When a contract says nothing about action in its alphabet, what should its modality be? It seems excessively restrictive to prohibit all actions by default. But if all actions are permitted by default, what is the point of having a permission operator at all? It seems clear that an *explicit* permission should be stronger than a *default* permission. This also becomes useful when detecting conflicts — having both a prohibition and an explicit permission to do the same action is clearly something we want to avoid. An alternative approach to this would be to have every action prohibited by default. The purpose of permission then becomes clear, and the function of the prohibition operator is then to introduce a reparation for the violation.

Declarative statements. All the normative texts we have looked at so far generally include some non-normative statements, often defining the roles or details of the parties involved. We refer to these kinds of statements collectively as *declarations*. While important to the text from a legal point of view, these statements may or may not play a role in the application of the normative clauses with which we are concerned. Our approach so far has been to ignore such statements when modelling contracts, although as we work towards a more user-oriented tool this feature will become more important. A proper investigation of the best way of incorporating these statements into our contract models has yet to be carried out.

1.6.3 CNL editing

Along with the CNL introduced in Chapter 3, we also present a web-based tool for working with CNL contracts. This tool offers some basic features which are common to modern text editors, including syntax highlighting, text completion, templates, and section folding. In addition to this, there is a help panel showing the main types of constructs available in the language. While useful, features such as these are usually of most utility for users who are already familiar with the underlying language and know what kinds of constructs it supports. Realistically however, a user who is untrained in the CNL would still need some amount of training before being able to jump in and use a tool such as this.

We plan to continue developing the user interface aspects of working with CNL, experimenting with a combined multi-modal editor which brings together the structural view of the *C-O Diagrams* with the natural linearisations of CNL. This must definitely be combined with a thorough evaluation of usability, to measure the effectiveness of the different methods with respect to modelling effort required.

1.6.4 Querying

Levels of abstraction. The methods of analysis presented in Chapter 4 are quite powerful, making use of timed automata and the TCTL-based property language of UPPAAL. However, from the point of view of a user, these tools are very low-level, and the connection between automata and normative contracts is not obvious. In addition to the issues related to abstraction levels during modelling, we also have similar problems when it comes to performing analysis. Given a question in NL, this must first be converted into a low-level query. This requires not just knowledge of the property language, but also the details of the model and our translation algorithm (for example, the significance of location names and variable values in the resulting NTA). For this we may consider adopting OCLR [27], a temporal extension to the Object Constraint Language (OCL), which was designed with the analysis of legal texts in mind.

Once this is done and the query can be run, the next problem is how to interpret the result. While yes/no responses may be self-explanatory, any result that includes a counter-example trace requires skill and time to understand. As before, an in-depth understanding of the model and the details of its encoding are needed, together with the patience to follow a potentially long trace to determine the point of failure. The problem of how to best convert such traces back into answers which are meaningful to the user is still an open one.

Scalability. Model checking may be a very powerful method of analysis, but it is well-known that it can easily become intractable for non-trivial automata. The time required for verification is very sensitive to factors such as the number of locations, amount of variables, and the use of channel synchronisations. As such our work thus far has not focused at all on optimisation or even on evaluating the time and space requirements for analysis, yet this is also considered important future work. One possibility for avoiding the potentially long execution times associated with traditional model checking would be to investigate the use of statistical model checking techniques for NTA, which can provide faster testing of properties within some degree of certainty [23].

1.6.5 Translation

An area which has not been touched upon in this thesis is the *translation* of normative contracts from one natural language to another in a meaning-preserving way. While this task is orthogonal to the idea of general contract analysis, the methods we use in fact make this kind of machine translation quite a realistic task. As we are using a language-independent semantic representation for modelling contracts, the job of translation becomes a case of parsing from NL into our interlingua and linearising that to another NL. This model of translation is that supported by GF, which is already used for other purposes in our work.

Chapter 2

AnaCon: a framework for conflict analysis of normative texts

Krasimir Angelov, John J. Camilleri and Gerardo Schneider

Abstract. In this paper we are concerned with the analysis of normative conflicts, or the detection of conflicting obligations, permissions and prohibitions in normative texts written in a Controlled Natural Language (CNL). For this we present AnaCon, a proof-of-concept system where normative texts written in CNL are automatically translated into the formal language $\mathcal{CL}$ using the Grammatical Framework (GF). Such $\mathcal{CL}$ expressions are then analysed for normative conflicts by the CLAN tool, which gives counter-examples in cases where conflicts are found. The framework also uses GF to give a CNL version of the counter-example, helping the user to identify the conflicts in the original text. We detail the application of AnaCon to two case studies and discuss the effectiveness of our approach.

Chapter contents

2.1 Introduction

In this paper we present the AnaCon framework as a proof-of-concept system for the analysis of normative texts. We start by considering NL contracts taken from the real world, and describe a CNL which attempts to represent them in a meaningful way. We then explain and demonstrate the use of the AnaCon framework to transform such CNL contracts into expressions in a formal language which can then be analysed with a conflict detection tool. AnaCon also allows the translation of counter-examples (witnessing the existence of conflicting clauses) back into our CNL, facilitating the identification of the problem in the original text.

A conceptual model of AnaCon was first introduced in the workshop paper [58]. In this work we keep the same fundamental idea introduced there and consider the same class of contracts—namely those which can be expressed as formulae in the formal language $\mathcal{CL}$ and thus processed with the CLAN analysis tool. We have thus not changed the name of the framework, though most of the system design and implementation of the individual sub-modules has been changed significantly. In summary, the contributions of this paper are:

1. The definition and implementation of a CNL for writing normative texts. The CNL analyser implemented allows the parsing of full sentences by identifying relevant verbs, in particular those connoting obligations, permissions and prohibitions.
2. A formal syntax for the input file format to AnaCon, along with a parser that automatically extracts action names from the CNL text, taking away from the user the burden of including an action dictionary.
3. A complete implementation of AnaCon. We provide fully-working versions of all the modules described in the framework, including the translation from resulting counter-examples in the formal language $\mathcal{CL}$ back into our CNL.
4. The application of AnaCon to 2 case studies:
   (i) A work description procedure for an airport check-in desk ground crew, and
   (ii) A legal contract between an Internet provider and a client.

The paper is organised as follows. In the next section we recall the necessary technical background the rest of the paper is based on, including $\mathcal{CL}$, CLAN, CNLs and GF. In Section 2.3 we present our framework in general terms, and provide some details on the implementation of AnaCon. We then go into the application of the framework on two separate case studies in Section 2.4, as proof-of-concepts to show the feasibility of our approach. Before concluding in the last section, we discuss related work in Section 2.5.

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

Figure 2.1: $\mathcal{CL}$ syntax.

2.2 Background

In this section we present the background relevant to understanding the main components of AnaCon. We first introduce the contract language $\mathcal{CL}$, and continue with a description of the conflict analysis tool CLAN. We then discuss controlled natural languages in general and finish with a presentation of the Grammatical Framework.

2.2.1 The contract language $\mathcal{CL}$

The formal language $\mathcal{CL}$ has been designed for specifying contracts containing clauses determining the obligations, permissions and prohibitions of the involved parties [68, 69, 66]. $\mathcal{CL}$ is inspired by dynamic, temporal, and deontic logic, and combines concepts from each. Being *action-based*, modalities in $\mathcal{CL}$ are applied to actions and not to *state-of-affairs*. Complex actions can be expressed in the language by using operators for choice, sequence, conjunction (concurrency) and the Kleene star. $\mathcal{CL}$ also allows the expression of what penalties (*reparations*) apply when obligations and prohibitions are not respected, which form a central part of how contracts are defined and used.

For these reasons, $\mathcal{CL}$ was chosen for the underlying representation of the class of contracts in which we are interested. Combined with the availability of the conflict-detection tool CLAN (Section 2.2.2), $\mathcal{CL}$ forms the formal basis of the AnaCon framework. In what follows we present the syntax of $\mathcal{CL}$, and give a brief intuitive explanation of its notations and terminology, following [69]. A contract in $\mathcal{CL}$ may be obtained by using the syntax grammar rules shown in Figure 2.1.

$\mathcal{CL}$ contracts in general consist of a conjunction of clauses representing (conditional) normative expressions, as specified by the initial non-terminal C in the definition. A contract is defined as an obligation (C_O), a permission (C_P), a prohibition (C_F), a conjunction of two clauses or a clause preceded by the dynamic logic square brackets. $\top$ and $\perp$ are the trivially satisfied and violating contracts respectively. O, P and F are deontic modalities; the obligation to perform an action α is written as $O_C(\alpha)$, showing the primary obligation to perform α, and the reparation contract C if α is not performed. This represents what is usually called in the deontic community a *Contrary-to-Duty* (*CTD*), as it specifies what is to be done if the primary obligation is not fulfilled. The prohibition to perform α is represented by the formula $F_C(\alpha)$, which not only specifies what is forbidden but also what is to be done in case the prohibition is violated (the contract C); this is called *Contrary-to-Prohibition* (*CTP*). Both CTDs and CTPs are useful to represent normal (expected) behaviour, as well as alternative (exceptional) behaviour. $P(\alpha)$ represents the permission of performing a given action α. As expected there is no associated reparation, as a permission cannot be violated.

In the description of the syntax, we have also represented what are the allowed actions (α and β in Figure 2.1). It should be noted that the usage of the Kleene star (*)—which is used to model repetition of actions—is not allowed inside the above described deontic modalities, though they can be used in dynamic logic-style conditions. Indeed, actions β may be used inside the dynamic logic modality (the bracket $[\cdot]$) representing a condition in which the contract C must be executed if action β is performed. The binary constructors (&, ., and $+$) represent (true) concurrency, sequence and choice over basic actions (e.g. *"buy"*, *"sell"*) respectively. Compound actions are formed from basic ones by using these operators. Conjunction of clauses can be expressed using the $\wedge$ operator; the exclusive choice operator ($\oplus$) can only be used in a restricted manner. 0 and 1 are two special actions that represent the impossible action and the skip action (matching any action) respectively.

The concurrency (or *synchrony*) action operator & should only be applied to actions that can happen simultaneously. $\mathcal{CL}$ offers the possibility to explicitly specify such actions by defining the following relation between actions: $a \# b$ if and only if it is not the case that $a \& b$. We call such actions *mutually exclusive* (or *contradictory*). An example of such actions would be *"the ground crew opens the check-in desk"* and *"the ground crew closes the check-in desk"*, which intuitively cannot occur at the same time.

It is worth mentioning that much care has been taken when designing $\mathcal{CL}$ to avoid deontic paradoxes, as this is a common problem when defining a language formalising normative concepts (*cf.* [53]). Besides this, $\mathcal{CL}$ enjoys additional properties concerning the relation between the different normative notions, as for instance that obligations implies permissions, and that prohibition may be defined as the negation of permission. It has also been proven that some undesirable properties do not hold, such as that the permission of performing a simple action does not imply the permission of performing concurrent actions containing that simple action (similarly for prohibitions). See [68, 70] for a more detailed presentation of $\mathcal{CL}$, including a proof of how deontic paradoxes are avoided as well as the properties of the language.

Example

As an example of how $\mathcal{CL}$ can be used to represent contracts, let us consider the following sample clause:

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

Taking a to represent *"two hours before the flight leaves"*, b to be *"the ground crew opens the check-in desk"*, and c to be *"the ground crew requests the passenger manifest"*, then this clause could be written in $\mathcal{CL}$ as $[a]O(b\&c)$. We may also wish to include an additional reparation clause, such as:

> *If the ground crew does not do as specified in the above clause then a penalty should be paid.*

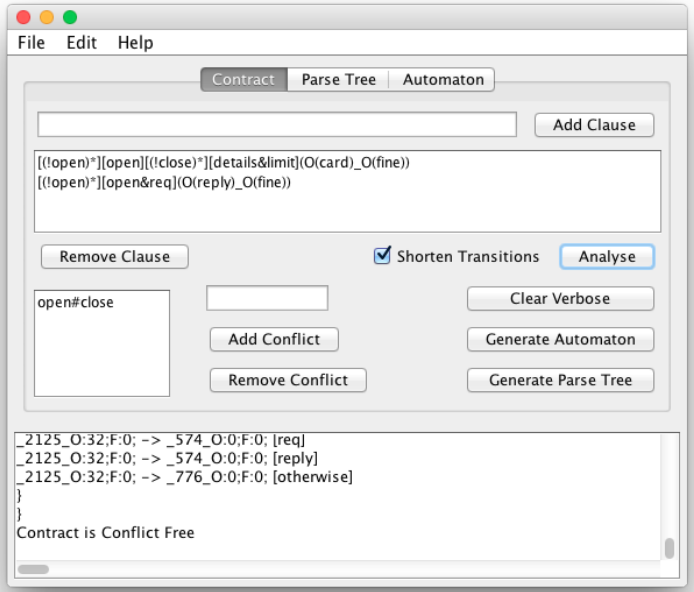
This penalty must be applied in case the ground crew does not respect the above obligations. Assuming that p represents the phrase *"paying a fine"*, one would capture all the above in $\mathcal{CL}$ as $[a]O_{O(p)}(b\&c)$.

This example serves not only to provide samples of normative statements written in $\mathcal{CL}$, but also to highlight the significant gap between natural language descriptions and formal representations of contracts. This paper attempts to bridge this gap through the introduction of an intermediary controlled natural language (CNL) to reconcile these two distinct representations. More background on CNLs can be found in Section 2.2.3.

2.2.2 CLAN

CLAN[1] is a tool aimed at the detection of normative conflicts in contracts written in $\mathcal{CL}$, giving the possibility for automatically generating a monitor for the $\mathcal{CL}$ formula [31]. There are four main kinds of conflicts in normative systems. The first arises when there is an obligation and a prohibition to perform the same action. Such cases will inevitably lead to a violation of the contract, independently of what the performed action is. The second type of conflict happens when there is a permission and a prohibition on the same action, which may or may not lead to a contradicting situation. The other two cases occur when there is an obligation to perform mutually exclusive actions, and when there exist both a permission *and* an obligation to perform mutually exclusive actions.

[1] `http://www.cs.um.edu.mt/~svrg/Tools/CLTool`, accessed 2015-08-13

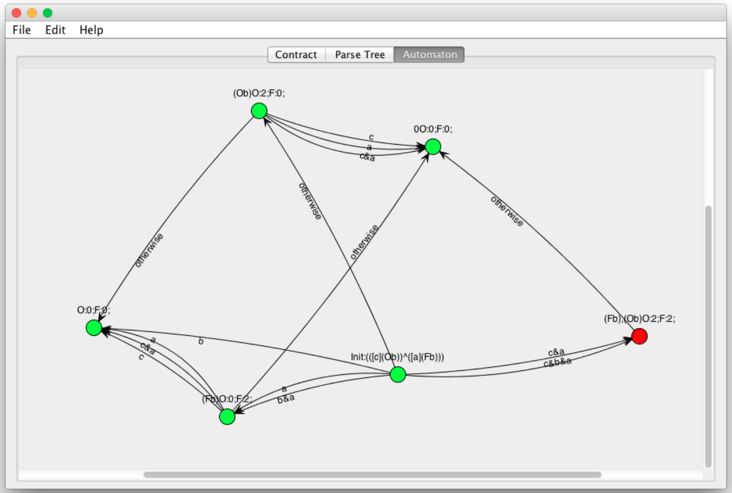


```
[(!open)*][open][(!close)*][details&limit](O(card)_O(fine))
[(!open)*][open&req](O(reply)_O(fine))
```

```
open#close
```

```
_2125_O:32;F:0; -> _574_O:0;F:0; [req]
_2125_O:32;F:0; -> _574_O:0;F:0; [reply]
_2125_O:32;F:0; -> _776_O:0;F:0; [otherwise]
}
}
Contract is Conflict Free
```

(a) Main user interface



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

Figure 2.2: Screenshots of the CLAN tool [31]

The core of CLAN is implemented in Java, consisting of just over 700 lines of code. The tool provides a graphical user interface as shown in the screen shot depicted in Figure 2.2a. CLAN allows the user to input a $\mathcal{CL}$ contract together with a list of the actions to be considered mutually exclusive. If a conflict is detected, CLAN gives a counter-example trace showing where the conflict arises and giving a sequence of actions realising the path to that conflict state. It is possible to visualise the corresponding automaton, as for instance shown in Figure 2.2b. The complexity of the automaton increases exponentially on the number of actions, since all the possible combinations to generate concurrent actions must be considered.

The analysis provided by CLAN enables the discovery of undesired conflicts. This is particularly useful both when a contract is being written, as well as before adhering to a given contract (to ensure its unambiguous enforcement). AnaCon uses CLAN as its back-end conflict analyser, yet abstracts over both the input to and output from CLAN via the CNL interface described in Section 2.3.2.

2.2.3 Controlled Natural Languages (CNLs)

CNLs are artificial languages engineered to be simpler versions of *full* (or *plain*) natural languages such as English. Such simplified languages are obtained through careful selection of vocabulary and restriction of grammatical rules, and are normally tailored to be used in a particular domain. Among other applications, CNLs are useful when considering human-machine interactions which aim for an algorithmic treatment of language. Unlike plain natural languages, the simplifications applied to CNLs usually allow them to be expressed and processed formally, while remaining easy to understand and use for speakers of the original parent natural language. This idea of using a CNL as a natural language-like interface for a formal system is not new [58, 35, 15], and is also the solution chosen in AnaCon.

In general, the richer a CNL is, the more complex is its automation. So, it is a challenge when designing CNLs to find a good trade-off between expressiveness (i.e. how close they are to natural languages) and formalisation. This trade-off is also affected by the richness of the parent NL and the formalism in which the CNL is defined [81].

As an example of the kinds of restrictions found in CNLs, consider again the following natural language clause:

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

Using the CNL introduced later in this paper, such a clause would be re-written as:

```
if {the flight} leaves {in two hours} then both
  - {the ground crew} must open {the check-in desk}
  - {the ground crew} must request {the passenger manifest}
```

Even though the structure of the CNL version is noticeably less natural, it is sufficient for our purposes to be merely *close enough* to English as to be understood by any non-technical person, while retaining the possibility of being unambiguously translated into an equivalent $\mathcal{CL}$ expression. It is worth mentioning that the conversion of NL to CNL is not necessarily a trivial process, owing to the ambiguities and potential for misinterpretation in NL. Conversely however, CNLs should be immediately understandable to any speaker of the parent NL, as the former is very much a subset of the other. This means that while it may require some training to convert a contract in NL to CNL, once that conversion has been made then anyone should be able to easily understand that CNL version of the contract. Further details about the design of the CNL for AnaCon is explained in Section 2.3.2.

2.2.4 The Grammatical Framework

With both the formal language $\mathcal{CL}$ and a controlled natural language for the framework in place, what remains is the software implementation for performing this bi-directional translation between representations. As in [58], we retain the use of the Grammatical Framework (GF) as a grammar formalism and runtime parser/lineariser for converting between CNL and $\mathcal{CL}$.

GF is a logical framework in the spirit of Harper, Honsell and Plotkin [40], which lets us define logics tailored for specific purposes, rather than trying to fit everything in a single model. At the same time, GF is also equipped with mechanisms for mapping abstract logical expressions to a concrete language. This is a distinct feature since most other logical frameworks come with a predefined syntax. This same feature is a notable characteristic of GF as a linguistic framework. While the logical framework encodes the language-independent structure (ontology) of the current domain, all language-specific features can be isolated in the definition of the concrete language. In other words, the definitions in the logical framework comprise the *abstract syntax* of the domain, while the *concrete syntax* is kept clearly separated [72]. This is a realisation of the separation between *tectogrammatical* and *phenogrammatical* features as was first proposed by Curry [21].

Furthermore, it is usual and actually very common to equip the same abstract syntax with several concrete syntaxes. Since GF has both a *parser* and a *lineariser*, in this case, the abstract syntax can serve as an interlingua. When a sentence is parsed from the source language, then the meaning of the sentence is extracted as an expression in the abstract syntax. The abstract expression then can be linearised back into some other language and this gives us bi-directional translation between any two concrete languages. Most of the time the concrete languages are natural languages, but it is also possible to define a linearisation into some formal language. In AnaCon, we have two concrete syntaxes—one for English (CNL) and one for the source language of CLAN ($\mathcal{CL}$). Thanks to the bi-directionality of GF we can go freely from CNL to logic and vice versa.

Another important advantage of GF from an engineering point of view is the availability of the *Resource Grammar Library* (RGL) [71]. Since every domain is logically different, it is also necessary to define different concrete syntaxes. When these are natural languages, then it means that a lot of tedious low-level details like word order and agreement have to be implemented again and again for each application. Fortunately, RGL provides general linguistic descriptions for several natural languages which can be reused by using a common language independent API. We implemented the AnaCon syntax for English by using this library, which both simplifies the development and makes it easy to port the system to other languages.

The GF runtime system also features an incremental parser, which can parse partial sentences and suggest valid completions according the underlying grammar [2]. While not used in the current version of AnaCon, this feature becomes very useful when composing sentences in CNL, as users do not necessarily need to know the specific grammar rules which define the language. In other words, the incremental parser can be used to provide a guided user-input experience. This feature was a further motivator for choosing GF as the framework for the implementation of AnaCon's CNL.

2.3 The AnaCon framework

In this section we start with the presentation of our framework, AnaCon, in general terms. We then discuss some issues concerning the particular CNL we are using as an input language for the framework, and present some details on the linearisation and parsing processes via GF.

2.3.1 System workflow

AnaCon takes as input a text file containing the description of a contract in two parts: (i) The contract itself written in CNL; (ii) A list of mutually exclusive actions.[2] Figure 2.3 shows a sample

[2]AnaCon can be downloaded from: `http://www.cse.chalmers.se/~gersch/anacon/`.

```
[clauses]
if {the flight} leaves {in two hours} then both
  - {the ground crew} must open {the check-in desk}
  - {the ground crew} must request {the passenger manifest}
[/clauses]
[contradictions]
{the ground crew} open {the check-in desk} # {the ground crew} request {the passenger manifest}
[/contradictions]
```

Figure 2.3: Sample contract file in AnaCon format.

of the input file to the framework, containing part of the description of what an airline ground crew should do before flights leave (details on the CNL syntax will be given in Section 2.3.2).

The entire system is summarised in Figure 2.4 where arrows represent the flow of information between processing stages. AnaCon essentially consists of a translation tool written in GF, the conflict analysis tool CLAN, and some script files used to connect these different modules together. The typical system workflow is as follows:

1. The user starts with a contract (specification, set of requirements, etc.) in plain English, which must be rewritten in CNL. This is primarily a modelling task, and it must be done manually. It requires no technical skills from the user, but does demand a knowledge of the CNL syntax and the set of allowed verbs.
2. The CNL version of the contract in AnaCon text format (Figure 2.3) is then passed to the AnaCon tool, which begins processing the file.
3. The clauses in the contract are translated into their $\mathcal{CL}$ equivalents using GF. This translation is achieved by parsing the CNL clauses into abstract syntax trees, and then relinearising these trees using the $\mathcal{CL}$ concrete syntax (see Section 2.3.3).
4. From the resulting $\mathcal{CL}$ clauses, a dictionary of actions is extracted. Each action is then automatically renamed to improve legibility of the resulting formulae, and a dictionary file is written. The list of mutually exclusive actions from the CNL contract is verified to make sure that each individual action actually does appear in the contract.
5. Using the renamed $\mathcal{CL}$ clauses from the previous step and the list of mutually exclusive actions, an XML representation of the contract is prepared for input into the CLAN tool.
6. This XML contract is then passed for analysis to CLAN via its command-line interface, which checks whether the contract contains any normative conflicts. If no such conflicts are found, the user is notified of the success. If CLAN does detect any potential conflicts, the counter-example trace it provides is linearised back into CNL using the GF translator

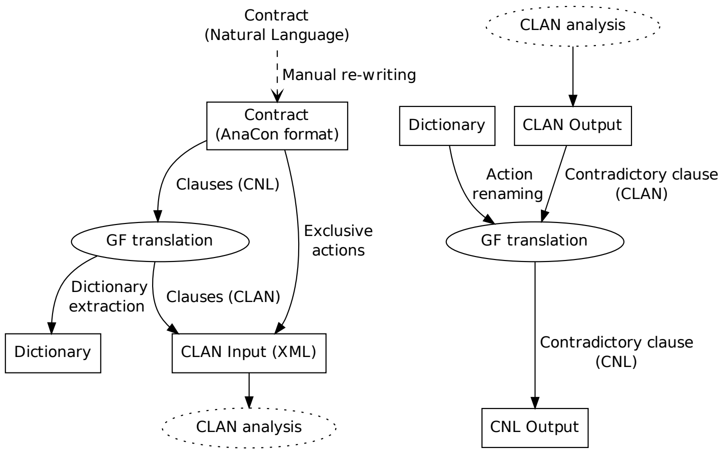


Figure 2.4: AnaCon processing workflow.

in the opposite direction. The dictionary file is used to re-instate the original action names.

7. The user must then find where the counter-example arises in the original contract. This last step must again be carried out manually, by following the CNL trace and comparing with the original contract.

2.3.2 About the CNL

Wyner et al. [81] have identified the following general questions one should ask when designing a CNL:

(i) Who are the intended users?
(ii) What is the main purpose of the language?
(iii) Is the language domain-dependent?

In our particular case we have the following answers to these questions:

(i) The intended user is any person writing normative texts;
(ii) The main purpose of the language is that it is close enough to English as to be understood by any person, yet at the same time structured in such a way that its translation into $\mathcal{CL}$ is feasible;
(iii) The language is not specifically tailored for an application domain, however, it should be easy to parse it in such a way that obligations, permissions and prohibitions are easily

identified.

Actions. The most primitive element in $\mathcal{CL}$ is the action and this is the starting point in the design of our CNL. While in $\mathcal{CL}$ these are just variable names, in natural language they correspond to sentences stating who is doing what. As a very rough approximation, every English sentence has an SVO structure (subject, verb, object), for example:

the ground crew	**opens**	**the check-in desk**
subject	*verb*	*object*

This is what we take as the basic syntax for actions in our CNL. Obviously, if this is taken directly, it will rule out many natural language constructions like the usage of adverbs and the attachment of prepositional phrases. These constructions usually express different moods for performing the action (e.g. *quickly, slowly, immediately*, etc.) or define time and space locations for the action (e.g. *at the airport*). As this kind of information cannot be expressed in $\mathcal{CL}$, we omit it from the CNL altogether. Still, since we permit the subject and the object to be free text, the user has the freedom to include more information than just the noun phrase of the subject or the object. It is also possible to have ditransitive verbs, i.e. verbs with more than one object. In this case we simply insert both objects in the free text slot for the object. If the verb is intransitive (without objects) then we can just leave the object slot empty.

The slot for the verb is not free text and must come from a set of predefined verbs. While we do not have to analyse the subject and the object slots, the ability to analyse the verb is important since we use modal verbs like *must* and *may* to indicate obligation, prohibition and permission. The restriction to use known verbs is not so hard since the grammar has a lexicon with all verbs from the Oxford Advanced Learners Dictionary [45, 56]. A given user will almost certainly find the verb that is needed—or a synonym of it—in the lexicon. The verb should be always in the present tense, and it can be in first, second or third person, in singular or plural. We check the tense but we cannot check the agreement with number and person, since we do not analyse the subject of the sentence. The only exception is when the verb is used with some of the modal verbs, then it must be in the infinitive.

When analysing the action, we must be able to correctly identify the beginning and the end of each slot, which is difficult when there are free text slots. Our simple solution is to require that the object and subject must be surrounded with curly braces, i.e. the user actually writes:

```
{the ground crew} opens {the check-in desk}
```

In some cases, the system can do the splitting even without the help of the curly braces since from the context it knows where each slot starts, and can guess the end of the slot by looking for known words. For instance we can guess the end of the slot for *the ground crew* since the next word *opens* is a known verb. Unfortunately, with the big verb lexicon this is often ambiguous since for instance *ground* is also a verb although here it is used as an adjective. The guessing can be made more sophisticated by using statistical part of a speech tagger which will try to predict whether *ground* is used as a verb or as an adjective. Unfortunately even the best part of speech taggers are still far from perfect, with precision of about 95%–97% (the precision of the Stanford Tagger, for instance, is 96.86% [80]). Instead, we opted for a solution that is simple and predictable. Integration of statistical tools can be done later, while still keeping bracketing as a safe alternative.

Connectives over actions. The two main operations on actions are concurrency (&) and choice (+). In natural language, they are represented by joining the sentences for the different actions with the conjunctions *and* and *or*. When there are more than two actions the usual English rules apply, i.e. the first actions are separated by comma and the last two with the conjunction. When the same expression mixes concurrency and choice, then in order to avoid ambiguities we use the usual conventions in logic and we give higher priority to the concurrency. In other words, if we have the expression `a and b or c`, then it will be interpreted as `(a and b) or c`. The user can also use parenthesis to override the default priorities.

A sequence of actions (.) in the CNL is introduced with the keyword *first*, followed by a list of actions. The actions are separated by commas except the last two which are separated with a comma followed by the conjunction *then*. For example:

```
first {the ground crew} opens  {the desk},
then  {the ground crew} closes {the desk}
```

We omit from the CNL the two special actions 0 and 1 since they have no obvious equivalent in English. Although they have useful algebraic properties in the logic, they do not appear naturally in any real contracts. A notable exception is the construction $[1^*]C$ which means that the clause C must be enforced at any state. For this purpose, we added the keyword *always* which can be used in front of any clause, which adds the condition $[1^*]$ in the corresponding $\mathcal{CL}$ formula. Similarly we did not include the Kleene star in our CNL, except for its use in relation to *always*.

Deontic modalities. On the next level, from every action, we can construct a clause expressing the obligation, the prohibition or the permission to perform an action. For representing the modalities we use the modal verbs *must*, *shall* and *may*, and the adjectives *required* and *optional*. In this way we implement the Internet recommendation RFC 2119[3] for requirement levels. The only difference is that they also define the verb *should* which is used for recommendations. Since the $\mathcal{CL}$ logic does not support this modality, we do not have it in the CNL either.

More concretely, if we take for example the action *"the ground crew opens the desk"*, then in the different modalities it can be written in one of the following ways:

Obligation

```
{the ground crew} must open {the desk}
{the ground crew} shall open {the desk}
{the ground crew} is required to open {the desk}
```

Permission

```
{the ground crew} may open {the desk}
it is optional for {the ground crew} to open {the desk}
```

Prohibition

```
{the ground crew} must not open {the desk}
{the ground crew} shall not open {the desk}
```

The two operations on clauses—conjunction ($\wedge$) and the exclusive choice ($\oplus$)—are rendered in English with the keywords *both* (or *each of*) and *either*, followed by a bullet list of clauses. Each list item starts on a new line and begins with a dash. If some of the list items contain clauses which themselves contain conjunction or exclusive choice, then the list items for such clauses must be indented with more spaces than the spaces before the dash of the parent clause. Contrary to the case with the concurrency and choice over actions, here we do not have any risk of ambiguity since the indentation level clearly indicates the nested structure of the logical formula.

Reparations. In the case of obligation and prohibition, the user can specify a reparation clause which must be hold if the contract is violated. In the CNL the reparation is introduced with comma and the keyword *otherwise* after the main action. For example:

```
{the ground crew} must open {the desk}, otherwise
{the ground crew} must pay {a fine}
```

Here we can have an arbitrarily long list of clauses, which are applied in the order in which they are written. The last clause is not followed by *otherwise*, which is an indication its reparation is $\perp$. This is also the only way to introduce $\perp$ in the logic. Similarly to 0 and 1 for actions, the clauses $\top$ and $\perp$ cannot be used directly in the CNL.

[3]http://www.ietf.org/rfc/rfc2119.txt, accessed 2015-08-13

The last thing to mention about the CNL is the syntax for conditions. As already mentioned, the syntax for the special condition $[1^*]C$ is introduced with the keyword *always* followed by the content of the clause C. The general conditions are introduced with the usual *if …then* statements in English, for example:

```
if {the ground crew} opens {the desk}
then {the ground crew} must close {the desk}
```

Note that here the verb *opens* is not used with a modal verb; this is an indication that this is an action and not a clause. In fact the expression between *if* and *then* can be a combination of many actions joined with the different action operators.

2.3.3 Linearisation and parsing in GF

In what follows we present how the major features of $\mathcal{CL}$ are represented in the abstract syntax, and look at how these features are handled in the concrete syntax for our CNL and the symbolic language for CLAN. As the chosen CNL covers a *subset* of $\mathcal{CL}$'s full expressivity, some $\mathcal{CL}$ operators are accordingly absent from the grammars—namely $\top$, 0, 1 and a^*. With the GF grammars for our two representations, the framework provides parsing and linearisation to and from the abstract syntax for free. In this way we can achieve two-way translation between the CNL and the CLAN language by having one concrete syntax for each, with shared abstract syntax.

To begin with, we define the following categories based on the BNF of $\mathcal{CL}$ (square brackets denote lists over a category). These correspond to the left-hand-side of the productions in Figure 2.1.

```
cat
  Act; [Act]; Clause; [Clause]; ClauseX;
  ClauseO; [ClauseO]; ClauseP; [ClauseP]; ClauseF;
```

Conjunction of clauses. In the abstract syntax, conjunction over clauses is defined as a function collapsing a list of heterogeneous clauses into one.

```
fun
  andC : [Clause] -> Clause ;
```

Our CNL as defined in Section 2.3.2 dictates that two or more clauses joined by conjunction should be bulleted and indented (for legibility and to avoid ambiguity), and preceded with a keyword token *both* or *each of*. As there are no other binary operations over clauses, operator precedence is not an issue (unlike for the action operators) and our code is fairly simple:

```
lin
  andC lst = indentS ("both"|"each of") (mkS bullet_Conj lst) ;
oper
  indentS : Str -> S -> S = \keyword,sen -> lin S {
    s = keyword ++ "[" ++ sen.s ++ "]" ;
  } ;
  bullet_Conj = mkConj "-" "-" ;
```

A few different things are happening here. Firstly, the linearisation of `andC` is delegated to the `indentS` operation[4], which prefixes our list with either of the variants *both* or *each of*, and encloses the rest of the term in square brackets. The role of the brackets is to encode the beginning and the end of an indentation level. Since GF grammars work on token level and the spaces and the new lines are ignored, they cannot handle the indentation directly. Instead a *custom lexer* and *unlexer* are used to convert between the square brackets and the indentation levels. In this way the indentation is handled outside of the grammars. We also see the reference to `mkS`, an operation defined in the GF Resource Grammar Library (RGL). This library call does all the work of joining our clauses into a single token list using a hyphen symbol as a delimiter (`bullet_Conj`).

Conditionals. The modality $[\beta]C$ is used to express conditional obligations, permissions and prohibitions, where the condition is a simple or compound action. The abstract syntax declaration and CNL linearisation are given below:

```
fun
  when : Act -> Clause -> Clause ;
lin
  when act c =
    mkS if_then_Conj (act.s ! Default) c ;
```

This example makes use of another version of the overloaded `mkS` operation from the RGL, which constructs an English *if ...then* sentence given the appropriate arguments. The linearisation of such a clause in the $\mathcal{CL}$ concrete syntax is a simple string concatenation:

```
lin
  when act c = "[" ++ act.s ++ "]" ++ "(" ++ c.s ++ ")" ;
```

Obligations, permissions and prohibitions. Obligations, permissions and prohibitions have a similar implementation as they all follow the same pattern. Each is built from an action and a

[4] **oper** judgements in GF are operations which can be re-used by linearisation judgements, but do not themselves represent linearisations of syntactic constructors.

reparation clause (CTP or CTD) where appropriate. Choice over obligations and permissions is defined in the same way as conjunction of clauses above.

```
fun
  O : Act -> ClauseX -> ClauseO ;
  P : Act -> ClauseP ;
  F : Act -> ClauseX -> ClauseF ;
  choiceO : [ClauseO] -> ClauseO ;
  choiceP : [ClauseP] -> ClauseP ;
```

Understanding the linearisation of an obligation also requires a look at the reparation clauses. While $\mathcal{CL}$ uses the bottom symbol $\perp$ to indicate a null CTD, in natural language it sounds very awkward to say something like *"one is obliged to pay a fine, otherwise nothing"*. It is much more natural to simply omit the *"otherwise nothing"* altogether. So, the linearisation of obligations is dependent on the type of the reparation clause (the `ty` field, where `False` indicates a null CTD).

```
lincat
  ClauseO = S ;
  ClauseX = {s : S; ty : Bool} ; -- CTD/CTP
lin
  O act cl = case cl.ty of {
    True  => mkS (mkConj ", otherwise") (act.s ! Obligation) cl.s ;
    False => lin S {s = cl.s.s ++ (act.s ! Obligation).s}
  } ;
  reparation c = { s = c ; ty = True } ;
  failure = { s = lin S {s=""} ; ty = False } ;
```

Actions. Atomic actions are defined as a "triple" containing a subject, a verb and an object, e.g. <*the crew, requests, the boarding pass*>. These are covered by the lexical categories `NP` (noun phrase), `V` (verb) and `NP` respectively:

```
flag
  literal = NP ;
cat
  NP ; V ;
fun
  atom   : NP -> V -> NP -> Act ;
```

By specifying the `literal = NP` flag, the GF compiler is instructed to treat `NP` as a literal category, which means its linearisation is that of a simple string. To achieve a degree of modularity between the logical and the linguistic, all verbs are defined in a separate abstract GF module `Verbs.gf`. In this case, the CNL concrete syntax `VerbsEng.gf` is also imported in the $\mathcal{CL}$ concrete syntax, exhibiting how GF's module system may help avoid duplication of code. The verbs themselves are also defined using the RGL, such that all that is required in our linearisation is a call to the `mkV` smart paradigm:

```
fun
  close_V : V;
  request_V : V;
  ...
lin
  close_V = mkV "close" "closes" "closed" "closed" "closing";
  request_V = mkV "request";
  ...
```

The CNL linearisation of actions is defined as a table parametrised with a *mode*. This essentially reflects the idea that a single action can be realised in four different modalities:

- Default: *the crew requests the boarding pass*
- Obligation *the crew must request the boarding pass*
- Permission: *the crew may request the boarding pass*
- Prohibition: *the crew shall not request the boarding pass*

With this approach, each atomic action internally contains each of these possible linearisations, which must be selected elsewhere in the grammar using the selection operator `!`.

To add a degree of naturalness to the grammar, we also introduce the concept of *linearisation variants*. Variants are a way of adding alternative, non-deterministic linearisations to an abstract syntax tree, and are defined in GF using the pipe symbol `|`. Using variants, we allow the single abstract syntax tree `O (atom (np "the crew") close_V (np "the check-in desk"))` to have any of the following linearisations:

1. *the crew is required to close the check-in desk*
2. *the crew shall close the check-in desk*
3. *the crew must close the check-in desk*

```
param
  Mode = Default | Obligation | Permission | Prohibition ;
lincat
  Act   = {s : Mode => S; p : Prec} ;
lin
  atom = mkAtom 0 | mkAtom 1 | mkAtom 2 ;
oper
  mkAtom : Ints 2 -> NP -> V -> NP -> {s : Mode => S; p : Prec} ;
  mkAtom n s p o = {
    s = table {
      Default     => mkS (mkCl s (mkVP (mkV2 p) o)) ;
      Obligation  => case n of {
        0 => mkS ...
        1 => mkS ...
        2 => mkS ...
      } ;
      Permission  => ...
      Prohibition => ...
    } ;
    p = highest
  } ;
```

Operations over actions are defined in the abstract syntax in a way which we are already familiar with. As $\mathcal{CL}$ defines more than one operator over actions, an order of precedence must be enforced to avoid ambiguities in phrases involving compound actions. With the help of the RGL's `Precedence` module, this is achieved by including a precedence field (`p : Prec`) in the linearisation type of actions. The linearisations of the operators are then explicitly given precedence levels, where conjunction is the highest (`p = 2`) and sequence is the lowest (`p = 0`).

```
fun
  andAct, choiceAct, seqAct : [Act] -> Act ;
lin
  andAct    as = {s = \\m => mkS and_Conj  (as!2!m); p=2} ;
  choiceAct as = {s = \\m => mkS or_Conj   (as!1!m); p=1} ;
  seqAct    as = {s = \\m => mkS then_Conj (as!0!m); p=0} ;
```

2.4 Case studies

In this section we apply AnaCon to two case studies, as a proof-of-concept of the feasibility of our approach. The first is concerned with the workflow description of an airline check-in, including the penalties applicable when the work is not carried out as prescribed. The second case study is a legal contract concerning the provision of Internet services. We finish the section with a discussion on the lessons learned from the case studies.

2.4.1 Case Study 1: Airline check-in process

Our first case study has been taken from [30]. It consists of the description of the check-in process of an airline company, given in Figure 2.5.

To show the modelling and re-writing process, we will first consider two clauses from this contract and show their equivalent CNL representations. Note that in our $\mathcal{CL}$ expressions, the

1. The ground crew is obliged to open the check-in desk and request the passenger manifest from the airline two hours before the flight leaves.
2. The airline is obliged to provide the passenger manifest to the ground crew when opening the desk.
3. After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass.
4. If the luggage weighs more than the limit, the crew is obliged to collect payment for the extra weight and issue the boarding pass.
5. The ground crew is prohibited from issuing any boarding passes without inspecting that the details are correct beforehand.
6. The ground crew is prohibited from issuing any boarding passes before opening the check-in desk.
7. The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before.
8. After closing check-in, the crew must send the luggage information to the airline.
9. Once the check-in desk is closed, the ground crew is prohibited from issuing any boarding pass or from reopening the check-in desk.
10. If any of the above obligations and prohibitions are violated a fine is to be paid.

Figure 2.5: Airline contract case study. [30]

actions have been renamed for brevity. This replacement is performed automatically by AnaCon and is completely reversible.

Original: *The ground crew is obliged to open the check-in desk and request the passenger manifest from the airline two hours before the flight leaves.*

CNL:

```
if {the flight} leaves {in two hours} then {the ground crew} must open {the check-in
     ↪ desk} and {the ground crew} must request {the passenger manifest from the
     ↪ airline}
```

For this clause, AnaCon gives the following $\mathcal{CL}$ formula as output:

$\mathcal{CL}$: `[b3]O(a7&b2)`

where from the dictionary file we see that:

```
b3 = {the flight} leave {in two hours}
a7 = {the ground crew} open {the check-in desk}
b2 = {the ground crew} request {the passenger manifest from the airline}
```

In the example above we see an obligation over two concurrent actions, which only become effective after an initial constraint is met—i.e. *if it is two hours before the flight leaves*. Note how this constraint is moved to the beginning of the clause and expressed using the *if* keyword. As defined by our CNL, conjunction over actions is expressed by joining together the individual actions with the keyword *and*. Conjunction over clauses however must be handled differently, as shown in the second example below:

> **Original:** *Once the check-in desk is closed, the ground crew is prohibited from issuing any boarding pass or from reopening the check-in desk.*
>
> **CNL:**

```
if {the ground crew} closes {the check-in desk} then both
  - {the ground crew} must not issue {boarding pass}
  - {the ground crew} must not reopen {the check-in desk}
```

AnaCon gives the following $\mathcal{CL}$ formula as output (again generating the corresponding action names in the dictionary file:

> $\mathcal{CL}$: `[b6]((F(a1))∧(F(a4)))`

In this case, using *and* to separate our clauses would be ambiguous with the conjunction over actions (shown above). Thus the bullet syntax is used here to clearly indicate the level of the conjunction.

While we have taken the above two examples individually, in real contracts clauses often refer to and depend on each other. When read in NL the reader can easily make the connections between the different clauses, but when it comes to modelling the contract formally these need to be handled explicitly.

Firstly, it is a common assumption that all the individual clauses in a contract are active together and thus there is an implicit conjunction between them. Furthermore, note how clause 10 in the example specifies a CTD for violating *any part* of the contract. Thus combining clauses 1, 8, 9, and 10 from the contract in Figure 2.5 we end up with:

> **CNL:**

```
if {the flight} leaves {in two hours} then each of
  - {the ground crew} must open {the check-in desk} and {the ground crew} must request
       ↪  {the passenger manifest from the airline}
  - if {the ground crew} closes {the check-in desk} then each of
    - {the ground crew} must send {luggage information to airline}
    - {the ground crew} must not issue {boarding pass}
    - {the ground crew} must not reopen {the check-in desk}
```

which results in the following $\mathcal{CL}$ formula:

```
CL: [b4]((O(b1&a2))∧[b6]((O(b2))∧((F(a1))∧(F(a4)))))
```

When processed with AnaCon, the first conflicting state reported was reached after a single action:

```
1 counter example found
Clause:
    (((O(a7&b2))_(Oa3))^(((Oa2)_(Ob1))^(([a7]((O(a6.(b4.(a8.a5))))_(Ob7)))^(((F(b5)_(
         ↪ Oa3))^(((Ob6)_(Oa3))^(([b6](Oa9))^(([b6](Fa1))^([b6](Fa4))))))))
Trace:
    1. the flight leave in two hours
```

Note that the counter-example above contains two parts: (i) a $\mathcal{CL}$ formula, and (ii) a trace in CNL. The first part is the formula representing the state of the automaton where the normative conflict happens, which is not particularly helpful for the end user. The second part is a linearisation of the output of CLAN showing what is the sequence of actions leading to the conflict; in this case only one.

A quick analysis of the original contract reveals that the two mutually exclusive actions *opening the check-in desk* and *closing the check-in desk* were erroneously obliged at the same level in the contract. This is a modelling error, and is corrected in a second version of the case study CNL.

When rewriting the second version we have not only addressed the issue of the arrangement of the actions corresponding to opening and closing the check-in desk, but we have also added more mutually exclusive actions. Such actions are considered mutually exclusive because they are logically contradictory and thus cannot happen at the same time, or because they cannot occur simultaneously due to physical constraints (e.g. *"the check-in crew issue the boarding pass"* and *"the check-in crew check that the passport details match what is written on the ticket"*). By adding such pairs of mutually exclusive (contradictory) actions we are avoiding some possible unnatural traces and at the same time reducing the size of the CLAN automaton, improving its time and space requirements.

By executing AnaCon a third time and analysing the counter-example given, it becomes apparent that there is something wrong with clause 5 (*cf.* Figure 2.5). In effect, this clause has two problems: (i) it is ambiguous as the whether the *"details"* refer to the passport or to the ticket;

(ii) it is redundant as it is somehow contained in clause 3. The latter adds some complexity to the analysis, so we decided to eliminate clause 5, without changing the intended meaning of the description.

Re-running AnaCon on this new contract also reveals another conflict, relating to the initiation of check-in and the closing of the gate being obliged at the same level in the contract:

CNL:

```
if {the airline crew} provides {the passenger manifest to the ground crew} then each
     ↪ of
  - first {the check-in crew} must initiate {the check-in process} ...
  - {the ground crew} must close {the check-in desk 20 mins before flight leaves} ...
  - if {the ground crew} closes {the check-in desk 20 mins before flight leaves} then
       ↪ ...
```

CL: `[a5]((O(a8&…))∧((O(b5))∧[b5](…)))`

Resulting AnaCon output:

```
4 counter examples found (only showing first)
Clause:
    ((((Oa8)_(Ob6))^([a8]((O(b4.(a7.a6)))_(Ob6))))^(((Ob5)_(Oa3))^(([b5](Ob1))^(([b5](
        ↪ Fa1))^([b5](Fa4))))))
Trace:
    1. the flight leave in two hours
    2. the ground crew open the check-in desk 2 hours before
    3. the ground crew request the passenger manifest from the airline
    4. the airline crew provide the passenger manifest to the ground crew
```

This leads to yet another re-writing of this final part of the contract, where the closing of the gate is now properly obliged *after* the initiation of the check-in process (note that by adding a new action, the re-written action names have changed):

CNL:

```
if {the airline crew} provides {the passenger manifest to the ground crew} then each
     ↪ of
  - first {the check-in crew} must initiate {the check-in process} ...
  - if {the flight} leaves {in 20 mins} then both
    - {the ground crew} must close {the check-in desk}
    - if {the ground crew} closes {the check-in desk} then each of
       - {the ground crew} must send {the luggage information to the airline}
       - {the ground crew} must not issue {boarding pass}
       - {the ground crew} must not reopen {the check-in desk}
```

1. The **Client** shall not:
   (a) supply false information to the Client Relations Department of the **Provider**.
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.
3. If the **Client** delays the payment as stipulated in clause 2, after notification he must immediately lower the Internet traffic to the **normal** level, and pay later twice $(2 * [price])$.
4. If the **Client** does not lower the Internet traffic immediately, then the **Client** will have to pay $3 * [price]$.
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**.
6. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:
   (a) Suspend Internet Services immediately if **Client** is in breach of clause 1;

Figure 2.6: ISP Contract case study.

Generated $\mathcal{CL}$: `[a6]((O(a9&…))∧`
`([a5]((O(b6))∧[b6]((O(b2))∧((F(a7))∧(F(a4))))))`

In order to truly cut down the size of the generated automaton to a bare minimum, a cross product of all possible mutually exclusive actions is generated using a simple shell script. From this, only the actions that *are allowed* to occur concurrently are removed; namely all those including the paying of fines, since a fine can be paid at any time. As this case study turns out to have a highly sequential nature, it makes sense that the list of mutually exclusive actions should be quite large.

Finally, after the iteration process described above we arrive at a final version of the contract without conflicts. It should be noted that for this case study we modelled the contract as a single instance of a sequence of events, i.e. considering a single airline and ground crew, a single check-in desk and indeed a single passenger. Extending the example with the *always* operator to model multiple check-ins occurring simultaneously introduces a number of difficulties and moreover reveals certain shortcomings of $\mathcal{CL}$ and CLAN. These are discussed further in Section 2.4.3 and Section 2.6.

2.4.2 Case Study 2: Internet Service Provider

We apply AnaCon here to part of a contract between an Internet provider and its clients, taken from [64]. The fragment of the contract which we will consider is reproduced in Figure 2.6.

The first clause imposes a prohibition for the client to give false information, while clauses 2 through 5 stipulate the obligations of the client in what concerns keeping the use of Internet below a certain limit (here specified as *high*) and the penalties to be paid in case these clauses are not respected. Clause 6 refers to the right of the provider to suspend the service if the client provides false information.

In what follows we rewrite the above clauses into our CNL and apply AnaCon. Our first attempt to analyse our CNL contract produces a parsing error on the following fragment:

CNL:

```
if {Internet traffic} becomes {high} then either
  - {the Client} must pay {price P}
  - each of
    - {the Client} must notify {the Provider ...}
    - if {the Client} notifies {the Provider ...} then {the Client} must lower {
         ↪ Internet traffic to the normal level}, otherwise {the Client} is required
         ↪  to pay {price 3P}
    - if first {the Client} notifies {the Provider ...}, then {the Client} lowers {
         ↪ Internet traffic to the normal level} then {the Client} must pay {price 2
         ↪ P}
```

The syntax error in this example stems from the use of disjunction (*either* on line 1) over clauses, which this is not allowed by $\mathcal{CL}$ and therefore in our CNL. The solution in this case is to treat this disjunction as a reparation, which is indeed the intended meaning in such cases:

CNL:

```
if {Internet traffic} becomes {high} then {the Client} must pay {price P}, otherwise
     ↪ first {the Client} must notify {the Provider ...}, {the Client} must lower {
     ↪ Internet traffic to the normal level}, then {the Client} must pay {price 2P},
     ↪  otherwise {the Client} is required to pay {price 3P}
```

$\mathcal{CL}$: `[a4]((O(a8)_((O(a2.b1.a9)_((O(a3)))))))`

Note how rewriting the above clauses actually leads to a neater implementation, both in terms of the CNL and in the underlying $\mathcal{CL}$ expression.

Running this corrected version of the contract with AnaCon, we are returned with a list of no fewer than 473 counter-examples which CLAN determined would lead to a state of conflict. An excerpt of the full output from CLAN shown below indicates that there is no proper handling of inherent sequence of the preliminary steps in the contract—i.e. those referring to customer data and the application procedure—which should apply *before* any clauses about the Internet traffic are even considered.

```
473 counter examples found (only showing first)
Clause:
    ((((F(a7)_(Pa1))^([1]([(*1)]((F(a7)_(Pa1)))))^((((Oa9)_(Oa3))^(((Oa8)_(((Oa2)_(Oa3
          ↪ ))^(([a2]((Ob1)_(Oa3)))^([a2]([b1]((Oa9)_(Oa3)))))))^(([a4]((Oa8)_(((Oa2)
          ↪ _(Oa3))^(([a2]((Ob1)_(Oa3)))^([a2]([b1]((Oa9)_(Oa3))))))))^([1]([(*1)]([
          ↪ a4]((Oa8)_(((Oa2)_(Oa3))^(([a2]((Ob1)_(Oa3)))^([a2]([b1]((Oa9)_(Oa3))))))
          ↪ )))))))^(([a5](Oa6))^([1]([(*1)]([a5](Oa6)))))))
Trace:
    1. Internet traffic become high
    2. the Client provide false information to the Client Relations Department of the
          ↪ Provider and the Internet Service become operative
    3. the Client notify the Provider ... and the Internet Service become operative
          ↪ and the Client submit ... the Personal Data Form ...
    4. the Client notify the Provider ... and the Internet Service become operative
          ↪ and the Client submit ... the Personal Data Form ...
    5. the Internet traffic become high and the Client lower Internet traffic to the
          ↪ normal level and the Client submit ... the Personal Data Form ...
```

More than a modelling problem, this tends to indicate some underlying assumptions in the original contract which need to be explicitly handled. This leads to the restructuring of the contract. In particular, the new contract was conceptually split into two sections, where all clauses referring to the application process form a prefix to the rest of the contract, which subsequently deals with the service once it has been activated. This is shown below (the corresponding $\mathcal{CL}$ formula has been omitted):

CNL:

```
if {the Client} submits {the data} then each of
  - {the Provider} must check {the data}
  - if first {the Provider} checks {the data}, then {the Provider} disapproves {the
        ↪ data} then {the Provider} may cancel {the contract}
  - if first {the Provider} checks {the data}, then {the Provider} approves {the data}
        ↪  then each of
    - {the Internet Service} must become {operative}
    - if {the Internet Service} becomes {operative} then always ...
```

It should be noted that this rewriting of the contract may in fact depart from the original meaning of the natural language contract we began with. This however should not be seen as a flaw;

indeed the very aim of contract analysis tools like AnaCon is to help identify weaknesses in existing contracts and facilitate their improvement.

Running this new contract through AnaCon produces a reduced—though still large—set of counter-examples from CLAN:

```
147 counter examples found (only showing first)
Clause:
    (((Ob1)_(Oa2))^(((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))
         ↪ )))))^(([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(
         ↪ Oa2)))))))))^([1]([(*1)]([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([
         ↪ a1]([b2]((Ob1)_(Oa2))))))))))))
Trace:
    1. the Client submit the data
    2. the Provider check the data
    3. the Provider approve the data
    4. the Internet Service become operative
    5. Internet traffic become high
    6. Internet traffic become high
    7. Internet traffic become high and the Client pay price P and the Client notify
         ↪ the Provider ...
    8. Internet traffic become high and the Client pay price P and the Client notify
         ↪ the Provider ...
    9. Internet traffic become high and the Client pay price P and the Client lower
         ↪ Internet traffic to the normal level
```

The initial reaction to this large number of counter-examples is to explicitly add more mutually exclusive actions to the contract to reduce the size of the automaton produced. While adding 5 pairs of exclusive actions reduces the number of possible counter-examples to just 18, a new issue with the contract emerges. In the new trace produced by CLAN it can be seen that if the action of the *Internet traffic becoming high* occurs twice (or more) in succession, the contract will always end in conflict, as shown in the counter-example below.

```
18 counter examples found (only showing first)
Clause:
    ((Oa2)^(((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))))))
         ↪ ^(([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))
         ↪ ))))))^([1]([(*1)]([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([
         ↪ b2]((Ob1)_(Oa2))))))))))))
Trace:
    1. the Client submit the data
    2. the Provider check the data
    3. the Provider approve the data
    4. the Internet Service become operative
    5. Internet traffic become high
    6. Internet traffic become high
    7. the Client pay price P and the Client notify the Provider ...
    8. the Client notify the Provider ...
    9. Internet traffic become high
```

Further analysis of CLAN output indicates that this issue is actually due to the use *always* operator, which essentially allows for parallel branches to be created in the contract automaton which cannot then both be satisfied. This ultimately points to a weakness in $\mathcal{CL}$. In our case we were able to achieve a contract-free contract by removing the *always* keyword on line 10, however this would arguably result in a non-intended meaning. A proper solution would require further remodelling or even augmenting $\mathcal{CL}$ itself.

2.4.3 Some reflections concerning the case studies

The two case studies examined in this paper come from unrelated domains. However they both share the property that they treat norms, and thus fall into the general group of texts which we are interested in analysing. While we do not claim that AnaCon is yet general enough to handle *any* such contract, we believe that these two case studies serve as a good proof-of-concept of the framework.

Applying AnaCon to the above 2 case studies provides us with some interesting insights on how to improve our framework.

The first observation is that our CNL is quite rich in terms of vocabulary and it is suitable as a high level language to be translated into $\mathcal{CL}$. However, the contract author needs to know the CNL syntax and be able to mentally convert NL clauses into valid CNL. This is for instance the case when writing obligations over sequences: it is not possible to write that in our CNL, and they must instead be written as a *sequence of obligations*, with only one CTD associated to the whole sequence. Though this is a limitation at the CNL level, it is not the case for $\mathcal{CL}$, as sequences of obligations cannot be expressed directly.

A second observation is that it would be desirable to have causal/temporal relationships among actions in addition to the declaration of mutual exclusive actions (#). This would allow a radical reduction in the size of the underlying CLAN automaton and thus improve efficiency and avoid some redundant counter-examples which are eliminated by rephrasing the CNL document. This redundancy is due to the semantics of &, discussed later in this section.

Concerning the output of CLAN, when a conflict is found the output produced by the CLAN tool consists of a list of tuples containing a conflict state and an action trace, as shown in Fig-

```
(((Ob1)_(Oa2))^(((Oa6)_(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2)))))))^(([a9]((Oa6)
      ↪ _(((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))))))))^([1]([(*1)]([a9]((Oa6)_
      ↪ (((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2)))))))))))
b3,a7,a8,a5,a9,a9,a9&a6&a1,a9&a6&a1,a9&a6&b2

(((Ob1)_(Oa2))^((((Oa1)_(Oa2))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))))))^(((Oa6)_(((Oa1)_(Oa2
      ↪ ))^(([a1]((Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2)))))))^(([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((
      ↪ Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))))))))^([1]([(*1)]([a9]((Oa6)_(((Oa1)_(Oa2))^(([a1]((
      ↪ Ob2)_(Oa2)))^([a1]([b2]((Ob1)_(Oa2))))))))))))
b3,a7,a8,a5,a9,a9,a9&a6&a1,a9&a6&a1,a9&b2
```

Figure 2.7: Sample CLAN output.

ure 2.7. In this output, CLAN is reporting all possible combinations of actions that would lead to a state of contradictions. As one can imagine this number could explode exponentially as the total number of actions increases, and for this reason adding multiple mutually exclusive actions to the $\mathcal{CL}$ contract helps to keep this under control. The two traces shown in Figure 2.7 end with the action expressions `a9&a6&b2` and `a9&b2` respectively, and it is fairly obvious to notice that in this example the performing of action `a6` along with `a9` and `b2` is, for our purposes, irrelevant. From this observation, it follows that we are not necessarily interested in all possible action combinations which could lead to a state of conflict; rather, we are interested only in the *minimal subset* of them. A fairly simple algorithm could be given to determine which are minimal counter-examples knowing then that any other counter-example would be thus redundant.

In fact, the above problem could easily be solved by eliminating the & action operator in $\mathcal{CL}$. After working on the above (and other small) case studies it would seem that it is not needed, as in most practical cases actions happening simultaneously are either uncommon, or can be expressed using interleaving. The elimination of this action operator will not only simplify the syntax but will radically reduce the complexity of CLAN (the main reason of exponential blow-up in CLAN's execution is due to such concurrent actions).

2.5 Related work

The basic ideas of this journal paper have appeared on the workshop paper [58], where the conceptual model of AnaCon was first introduced. The only commonalities between our current version of AnaCon and the one in [58] are the use of $\mathcal{CL}$ [69] and CLAN [31], besides the overall idea of the framework. In [58] it was shown that it was possible to relate the formal language for contracts $\mathcal{CL}$ and a restricted NL by using GF [72]. Our CNL, however, is based on a formal grammar inspired from NL sentences (i.e. using a subject, verb and complement) unlike that in [58] which was very much an "if-then-else" language enriched with keywords for obligation, permission and prohibition. Besides this, we have made extensive use of GF libraries and state-of-the-art constructions to make the definition of the abstract and concrete syntaxes much clearer and modular. We have reimplemented all the modules and implemented the counter-example generation in CNL, not done in the previous paper. Though we do not have (formal) experimental results to show the advantages of this new implementation, we do claim an improvement in performance and clarity of presentation based on its use in the case studies presented in this paper.

Using CNLs as a means to obtain a tractable language which is understandable to humans is not new. To date at least 40 different CNLs have been defined with different purposes and thus following different design decisions (*cf.* [81]).

A notable example of this is Attempto Controlled English (ACE) [34]. The difference between Attempto and our CNL is that while ACE aims to be an universal domain-independent language, we choose to make a language that is specifically tailored for the description of normative texts. Although ACE has syntactic constructions for expressing modalities, it also covers a lot of other constructions that we cannot handle in $\mathcal{CL}$. The proper handling of the whole language would make the underling logic unnecessarily complicated. Furthermore, ACE tries to perform full sentence analysis, while in our case this is not necessary since the semantics of the sentence would not be expressible in the logical fragment of $\mathcal{CL}$. Instead, we combine controlled language with free text which allows us to analyse only the relevant structures, while taking the rest as atomic literals. Another advantage of our choice is that the user does not need, as in ACE, to add new words for each domain since there is already a large lexicon of verbs and the nouns are just literals. A reimplementation of the original ACE grammar in GF has been presented in [3] where this controlled language was also ported from English to French, German and Swedish.

An initial exploratory design of another CNL specifically targeted for contracts is presented in [61], where the underlying logic and a sketch for the language are discussed. The chosen logic is actually close to $\mathcal{CL}$ except that it is more liberal. This broader logic gives flexibility in the translation to and from CNL, but it does not automatically exclude the possibility of paradoxes. In addition, their logic adds to $\mathcal{CL}$ temporal features as well as test operators for querying over the external game state. The logic is implemented with their own custom-build reasoner instead of CLAN. The actual CNL, however, is not implemented yet, and it remains only a sketch. Still, the initial design can be traced in Camilleri et al. [15], where, in their implementation of the game of Nomic, they employed a specialized CNL based on the same logic. The latter also used GF to translate between natural and logical representations, but their CNL involves only predefined actions and thus avoids the treatment of free-text, verbs, and actions as triples as in our approach. This also means that the system in [15] cannot be used for diverse contracts as in our case.

Our work is also similar to [38] where Hähnle et al. describe how to get a CNL version of specifications written in OCL (Object Constraint Language). The paper focuses on helping to solve problems related to authoring well-formed formal specifications, maintaining them, mapping different levels of formality and synchronising them. The solution outlined in the paper illustrates the feasibility of connecting specification languages at different levels, in particular OCL and NL. The authors have implemented different concepts of OCL such as classes, objects, attributes, operations and queries. The difference with our work is that $\mathcal{CL}$ is a more abstract and general logic, allowing the specification of normative texts in a general sense. In addition, we are not interested only in logic to language translation but rather in the use of the formal language to further perform verification (in our case conflict analysis) which is then integrated within our framework by connecting GF's output into CLAN, and vice versa.

It is worth mentioning that there is a general interest in the application of CNL for authoring and maintenance of legislative text. For instance [44] studies the typical linguistics structures in the German laws and relates them to constructions in first-order logic and deontic logic. The ultimate goal is the creation of Controlled Legal German as a human-oriented CNL for defining laws. Similarly [43] studies the legislative drafting guidelines for Austria, Germany and Switzerland, issued by the Professional Association for Technical Communication, from the perspective of controlled language. In both cases, however, the controlled language is aimed for human-to-human communication and its level of formalization is far from what is needed for computer based interpretation.

Rosso et al. [73] have used the passage retrieval tool JIRS to search for occurrences of words from a counter-example in natural language legal texts. In particular, they have applied their technique to a counter-example generated by CLAN on the airline check-in desk case study (the very same we have presented here as Case Study 1). JIRS is fed with a manual translation into English from $\mathcal{CL}$ formulae representing the counter-example given by CLAN, and uses an *n-gram* approach to automatically retrieve those sentences in the contract where the conflict occurs. JIRS does this by returning a ranking list with the passages found to be most similar to each query. We briefly discuss in next section how our work could be combined with passage retrieval tools like JIRS.

Finally, in what concerns deontic logic, and a presentation on the classical paradoxes, please refer to McNamara's article [53], and references therein.

2.6 Conclusion

We have presented in this paper AnaCon, a framework aimed at analysing normative texts containing obligations, permissions and prohibitions. We introduced a CNL for writing such texts, and provided a new and complete implementation of the AnaCon framework. AnaCon automatically converts normative texts written in CNL into the formal language $\mathcal{CL}$, using GF as a technology to perform bi-directional translations. The analysis performed on such texts is currently limited to the detection of normative conflicts, using the tool prototype CLAN. In line with the aims listed in the beginning of this paper, we have applied our framework to two case studies as a proof-of-concept of the system, detailing the iterative process that writing and revising such contracts involves. These two case studies have been specifically chosen from unrelated domains (one a document describing the working procedure of a check-in ground crew, and the other a legal contract on Internet services) in order to demonstrate that the CNL used is a general one. AnaCon is indeed agnostic in what concerns the content or final intention of the document to be analysed; what is important is that it contains clauses that could be analysed for normative conflicts.

While the mapping between $\mathcal{CL}$ and our CNL may seem trivial, we believe that the use of an intermediary CNL has some important benefits. As the CNL is more human-focused than the purely logical $\mathcal{CL}$, certain unnatural logical constructions have no equivalent representation in the CNL. In this sense, the CNL is strictly *less expressive* than $\mathcal{CL}$. Yet the nearness of CNL to regular unrestricted natural language, when compared to a purely formal language like $\mathcal{CL}$ can go a long way towards making the authoring of such contracts easier. The use of our CNL also allows actions names to contain arbitrary strings, which may convey valuable information for the human reading of the contract. They can also be very helpful when it comes to understanding the output of the conflict analysis step and identifying the source of conflicts within a contract. This is in fact a general property of CNLs; while it is true that constructing valid sentences in a CNL does require *some* training (although still less than is required to write pure logical formulas), *understanding* something written in CNL should be effortless for any speaker of the parent NL. In other words, the benefits of using CNL as a verbalisation for some formal language can be felt by both authors and readers.

2.6.1 Limitations

The intention for AnaCon is that it can become a general framework for analysis of any text which contains normative clauses. While the two case studies presented in this paper make a good argument for the framework's generalisability, we recognize that more extensive work would be required for it to reach that stage. Aside from this, we also identified a number of smaller issues with the current implementation.

First, though $\mathcal{CL}$ can be used as a formal language to specify normative texts in general, many aspects have to be abstracted away from, such as for instance timing constraints. Other limitations of $\mathcal{CL}$, and similar contract languages, are described in [63].

Secondly, there is the issue of CLAN efficiency. The current version is not optimised to obtain small non-redundant automata. The tool is very much a specialised explicit model checker, where a high number of transitions is generated due to the occurrence of concurrent actions. One practical way to reduce the size of the automaton created by CLAN is to try to identify and list as many mutually exclusive actions as possible. Note that some of the actions in our case studies are obviously mutually exclusive from the logical point of view (e.g. *open the check in desk* and *close the check in desk*), while others are mutually exclusive in a pragmatic sense, that is we know that they cannot occur at the same time (for instance, *issue a fine* and *issue the boarding pass*, if we consider that these actions are done by the same person). The performance of CLAN might be considerably improved by reducing the size of the automaton while building it, though a more fundamental way of improving it would be by eliminating & from $\mathcal{CL}$ as discussed in Section 2.4.3.

Third, CLAN is limited to conflict analysis and clearly it could be replaced by a more general model checker to check richer properties of normative documents in general, and contracts in particular.

As noted in Section 2.4.1 and Section 2.4.2, during the modelling and analysis of our two case studies problems were encountered with the *always* operator, expressed in $\mathcal{CL}$ as the prefix `[1*]`. While conceptually it is convenient and easy to think of a clause applying at all times, when modelled in $\mathcal{CL}$ and interpreted in CLAN it becomes clear that the true meaning of *always* in the natural sense is harder to formalise than anticipated. In order to overcome these issues, in this paper we were forced to exclude the use of this operator and instead model each contract as only covering a single "instance". The justification behind this is that if a contract holds for a single sequence of events, then it could later be generalised to run on concurrent instances of such sequences. In particular, we could consider adding features to the language to being able to distinguish between different instances of a contract, as done in the language FLAVOR [79].

2.6.2 Future work

Though in this paper we are not directly concerned with the translation from NL into CNL, it is worth mentioning that such translations could be carried out in a semi-automatic manner using guided-input techniques, or even better by using machine-translation.

In what concerns the ease of using CNL (vs. the use of a formal language) it could be very informative to perform experiments on different groups of users to have a qualitative analysis on the use of CNL and $\mathcal{CL}$. Evaluating CNL is not easy in general, and any experiment to do so should be carefully designed [47].

Another interesting future work concerns the use of passage retrieval tools like JIRS [73, 14] to help finding the counter-examples in the original English contract. This could be done by sending the CNL output from AnaCon to JIRS to automatically get a list of possible clauses where a conflict may arise. We envisage in this way a big increase in efficiency and precision when analysing counter-examples.

Finally, we believe that the development of a legal corpus could improve our CNL, giving the possibility to get a richer language even closer to natural language and enhancing the potential for obtaining a semi-automatic translation from NL documents into CNL.

Chapter 3

A CNL for *C-O Diagrams*

John J. Camilleri, Gabriele Paganelli and Gerardo Schneider

Abstract. We present a first step towards a framework for defining and manipulating normative documents or contracts described as *Contract-Oriented (C-O) Diagrams*. These diagrams provide a visual representation for such texts, giving the possibility to express a signatory's obligations, permissions and prohibitions, with or without timing constraints, as well as the penalties resulting from the non-fulfilment of a contract. This work presents a CNL for verbalising *C-O Diagrams*, a web-based tool allowing editing in this CNL, and another for visualising and manipulating the diagrams interactively. We then show how these proof-of-concept tools can be used by applying them to a small example.

Chapter contents

3.1 Introduction and background

Formally modelling normative texts such as legal contracts and regulations is not new. But the separation between logical representations and the original natural language texts is still great. CNLs can be particularly useful for specific domains where the coverage of full language is not needed, or at least when it is possible to abstract away from some irrelevant aspects.

In this work we take the *C-O Diagram* formalism for normative documents [26], which specifies a visual representation and logical syntax for the formalism, together with a translation into timed automata. This allows model checking to be performed on the modelled contracts. Our concern here is how to ease the process of writing and working with such models, which we do by defining a CNL which can translate unambiguously into a *C-O Diagram*. Concretely, the contributions of our paper are the following:

1. Syntactical extensions to *C-O Diagrams* concerning executed actions and cross-references (Section 3.2.3);
2. A CNL for *C-O Diagrams* implemented using the Grammatical Framework (GF), precisely mapping to the formal grammar of the diagrams (Section 3.3).
3. Tools for visualising and manipulating *C-O Diagrams* (Section 3.2):
   (a) A web-based visual editor for *C-O Diagrams*;
   (b) A web-based CNL editor with real-time validation;
   (c) An XML format COML used as a storage and interchange format.

We also present a small example to show our CNL in practice (Section 3.4) and an an initial evaluation of the CNL (Section 3.5). In what follows we provide some background for *C-O Diagrams* and GF.

3.1.1 *C-O Diagrams*

Introduced by Martínez et al. [51], *C-O Diagrams* provide a means for visualising normative texts containing the modalities of obligation, permission and prohibition. They allow the representation of complex clauses describing these norms for different signatories, as well as *reparations* describing what happens when obligations and prohibitions are not fulfilled. The basic element is the *box* (see Figure 3.4), representing a basic contract clause. A box has four components:

(i) *guards* specify the conditions for enacting the clause;
(ii) *time restrictions* restrict the time frame during which the contract clause must be satisfied;
(iii) the *propositional content* of a box specifies a modality applied over actions, and/or the actions themselves;

$$
\begin{aligned}
C &:= (agent, name, g, tr, O(C_2), R) \\
&\mid (agent, name, g, tr, P(C_2), \epsilon) \\
&\mid (agent, name, g, tr, F(C_2), R) \\
&\mid (\epsilon, name, g, tr, C_1, \epsilon) \\
C_1 &:= C\ (And\ C)^+ \mid C\ (Or\ C)^+ \mid C\ (Seq\ C)^+ \mid Rep(C) \\
C_2 &:= a \mid C_3\ (And\ C_3)^+ \mid C_3\ (Or\ C_3)^+ \mid C_3\ (Seq\ C_3)^+ \\
C_3 &:= (\epsilon, name, \epsilon, \epsilon, C_2, \epsilon) \\
R &:= C \mid \epsilon
\end{aligned}
$$

Figure 3.1: Formal syntax of *C-O Diagrams* [26]

(iv) a *reparation*, if specified, is a reference to another contract that must be satisfied in case the main norm is not.

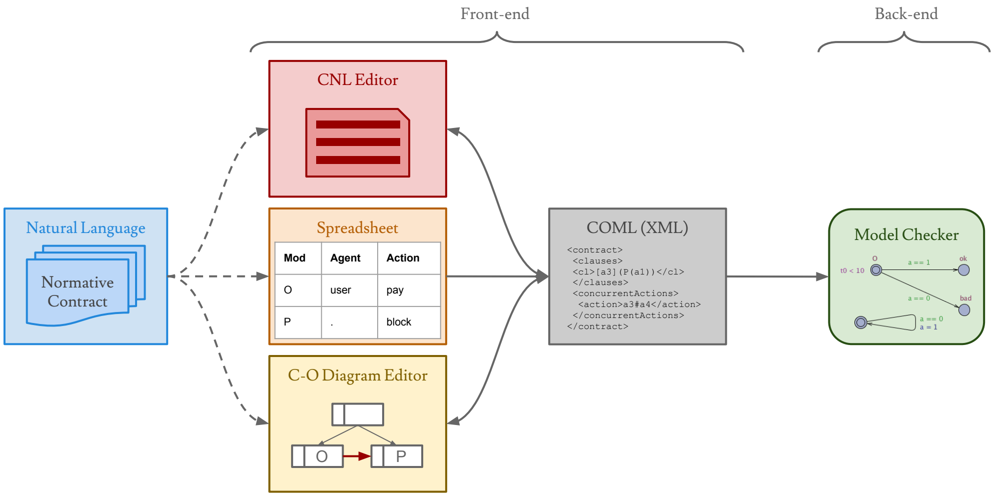
Each box also has an *agent* indicating the performer of the action, and a unique *name* used for referencing purposes. Boxes can be expanded by using three kinds of refinement: *conjunction*, *choice*, and *sequencing*.

The diagrams have a formal definition given by the syntax shown in Figure 3.1. For an example of a *C-O Diagram*, see Figure 3.5 (this example will be explained in more detail in Section 3.4).

3.1.2 Grammatical Framework

GF [72] is both a language for multilingual grammar development and a type-theoretical logical framework, which provides a mechanism for mapping abstract logical expressions to a concrete language. With GF, the language-independent structure of a domain can be encoded in the abstract syntax, while language-specific features can be defined in potentially multiple concrete languages. Since GF provides both a *parser* and *lineariser* between concrete and abstract languages, multi-lingual translation can be achieved using the abstract syntax as an interlingua.

GF also comes with a standard library called the *Resource Grammar Library* (RGL) [71]. Sharing a common abstract syntax, this library contains implementations of over 30 natural languages. Each resource grammar deals with low-level language-specific details such as word order and agreement. The general linguistic descriptions in the RGL can be accessed by using a common language-independent API. This work uses the English resource grammar, simplifying development and making it easier to port the system to other languages.



Mod	Agent	Action
O	user	pay
P	.	block

```
<contract>
 <clauses>
 <cl>[a3](P(a1))</cl>
 </clauses>
 <concurrentActions>
  <action>a3#a4</action>
 </concurrentActions>
</contract>
```

Figure 3.2: The contract processing framework. Dashed arrows represent manual interaction, solid ones automated interaction.

3.2 Implementation

3.2.1 Architecture

The contract processing framework presented in this work is depicted in Figure 3.2. There is a *front-end* concerned with the modelling of contracts in a formal representation, and a *back-end* which uses formal methods to detect conflicts, verify properties, and process queries about the modelled contract. The back-end of our system is still under development, and involves the automatic translation of contracts into timed automata which can be processed using the UPPAAL tool [50].

The front-end, which is the focus of this paper, is a collection of web tools that communicate using our XML format named COML.[1] This format closely resembles the *C-O Diagram* syntax (Figure 3.1). The tools in our system allow a contract to be expressed as a CNL text, spreadsheet, and *C-O Diagram*. Any modification in the diagram is automatically verbalised in CNL and vice versa. A properly formatted spreadsheet may be converted to a COML file readable by the other editors. These tools use HTML5 [59] local storage for exchanging data.

[1]An example of the format, together with an XSD schema defining the structure, is available online at `http://remu.grammaticalframework.org/contracts`

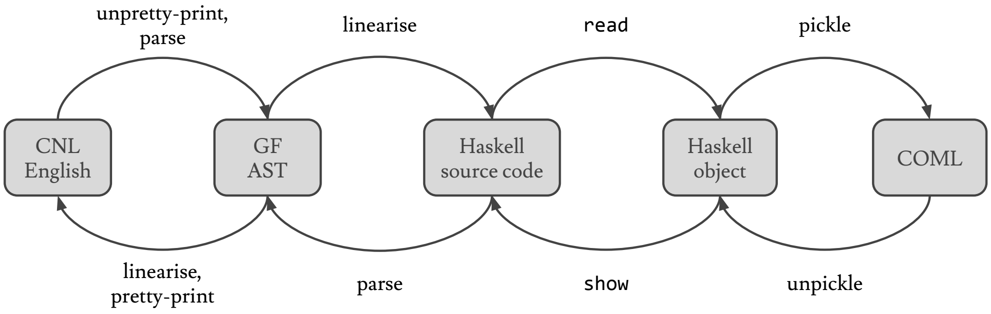


Figure 3.3: Conversion process from CNL to COML and back.

Translation process

The host language for all our tools is Haskell, which allows us to define a central data type precisely reflecting the formal *C-O Diagram* grammar (Figure 3.1). We also define an abstract syntax in GF which closely matches this data type, and translate between CNL and Haskell source code via two concrete syntaxes. As an additional processing step after linearisation with GF, the generated output is passed through a pretty-printer, adding newlines and indentations as necessary (subsubsection 3.3.2). The Haskell source code generated by GF can be converted to and from actual objects by deriving the standard `Show` and `Read` type classes. Conversion to the COML format is then handled by the HXT library, which generates both a parser and generator from a single *pickler* function. The entire process is summarised in Figure 3.3.

3.2.2 Editing tools

The visual editor allows users to visually construct and edit *C-O Diagrams* of the type seen in Section 3.4. It makes use of the mxGraph JavaScript library providing the components of the visual language and several facilities such as converting and sending the diagram to the CNL editor, validation of the diagram, conversion to PDF and PNG format.

The editor for CNL texts uses the ACE JavaScript library to provide a text-editing interface within the browser. The user can verify that their CNL input is valid with respect to grammar, by calling the GF web service. Errors in the CNL are highlighted to the user. A valid text can then be translated into COML with the push of a button.

3.2.3 Syntactic extensions to *C-O Diagrams*

This work also contributes two extensions to *C-O Diagram* formalism:

1. To the grammar of guards, we have add a new condition on whether an action a has been performed ($done(a)$);
2. We add also a new kind of box for cross-references. This enhances *C-O Diagrams* with the possibility to have a more modular way to "jump" to other clauses. This is useful for instance when referring to *reparations*, and to allow more general cases of "repetition".

Our tool framework also includes some additional features for facilitating the manipulation of *C-O Diagrams*. The most relevant to the current work is the automatic generation of clocks for each action. This is done by implicitly creating a clock `t_name` for each box `name`. When the action or sub-contract `name` is completed, the clock `t_name` is reset, allowing the user to refer to the time elapsed since the completion of a particular box.

3.3 CNL

This section describes some of the notable design features of our CNL. Examples of the CNL can be found in the example in Section 3.4.

3.3.1 Grammar

The GF abstract syntax matches closely the Haskell data type designed for *C-O Diagrams*, with changes only made to accommodate GF's particular limitations. Optional arguments such as guards are modelled with a category `MaybeGuard` having two constructors `noGuard` and `justGuard`, where the latter is a function taking a list of guards, `[Guard]`. The same solution applies to timing constraints. Since GF does not have type polymorphism, it is not possible to have a generalised `Maybe` type as in Haskell. To avoid ambiguity, lists themselves cannot be empty; the base constructor is for a singleton list.

In addition to this core abstract syntax covering the *C-O Diagram* syntax, the GF grammar also imports phrase-building functions from the RGL, as well as the large-scale English dictionary `DictEng` containing over 64,000 entries.

3.3.2 Language features

Contract clauses

A simple contract verbalisation consists of an **agent**, **modality**, and an **action**, corresponding to the standard subject, verb and object of predication. The modalities of obligation, permission and prohibition are respectively indicated by the keywords `required`, `may` (or `allowed` when referring to complex actions) and `mustn't` (or `forbidden`).

Agents are noun phrases (NP), while actions are formed from either an intransitive verb (V), or a transitive verb (V2) with an NP representing the object. This means that every agent and action must be a grammatically-correct NP/VP, built from lexical entries found in the dictionary and phrase-level functions in the RGL. This allows us to correctly inflect the modal verb according to the agent (subject) of the clause:

```
1 : Mary is required to pay
2 : Mary and John are required to pay
```

Constraints

The arithmetic in the *C-O Diagram* grammar covering guards and timing restrictions is very general, allowing the usual comparison operators between variable or clock names and values, combined with operators for negation and conjunction. Their linearisation can be seen in line 9 of Figure 3.6.

Each contract clause in a *C-O Diagram* has an implicit timer associated with it called `t_name`, which is reset when the contract it refers to is completed. These can be referred to in any timing restriction, effectively achieving relative timing constraints by referring to the time elapsed since the completion of another contract.

Conjunction

Multiple contracts can be combined by conjunction, choice and sequencing. GF abstract syntax supports lists, but linearising them into CNL requires special attention. Lists of length greater than two must be bulleted and indented, with the entire block prefixed with a corresponding keyword:

```
1 : all of
  - 1a : Mary may eat a bagel
  - 1b : John is required to pay
```

When unpretty-printed prior to parsing, this is converted to:

```
1 : all of { - 1a : Mary ... bagel - 1b : John ... pay }
```

For a combination of exactly two contracts, the user has the choice to use the bulleted syntax above, or inline the clauses directly using the appropriate combinator, e.g. `or` for choice. This applies to combination of contracts, actions and even guards and timing restrictions.

In the case of actions the syntax is slightly different since there is a single modality applied to multiple actions. Here, the actions appear in the infinitive form and the combination operator appears at the end of each line (except the final one):

```
2 : Mary is allowed
  - 2a : to pay , or
  - 2b : to eat a bagel
```

This list syntax allows for nesting to an arbitrary depth.

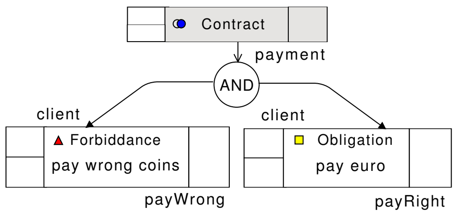
Names

The *C-O Diagram* grammar dictates that all contract clauses should have a name (*label*). These provide modularity by allowing referencing of other clauses by label, e.g. in reparations and relative timing constraints. Since the CNL cannot be lossy with respect to the COML, these labels appear in the CNL linearisation too (see Figure 3.6). Clause names are free strings, but must not contain any spaces. This avoids the need for double quotes in the CNL. These labels do reduce naturalness somewhat, but we believe that this inconvenience can be minimised with the right editing tool.

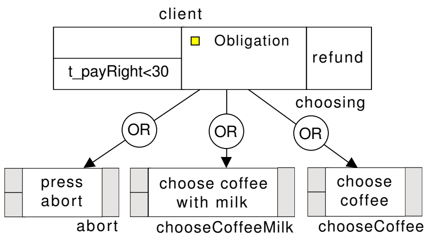
3.4 Coffee machine example

A user Eva must analyse the following description of the operation of a coffee machine, and construct a formal model for it. She will do this interactively, switching between editing the CNL and the visual representation.

To order a drink the client inputs money and selects a drink. Coffee can be chosen either with or without milk. The machine proceeds to pour the selected drink, provided the money paid covers its price, returning any change. The client is notified if more money is needed; they may then add more coins or cancel the order. If the order is cancelled or nothing happens after 30 seconds, the money is returned. The machine only accepts euro coins.



(a) Payment options



(b) Choices in selection

```
payment :
  payWrong : client mustn't pay wrong coins otherwise see refund and
  payRight : client is required to pay euro
choosing : when clock t_payRight less than 30 client is required
    - abort : to press abort , or
    - chooseCoffeeMilk : to choose coffee with milk , or
    - chooseCoffee : to choose coffee  otherwise see refund
```

Figure 3.4: Different kinds of complex contracts and their verbalisation.

Eva first needs to identify: (i) the *actors* (client and machine), (ii) the *actions* (pay, accept, select, pour, refund), (iii) and the *objects* (beverage, money, timer). The first sentence suggests that to obtain a drink the client *must* insert coins. Eva therefore drops an `obligation` box in the diagram editor and fills the name, agent and action fields. Only accepting euro is modelled as a prohibition to the client using a `forbiddance` box. The two boxes are linked using a `contract` box as shown in Figure 3.4a.

Eva now wants to model the choice of beverage, and the possibility the aborting of the process. She creates an `obligation` box named `choosing`, adding the timed constraint `t_payRight < 30` to model the 30 second timeout. She then appends two action boxes using the *Or* refinement, corresponding to the choice of drinks (see Figure 3.4b). Eva translates the diagram to CNL and modifies the text, adding the action `abort : to press abort` as a refinement of `choosing`. The result is shown in line 4 of Figure 3.6.

The *C-O Diagram* for the final contract is shown in Figure 3.5. It includes the handling of the `abort` action and gives an ordering to the sub-contracts. Note how there are two separate contracts in the CNL verbalisation: `coffeeMachine` and `refund`, the latter being referenced as a reparation of the former.

The *C-O Diagram* editor allows changes to be made locally while retaining the contract's overall structure, for instance inserting an additional option for a new beverage. The CNL editor is instead most practical for replicating patterns or creating large structures such as sequences of clauses, that are faster to outline in text and rather tedious to arrange in a visual language. The two editors have the same expressive power and the user can switch between them as they please.

3.5 Evaluation

3.5.1 Metrics

The GF abstract syntax for basic *C-O Diagrams* contains 48 rules, although the inclusion of large parts of the RGL for phrase formation pushes this number up to 251. Including the large-scale English dictionary inflates the grammar to 65,174 rules. As a comparison, a previous similar work on a CNL for the contract logic $\mathcal{CL}$ [4] had a GF grammar of 27 rules, or 2,987 when including a small verb lexicon.

3.5.2 Classification

Kuhn suggests the PENS scheme for the classification of CNLs [49]. We would classify the CNL presented in the current work as $P^5E^1N^{2\text{-}3}S^4$, F W D A. P (precision) is high since we are implementing a formal grammar; E (expressivity) is low since the CNL is restricted to the expressivity of the formalism; N (naturalness) is low as the overall structure is dominated with clause labels and bullets; S (simplicity) is high because the language can be concisely described as a GF grammar. In terms of CNL properties, this is a written (W) language for formal representation (F), originating from academia (A) for use in a specific domain (D).

The P, E and S scores are in line with the problem of verbalising a formal system. The low N score of between 2–3 is however the greatest concern with this CNL. This is attributable to a sentence structure is not entirely natural, somewhat idiosyncratic punctuation, and a bulleted structure that could restrict readability. While these features threaten the naturalness of the CNL in raw form, we believe that sufficiently developed editing tools have a large part to play in dealing with the structural restrictions of this language. Concretely, the ability to hide clause labels and fold away bulleted items can significantly make this CNL easier to read and work with.

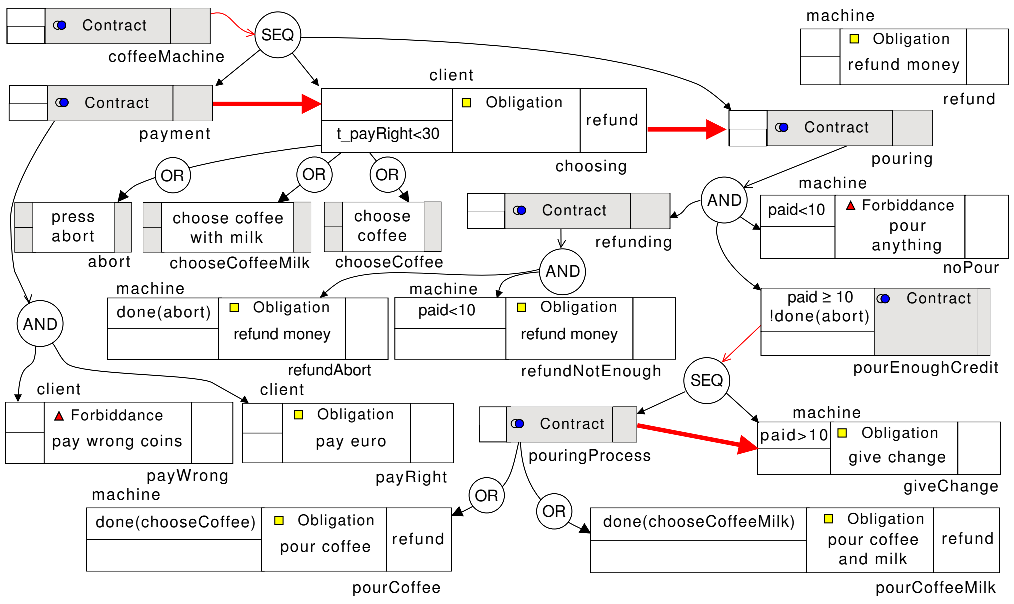


Figure 3.5: The complete *C-O Diagram* for the coffee machine example.

```
coffeeMachine : the following, in order
  - payment : payWrong : client mustn't pay wrong coins otherwise see refund and payRight : client
        ↪ is required to pay euro
  - choosing : when clock t_payRight less than 30 client is required
    - abort : to press abort , or
    - chooseCoffeeMilk : to choose coffee with milk , or
    - chooseCoffee : to choose coffee  otherwise see refund
  - pouring : all of
    - pourEnoughCredit : when abort is not done and variable paid not less than 10 first
        ↪ pouringProcess : pourCoffee : if chooseCoffee is done machine is required to pour
        ↪ coffee otherwise see refund or pourCoffeeMilk : if chooseCoffeeMilk is done machine is
        ↪ required to pour coffee and milk otherwise see refund , then giveChange : if variable
        ↪ paid greater than 10 machine is required to give change
    - noPour : if variable paid less than 10 machine mustn't pour anything
    - refunding : refundNotEnough : if variable paid less than 10 machine is required to refund
        ↪ money and refundAbort : if abort is done machine is required to refund money
refund : machine is required to refund money
```

Figure 3.6: The final verbalisation for the coffee machine example.

3.6 Related work

C-O Diagrams may be seen as a generalisation of $\mathcal{CL}$ [68, 69, 70] in terms of expressivity.[2] In a previous work, Angelov et al. introduced a CNL for $\mathcal{CL}$ in the framework AnaCon [4]. AnaCon allows for the verification of conflicts (contradictory obligations, permissions and prohibitions) in normative texts using the CLAN tool [31]. The biggest difference between AnaCon and the current work, besides the underlying logical formalism, is that we treat agents and actions as linguistic categories, and not as simple strings. This enables better agreement in the CNL which lends itself to more natural verbalisations, as well as making it easier to translate the CNL into other natural languages. We also introduce the special treatment of two-item co-ordination, and have a more general handling of lists as required by our more expressive target language.

Attempto Controlled English (ACE) [34] is a controlled natural language for universal domain-independent use. It comes with a parser to discourse representation structures and a first-order reasoner RACE [33]. The biggest distinction here is that our language is specifically tailored for the description of normative texts, whereas ACE is generic. ACE also attempts to perform full sentence analysis, which is not necessary in our case since we are strictly limited to the semantic expressivity of the *C-O Diagram* formalism.

Our CNL editor tool currently only has a basic user interface (UI). As already noted however, it is clear that UI plays a huge role in the effectiveness of a CNL. While our initial prototypes have only limited features in this regard, we point to the ACE Editor, AceRules and AceWiki tools described in [48] as excellent examples of how UI design can help towards solving the problems of writability with CNLs.

3.7 Conclusion

This work describes the first version of a CNL for the *C-O Diagram* formalism, together with web-based tools for building models of real-world contracts.

The spreadsheet format mentioned in Figure 3.2 was not covered in this paper, but we aim to make it another entry point into our system. This format shows the mapping between original text and formal model by splitting the relevant information about modality, agent, object and constraints into separate columns. As an initial step, the input text can be separated into one sentence per row, and for each row the remaining cells can be semi-automatically filled-in using

[2]On the other hand, $\mathcal{CL}$ has three different formal semantics: an encoding into the μ-calculus, a trace semantics, and a Kripke-semantics.

machine learning techniques. This will help the first part of the modelling process by generating a skeleton contract which the user can begin with.

We plan to extend the CNL and *C-O Diagram* editors with better user interfaces for easing the task of learning to use the respective representations and helping with the debugging of model errors. We expect to have more integration between the two applications, in particular the ability to focus on smaller subsections of a contract and see both views in parallel. sWhile the CNL editor already has basic input completion, it must be improvemed such that completion of functional keywords and content words are handled separately. Syntax highlighting for indicating the different constituents in a clause will also be implemented.

We currently use the RGL *as is* for parsing agents and actions without writing any specific constructors for them, which creates the potential for ambiguity. While this does not effect the conversion process, ambiguity is still an undesirable feature to have in a CNL. Future versions of the grammar will contain a more precise selection of functions for phrase construction, in order to minimise ambiguity.

Finally, it is already clear from the shallow evaluation in Section 3.5 that the CNL presented here suffers from some unnaturalness. This can to some extent be improved by simple techniques, such as adding variants for keywords and phrase construction. Other features of the *C-O Diagram* formalism however are harder to linearise naturally, in particular mandatory clause labels and arbitrarily nested lists of constraints and actions. We see this CNL as only the first step in a larger framework for working with electronic contracts, which must eventually be more rigorously evaluated through a controlled usability study.

Chapter 4

Modelling and analysis of normative texts with *C-O Diagrams*

John J. Camilleri, Filippo del Tedesco and Gerardo Schneider

Abstract. Our work is concerned with the formal analysis of *normative texts* such as terms of use, privacy policies, and service agreements. We begin by modelling such documents in terms of obligations, permissions and prohibitions of agents over actions, restricted by timing constraints. This is done using the *C-O Diagram* formalism, which we have extended syntactically and defined a new trace semantics for. We then describe our method for translating models in this formalism into networks of timed automata, for which we have a complete working implementation. By applying this approach to a real-world case study, we show the kinds of analysis possible through both syntactic querying on the structure of the model, as well as verification of properties using Uppaal.

Chapter contents

4.1 Introduction

We frequently encounter normative texts such as terms of use, software licenses and service-level agreements, and often accept these kinds of contractual agreements without really reading them. Writing and understanding such documents usually requires legal experts, and ambiguities in their interpretation are commonly disputed. Our goal is to model such texts computationally, helping the authorship process and enabling possibilities for querying and analysis which would benefit all parties involved.

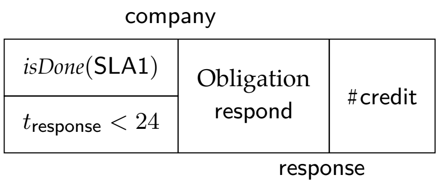
Formal analysis requires a formal language. Well-known generic formalisms such as first-order logic or temporal logic would not provide the right level of abstraction for a domain-specific task such as modelling normative documents. Instead, we choose to do this with a custom formalism based on the *deontic modalities* of **obligation**, **permission** and **prohibition**, and containing just the kinds of operators that are relevant to our domain.

Specifically, we use the *Contract-Oriented (C-O) Diagram* formalism [26], which provides both a logical language *and* a visual representation for modelling normative texts. This formalisation allows us to perform syntactic analysis of the models using predicate-based queries. Additionally, models in this formalism can be translated into networks of timed automata (NTA) which are amenable to model checking techniques, providing further possibilities for analysis.

Building such models from natural language texts is a non-trivial task which is currently done manually, and which therefore can benefit greatly from the right tool support. In previous work [16] we presented front-end user applications for working with *C-O Diagram* models both as graphical objects and through a controlled natural language (CNL) interface. The ability to work with models in different higher-level representations makes the formalism more attractive for real-world use when compared to other purely logical formalisms.

The current work is concerned with the back-end of this system, focusing on the details of the modelling language and how different kinds of analysis can be performed on these models. Concretely, the contributions of this paper are:

1. An extended definition of *C-O Diagrams* including syntactic extensions aimed at easing the modelling process (Section 4.2).
2. A novel trace semantics for these extended *C-O Diagrams* defined in terms of trace acceptance (Section 4.2.3).
3. A new translation function to NTA which fixes some issues with the previous translation (Section 4.3), and a proof of its correctness with respect to the trace semantics (Section 4.3.1).
4. The first full implementation of a system for working with *C-O Diagrams* and translating



The company must respond to an SLA1 request within 24 hours. If this target is not met, the customer is entitled to credit.

Figure 4.1: Example of a *C-O Diagram* box together with the natural language clause it models.

them into UPPAAL automata, written in Haskell.

5. A discussion of methods for syntactic querying and semantic property checking of normative texts modelled in this formalism (Section 4.4). These are demonstrated by applying our methods to a case study (Section 4.5).

We conclude with a comparison of some related work (Section 4.6) and a final discussion (Section 4.7).

4.2 A formalism for normative texts

The *C-O Diagram* formalism was introduced by Martínez et al. [51] as a means for visualising normative texts involving the modalities of obligation, permission and prohibition. They allow the representation of these norms for different agents and actions, as well as *reparations* when obligations and prohibitions are violated. The basic element in a *C-O Diagram* is the *box* (Figure 4.1), representing a simple clause. A box has four components (from top to bottom, left to right):

(i) *guards* specify the conditions for enacting the clause;
(ii) an *interval* restricts the time frame during which the clause must be satisfied;
(iii) the *propositional content* of a box specifies a modality applied over actions;
(iv) a *reparation*, if specified, refers to another clause that must be enacted if the main norm is not satisfied (a *prohibition* is violated or an *obligation* not fulfilled; there is no reparation for *permissions*).

Each box also has an *agent* indicating the performer of the action, and a unique *name* for referencing purposes. Boxes can be expanded by using three kinds of refinement: *conjunction*, *choice*, and *sequence*.

The formalism presented here adds a number of extensions to the definition given in [26] (for simplicity we continue to refer to this extended formalism as simply *C-O Diagrams*).

4.2.1 Formal syntax

Figure 4.2 shows the grammar of our extended *C-O Diagram* syntax. A **contract** specification is a forest of top-level clause trees, each of which is tagged as either *Main* (instantiated when the contract is executed) or *Aux* (instantiated only when referenced). A **clause** C is primarily a modal statement, expressing the **obligation** $O(\cdot)$, **permission** $P(\cdot)$, or **prohibition** $F(\cdot)$ of an agent over an action C_2, where the finite set of agents is $\mathcal{A}$. A clause can also be a refinement over sub-clauses using **conjunction** *And*, **sequence** *Seq* or **choice** *Or*. In either case, the clause is always given a unique name from a set of names $\mathcal{N}$ and an optional set of conditions. An action, as defined by C_2, may be a single atomic action from the set Σ, or a complex one obtained by conjunction, choice or sequence. A clause may also have a reparation R, specifying another clause to be enacted if the main part of the clause is not satisfied.

Conditions affect the applicability of a clause, and are defined as tuples of guards and intervals. A **guard** is a conjunction of variable and timing constraints which govern the enactment of a clause. An **interval** is a conjunction of only timing constraints, which govern the window in which a clause is applicable.

If we consider a finite set of integer variables $\mathcal{V}$, then a **constraint over variables** is a boolean formula of the form: $v \sim n$ or $v - w \sim n$, for $v, w \in \mathcal{V}$, $\sim \in \{<, =, >\}$ and $n \in \mathbb{Z}$. It can also be a predicate of the form $pred(name)$ where $pred \in \{isDone, isComplete, isSat, isVio, isSkip\}$, and $name \in \mathcal{N}$.

Similarly, considering a finite set of variables $\mathcal{C}$ standing for clocks, then a **timing constraint** is a conjunction of constraints of the form: $c \sim n$ or $c - d \sim n$, for $c, d \in \mathcal{C}$, $\sim \in \{<, =, >\}$ and $n \in \mathbb{Z}$. By convention, we assume that for every $name \in \mathcal{N}$ there is a clock in $\mathcal{C}$ with identifier t_{name}.

Well-formedness. Not all contracts which can be built from this grammar are considered valid. We define a *well-formed C-O Diagram* model to be one in which: (i) there is at least one main clause, (ii) all names are unique, (iii) all cross-references are valid, (iv) reparations and references do not lead to cycles, (v) clock names and predicates refer to existing boxes, and (vi) timing constraints in the interval can only refer to the current box.

4.2.2 Extensions to *C-O Diagrams*

This section describes the syntactic extensions we have made to the *C-O Diagram* language as it is presented by Díaz et al. [26]. The purpose behind these extensions is to help the modeller,

$$
\begin{aligned}
\mathit{Contract} &:= \left\{ \langle C, \mathit{Type} \rangle^{+} \right\} \text{ where } \mathit{Type} \in \{\mathit{Main}, \mathit{Aux}\} \\
C &:= \langle name, agent, Conditions, O(C_2), R \rangle \\
&\quad \mid \langle name, agent, Conditions, P(C_2) \rangle \\
&\quad \mid \langle name, agent, Conditions, F(C_2), R \rangle \\
&\quad \mid \langle name, Conditions, C_1, R \rangle \\
&\quad \mid \mathit{Ref} \\
C_1 &:= C\ (\mathit{And}\ C)^{+} \mid C\ (\mathit{Or}\ C)^{+} \mid C\ (\mathit{Seq}\ C)^{+} \\
C_2 &:= \mathit{action} \mid C_3\ (\mathit{And}\ C_3)^{+} \mid C_3\ (\mathit{Or}\ C_3)^{+} \mid C_3\ (\mathit{Seq}\ C_3)^{+} \\
C_3 &:= \langle name, C_2 \rangle \\
R &:= \mathit{Ref} \mid \top \mid \bot \\
\mathit{Ref} &:= \#name \\
Conditions &:= \langle \{Constraint^{*}\}, \{TimeConstraint^{*}\} \rangle \\
Constraint &:= v \sim n \\
&\quad \mid v - w \sim n \text{ where } v, w \in \mathcal{V} \cup \mathcal{C}, \sim \in \{<, =, >\} \text{ and } n \in \mathbb{Z} \\
&\quad \mid \mathit{isDone}(name) \mid \mathit{isComplete}(name) \mid \mathit{isSat}(name) \\
&\quad \mid \mathit{isVio}(name) \mid \mathit{isSkip}(name) \\
TimeConstraint &:= v \sim n \\
&\quad \mid v - w \sim n \text{ where } v, w \in \mathcal{C}, \sim \in \{<, =, >\} \text{ and } n \in \mathbb{Z}
\end{aligned}
$$

Figure 4.2: Extended version of the *C-O Diagram* grammar [26], defining the formal syntax of a contract, where $name \in \mathcal{N}$, $agent \in \mathcal{A}$ and $action \in \Sigma$.

by making common contract constructs naturally expressible in *C-O Diagrams* without requiring extra encoding.

Guards and intervals. We make a distinction between the conditions which determine the enactment of a clause ("guard"), and those which give a time range in which a clause may be completed ("interval"). The former give bounds on when the clause itself should be considered applicable, whereas the latter are applied only when the clause has been enacted.

Predicates as guards. In addition to variable comparisons, we also add a set of predicates over box and action names, which can be used as guards. They have been introduced into the syntax in order to help abstract away from low-level implementational details. They include $isDone$ which is true when an action has been performed or a clause has been satisfied, $isSat$ and $isVio$

which indicate if a clause has been satisfied or violated, $isSkip$ which is true when a clause was not enacted due to the guard not being satisfied, and $isComplete$ which is the disjunction of $isSat$ and $isSkip$.

Top/bottom reparations. When a clause has no reparation, we make a distinction between the trivially-satisfiable reparation $\top$ ("top") and the always unsatisfiable $\perp$ ("bottom"). Using the former means that even if a clause is violated, the contract may continue (though the violation will be recorded). This is the default when a reparation is not specified. The latter is used to indicate that a violation cannot be repaired.

Forests of clauses. Instead of modelling an entire contract as a single monolithic tree, we re-interpret a *C-O Diagram* as a set of clauses (or forest of trees) which are active in parallel. We allow a clause to be referred to from multiple parts of the contract, both within a guard and as a reparation. This follows the way in which most normative texts are written; with a generally flat structure, but with occasional references between clauses.

An additional *cross-reference operator* allows a single clause to be re-used, analogous to a subroutine. The name given as the reference must be a top-level clause in the contract. We also introduce a distinction between *main* and *auxiliary* top-level clauses in the contract forest, providing control over which clauses are instantiated when the contract is initialised.

4.2.3 Trace semantics

C-O Diagrams did not have a formal semantics: the "meaning" of *C-O Diagrams* were given by an encoding into network of timed automata [26]. We introduce here a completely new trace semantics for our extended formalism, beginning with the definition of a trace.

Definition 1. *An* event trace *(or simply* trace*) is a finite sequence of events $\sigma = [e_0, e_1, \ldots, e_n]$ where an event is a triple $e = \langle a, x, t\rangle$ consisting of an agent $a \in \mathcal{A}$, an action $x \in \Sigma$ and a time stamp t. The projection functions $agent(e)$, $action(e)$ and $time(e)$ extract the respective parts from an event.*

We give here some notation concerning traces: $\sigma(i)$ denotes the event at position i in trace σ, $\sigma(i..)$ denotes the finite subtrace starting at event in position i until the end of the trace, and $\sigma(..j)$ is the subtrace from the beginning of the trace to event $\sigma(j-1)$. Finally, $\sigma(i..j)$ is the subtrace between indices i and $j-1$.

The events in a trace are ordered by ascending time stamp value (earliest events first) and indexed from 0 onward. We say that a trace is *well-formed* iff $\forall i, j, 0 \leq i < n, i < j < n$:

$$\sigma \vDash \{Cl^1, \ldots, Cl^n\} \text{ iff } \bigwedge_{1 \leq i \leq n} \sigma \vDash^{\emptyset} C^i \text{ where } Cl^i = \langle C^i, \mathit{Main} \rangle \tag{4.1}$$

$$\sigma \vDash^c \langle n, a, c', O(C_2), R \rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash^{c \cup c'}_a C_2 \text{ or } \sigma \vDash^c R\right) \tag{4.2}$$

$$\sigma \vDash^c \langle n, a, c', P(C_2) \rangle \tag{4.3}$$

$$\sigma \vDash^c \langle n, a, c', F(C_2), R \rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash^{c \cup c'}_a C_2 \text{ implies } \sigma \vDash^c R\right) \tag{4.4}$$

$$\sigma \vDash^c \langle n, c', C_1, R \rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash^{c \cup c'} C_1 \text{ or } \sigma \vDash^c R\right) \tag{4.5}$$

$$\sigma \vDash^c C' \mathit{ And } C'' \text{ iff } \sigma \vDash^c C' \text{ and } \sigma \vDash^c C'' \tag{4.6}$$

$$\sigma \vDash^c C' \mathit{ Seq } C'' \text{ iff } \exists i : 0 < i < \mathit{length}(\sigma) \text{ and } \sigma(..i) \vDash^c C' \text{ and } \sigma(i..) \vDash^c C'' \tag{4.7}$$

$$\sigma \vDash^c C' \mathit{ Or } C'' \text{ iff either } \sigma \vDash^c C' \text{ or } \sigma \vDash^c C'' \tag{4.8}$$

$$\sigma \vDash^c_a x \text{ iff } \exists i : 0 < i < \mathit{length}(\sigma) \text{ and } \langle a, x, t \rangle = \sigma(i) \text{ and } \mathit{check}_i(c, t) \tag{4.9}$$

$$\sigma \vDash^c_a C'_3 \mathit{ And } C''_3 \text{ iff } \sigma \vDash^c_a (C'_3 \mathit{ Seq } C''_3) \mathit{ Or } (C''_3 \mathit{ Seq } C'_3) \tag{4.10}$$

$$\sigma \vDash^c_a C'_3 \mathit{ Seq } C''_3 \text{ iff } \exists i : 0 < i < \mathit{length}(\sigma) \text{ and } \sigma(..i) \vDash^c_a C'_3 \text{ and } \sigma(i..) \vDash^c_a C''_3 \tag{4.11}$$

$$\sigma \vDash^c_a C'_3 \mathit{ Or } C''_3 \text{ iff either } \sigma \vDash^c_a C'_3 \text{ or } \sigma \vDash^c_a C''_3 \tag{4.12}$$

$$\sigma \vDash^c_a \langle n, C_2 \rangle \text{ iff } \sigma \vDash^c_a C_2 \tag{4.13}$$

$$\sigma \vDash^c \top \tag{4.14}$$

$$\sigma \nvDash^c \bot \tag{4.15}$$

$$\sigma \vDash^c \#\mathit{name} \text{ iff } \sigma \vDash^c C' \text{ where } C' = \mathit{lookup}(\mathit{name}) \tag{4.16}$$

Figure 4.3: Trace semantics for our extended *C-O Diagram* language.

$time(\sigma(i)) \leq time(\sigma(j))$. The definition naturally extends to infinite traces, however we do not consider them here. We assume all our traces are well-formed.

The semantics of our language is defined in terms of the *respects* relationship ($\vDash$) between traces and contracts:

Definition 2. *We write* $\sigma \vDash K$ *to mean* trace σ respects the contract K *and* $\sigma \nvDash K$ *for* trace σ does not respect contract K. *The set of all traces respected by a contract defines its* trace semantics. *This relationship may be parametrised by a set of conditions c and an agent a, written as* $\vDash^c_a$.

The rules defining the *respects* relation are given in Figure 4.3. Actions are not consumed from the trace when they satisfy a particular clause. In other words, we do not recurse over the length of the trace, but rather over the structure of the *C-O Diagram*, considering the entire trace in each case. Each rule searches for the earliest event that satisfies it. Rules for sequential refinement (4.7 and 4.11) are the only ones that divide a trace into subtraces, as they enforce order.

The evaluation of clause constraints requires an environment Γ of variables and clocks. As these values may change over time, we model the environment as a function from a time stamp to set of valuations:

$$Env : TimeStamp \to (\mathcal{V} \cup \mathcal{C}) \to \mathbb{Z} \tag{4.17}$$

$$get : Env \to TimeStamp \to (\mathcal{V} \cup \mathcal{C}) \to \mathbb{Z} \tag{4.18}$$

$$set : Env \to TimeStamp \to (\mathcal{V} \cup \mathcal{C}) \to \mathbb{Z} \to Env \tag{4.19}$$

The environment can be queried via the get function (4.18) and updated using the set function (4.19), allowing clock resets and re-assignment of variables. An update affects all valuations from the given time stamp onward. Environment updates are not detailed in the trace semantics described in Figure 4.3, however they are an essential part of the operational definition of our language. Similarly, for the sake of clarity we do not explicitly mark the environment in the rules, though it is implied to be globally accessible. The environment also contains one clock never-reset t_0 which represents the "current time", such that $\forall t : get(\Gamma, t, t_0) = t$.

Checking *condition satisfaction* is defined in the $check_g$ function for guards (4.20) and the $check_i$ function for intervals (4.21). These look up the state of the environment Γ at the current time t_0 and return the conjunction of each of the individual boolean expressions in the respective part of the condition.

$$check_g\Big(\langle\{c_1,\ldots,c_n\},\_\rangle\Big) = \bigwedge_{1\leq j\leq n} eval(c_j, t_0) \tag{4.20}$$

$$check_i\Big(\langle\_,\{c_1,\ldots,c_n\}\rangle\Big) = \bigwedge_{1\leq j\leq n} eval(c_j, t_0) \tag{4.21}$$

$$eval\big(v \sim n, t\big) = get(\Gamma, t, v) \sim n \tag{4.22}$$

$$eval\big(v - w \sim n, t\big) = get(\Gamma, t, v) - get(\Gamma, t, w) \sim n \tag{4.23}$$

$$eval\big(isPred(name), t\big) = get(\Gamma, t, Pred\_name) \tag{4.24}$$

$$\text{where } Pred \in \{Done, Sat, Vio, Skip\}$$

$$\begin{aligned} eval\big(isComplete(name), t\big) &= eval\big(isSat(name), t\big) \\ &\vee eval\big(isSkip(name), t\big) \end{aligned} \tag{4.25}$$

The environment also contains a number of implicit boolean variables which represent the status of every box and action in a model, for example whether an action has been completed or a clause has been violated. The predicates listed in Section 4.2.2 are encoded as comparisons involving these variables.

Finally, we also need the function $lookup : Contract \to \mathcal{N} \to C$ for resolving references by name. This function searches recursively over the structure of the contract model, returning $\perp$ if not found.

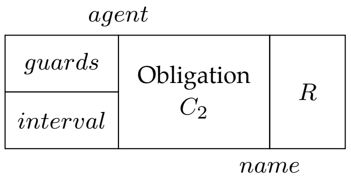
4.3 Translation to timed automata

In order to enable property-based analysis on these models, Díaz et al. [26] introduce a translation from *C-O Diagrams* into *Networks of Timed Automata* (NTAs). A *timed automaton* (TA) [1] is a finite automaton extended with clock variables which increase their value as time elapses, all at the same rate. These clocks can be used in guards on transitions and invariants on locations, and can be reset to zero during the execution of a transition. The model is also extended with clock constraints, which are conditions on the transitions that restrict the behaviour of the automaton. An *NTA* is a set of TAs which are run in parallel, using the same set of clocks. An NTA also defines a set of channels which allow synchronisation between automata.

Díaz et al. [26] describe a translation from *C-O Diagrams* into abstract NTA, followed by explanations of how these can be encoded in the UPPAAL tool [50]. In this work we present a revised translation function *trf* : *Contract* → *UPPAAL*, together with a full implementation in Haskell. Our translation contains a number of modifications which fix some encoding problems found in the original. As there is no difference in abstraction level between NTA and UPPAAL models, we skip the intermediate representation altogether and directly produce an UPPAAL model from a *C-O Diagram*.

Figure 4.4 shows a generic obligation clause together with the UPPAAL automata produced from its translation. Informally, this box is interpreted as follows: when $guards$ become true, $agent$ is obliged to do action C_2 within the time frame described by $interval$. Should they fail to do this, the reparation clause R will come into effect.

Our translation separates this single clause into two concerns: (i) the processing of the conditions which would enable the obligation, and (ii) the obligation itself. The former is handled by an automaton we call the *thread*, shown in Figure 4.4 (bottom left). The guard from the original clause is separated into lower and upper bound timing constraints, and variable constraints. First the lower bounds must be satisfied in order to progress in the automaton. The variable con-



$$C = \langle name, agent, \langle guards, interval\rangle, O(C_2), R\rangle$$
$$\text{where } guards = g_{low} \wedge g_{upp} \wedge g_{vars}$$

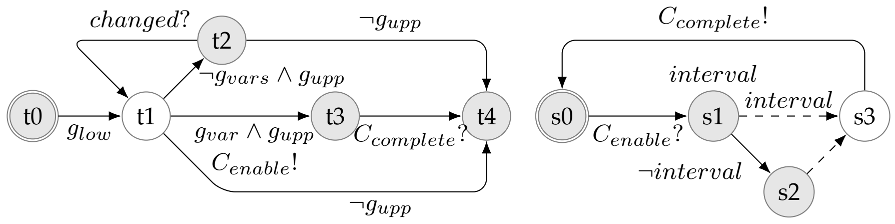


Figure 4.4: Simplified translation of an obligation clause (top) into two timed automata: the *thread* (left) and *main* automaton (right). The dashed edges $s1$–$s3$ and $s2$–$s3$ are filled with the translations of the complex action C_2, and of the reparation R, respectively. White nodes indicate committed locations.

straints g_{vars} are then actively checked within the given time window (until the expiration of the upper bounds), such that the main obligation is enabled as soon as the constraints are satisfied. This is achieved by having separate *check* and *wait* states ($t1$ and $t2$). The *check* state is committed, meaning that no time can pass while in this state. Whenever a variable changes value in the system, a broadcast signal is sent on the *changed* channel which causes the waiting automaton to re-check its constraints. When the constraints are met, the thread automaton transitions to $t3$, activating the main automaton.

Once activated, the main automaton may wait for as long as its intervals allow (enforced by an invariant on location $s1$). From here, either the top transition is taken before expiration, corresponding to the action being done, or the time expires and the lower path is taken, enacting the clause's reparation. Finally, the main automaton synchronises with the thread and enters the initial idle state (where it could possibly be re-triggered), while the thread automaton reaches a final end state. For further details of the translation, refer to Section 4.8.

4.3.1 Correctness of the translation

The previous section informally describes the translation function *trf*, which converts a *C-O Diagram* into an UPPAAL model. In order to trust any analysis performed on this translated model, we need to be certain that the translation itself is correct with respect to the trace semantics defined in Section 4.2.3. We approach this by discussing the relation of our trace semantics with those of UPPAAL.

David et al. [22] define a trace of an UPPAAL model as a sequence of *configurations*, where a configuration describes the current location of all automata in a system and gives valuations for all its variables and clocks. A *timed trace* is a trace which begins from an initial configuration and ends in a maximally extended one, where each consecutive configuration can be reached from its previous one in a single step (further details on this can be found in Section 4.8).

Let $\mathcal{T}_U(M)$ denote the set of *timed traces* for an UPPAAL model M. This set includes all timed traces which are either infinite or maximally extended (deadlocked). We are however interested in a subset of $\mathcal{T}_U(M)$, namely *finite* traces ending in a configuration which represents the completion of all top-level clauses in our contract K. We shall indicate this set with $\mathcal{T}_U^K(M)$.

Let us assume an abstraction function $\mathit{abstr} : \mathcal{T}_U \to \mathcal{T}$, which transforms an UPPAAL trace σ_U into an event trace σ by extracting the time stamps at which each action was performed. The following theorem then relates our trace semantics for *C-O Diagrams* with UPPAAL model traces:

Theorem 1. *Given a contract K and its translation into an UPPAAL model $M = \mathit{trf}(K)$, for every σ in $\mathcal{T}$ it is the case that:*

$$\sigma \vDash K \;\; \mathit{iff} \;\; \exists\, \sigma_U \in \mathcal{T}_U^K(M) : \sigma = \mathit{abstr}(\sigma_U)$$

Proof sketch

The proof is performed by structural induction over the *C-O Diagram* syntax (see Figure 4.2). In each case, we consider its translation into an UPPAAL model by the *trf* function. Using the formalisation of UPPAAL models and their trace semantics given by David et al. [22], we then characterise the set of UPPAAL traces which represent the satisfaction of the case we are modelling. We then show how this set of UPPAAL traces is related to the event traces which would respect the original clause, effectively characterising the *abstr* function. Further details of this proof are included in Section 4.8. □

As a corollary of Theorem 1, we have that the translation is sound.

Corollary 1. *The translation function trf is sound with respect to the trace semantics as defined in Section 4.2.3.*

4.4 Analysis

The main purpose of formalising normative texts is to be able to perform automated analysis, by which we mean running queries of various kinds against our model. This of course pre-supposes

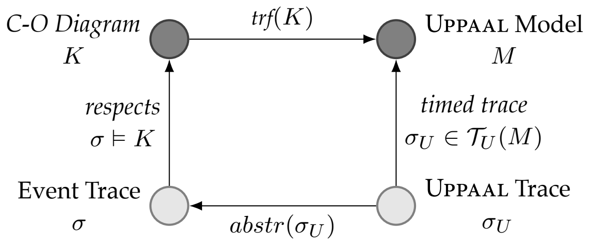


Figure 4.5: Different representations in our framework and the relations between them.

that the model is an accurate representation of the original text. Some queries can be checked at a syntactic level, such as checking if a normative text contains any permissions for a particular agent or identifying obligations without constraints or reparations. We refer to these as *syntactic properties* as they can be computed purely syntactically on the model.

Other properties cannot be tested in this way, such as checking whether a clause may be enacted within a certain amount of time. This may depend on a previous sequence of events, and determining whether these events could happen or not cannot be inferred from the syntax alone. We call these *semantic properties*, and verify them by converting our models to timed automata and using model checking techniques.

4.4.1 Syntactic analysis

We begin by introducing *predicates* over single clauses, which are the building blocks for defining syntactic properties. The predicate $\mathit{isObl}(C)$ for example is true if the clause C is an obligation. Predicates may also take additional arguments, such as $\mathit{agentOf}(a, C)$, which is true if agent $a \in \mathcal{A}$ is responsible for clause C. The predicates here cover general properties over clauses, which can be used as building blocks for a general property language:

$$
\begin{aligned}
\mathit{isObl}, \mathit{isFor}, \mathit{isPer}(C) &\text{ iff clause } C \text{ is an obligation, prohibition, permission}\\
\mathit{isAnd}, \mathit{isOr}, \mathit{isSeq}(C) &\text{ iff clause } C \text{ contains conjunction, choice, sequence}\\
\mathit{hasUppBound}(C) &\text{ iff clause } C \text{ has an upper bound in its interval}\\
\mathit{agentOf}(a, C) &\text{ iff agent } a \text{ is responsible for clause } C\\
\mathit{repair}(r, C) &\text{ iff } (\mathit{isObl}(C) \text{ or } \mathit{isFor}(C)) \text{ and } r \text{ is a reparation of } C
\end{aligned}
$$

The syntactic properties defined for single clauses can also be extended to contract specifications as a whole. In this way we can, for example, collect all the obligations contained in a

$$\mathcal{Q} : (C \to Boolean) \to Contract \to \mathcal{P}(C) \tag{4.26}$$

$$\mathcal{Q}(\psi, \{\langle C^1, T^1\rangle, \ldots, \langle C^n, T^n\rangle\}) = \bigcup_{1\leq i\leq n} \begin{cases} \{C^i\} & \text{if } \psi(C^i) \\ \emptyset & \text{otherwise} \end{cases} \tag{4.27}$$

Figure 4.6: Definition of the $\mathcal{Q}$ operator for querying a contract.

contract. We refer to syntactic properties that apply to contract specifications as *queries*, since they are the result of querying a contract with clause properties. The $\mathcal{Q}$ function, defined in Figure 4.6, returns the set of all clauses in the contract that satisfy the predicate provided as the first argument. This has also been implemented as a Haskell function and command-line program, together with the translation function from the previous section.

4.4.2 Semantic analysis

Syntactic analysis alone cannot be used to answer queries about the possibility of certain situations arising within a contract. Doing this involves taking into account the conditions applied to each box, as well as a possible trace of previous events. These kinds of *semantic properties* are computed by first translating a contract model into a network of timed automata (NTA) and then applying model checking techniques. This translation is based on that introduced in [26], yet we have made a number of changes to it in order to fix problems discovered in the original definition, as well as to match our updated syntax. We have a fully working implementation of this translation function, written as a Haskell program which takes a *C-O Diagram* as input and produces an UPPAAL file.

Given a contract and a property, we compute the UPPAAL representation of the contract using the function *trf* (Section 4.3), and encode the property in UPPAAL's specification language, which is a subset of TCTL [8]. We then use UPPAAL to verify whether the property holds (is satisfiable) against the model. Any formula that can be expressed within the UPPAAL language can be interpreted as a semantic property.

Possibility. The query $E\Diamond\psi$ is satisfied if there is any sequence through the automaton where the expression ψ is true. We call this a *possibility* property. From a contract perspective, such a property checks whether the set of prescriptions contained in the contract make it possible for ψ to happen.

1.3 Customer may initiate a request for Standard Support via the technical helpdesk. A Support Request must include the following information: (i) type of service, (ii) details for contacting the Customer, and (iii) a clear description of Support required. Company may refuse a Support Request if it is unable to establish that the Support Request is made by an authorised person.

1.4 The table below sets forth the Response Time for any request for Support made in accordance with Section 1.3 above. The Response Time Target depends on the SLA level that the Customer has chosen.

SLA Level	**Response Time Target**
Basic	24 hours
Bronze	4 hours

1.5 In the event Company does not respond within the applicable Response Time Target, Customer shall be eligible to receive a Service Credit. If Customer does not pay a Monthly Recurring Charge then Customer shall not be eligible to any Response Time Credit.

1.6 Customer shall ensure that it will at all times be reachable on Customer's emergency numbers, specified in the Customer Details Form. No Response Time Credit shall be due in case the Customer is not reachable.

Figure 4.7: Extract from the SLA from LeaseWeb Inc. covering hosting services.

Invariance. Properties dealing with *invariance* are satisfied if the expression in question holds at all locations in the evolution of the system. Such queries are specified using the $A\square$ operator.

4.5 Case study

As a case study for applying these methods we choose a service level agreement (SLA) from the hosting company LeaseWeb Inc.[1] This 6-page document is divided into 6 chapters and 59 sections, many of which consist of multiple sentences. We have so far modelled one chapter of this agreement (7 sections), but for space reasons present here an abridged version of this chapter (Figure 4.7).

4.5.1 Model

The task of building a *C-O Diagram* model from this text is a manual one, which can be done using the front-end tools introduced in previous work [16]. In place of a diagram, we provide the model for this snippet as a formula in our language (Figure 4.8). The contract is built from main and auxiliary clauses, linked together using cross-referencing. The primary clause is `request`, which

[1] `https://www.leaseweb.com/legal`

$$
\begin{aligned}
K = \{ & \big\langle \langle \mathsf{request}, \epsilon, \#\mathsf{req\_type}\ \mathit{Seq}\ \#\mathsf{req\_info}\ \mathit{Seq}\ \#\mathsf{resp}, \top \rangle, \mathit{Main} \big\rangle, \\
& \big\langle \langle \mathsf{req\_type}, \mathsf{customer}, \epsilon, P(\mathsf{standard\ support}) \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{req\_info}, \mathsf{customer}, \epsilon, O(C_2), \top \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{cust\_auth}, \mathsf{customer}, \epsilon, O(\mathsf{prove\ authorisation}), \top \rangle, \mathit{Main} \big\rangle, \\
& \big\langle \langle \mathsf{req\_refuse}, \mathsf{company}, \langle \neg isDone(\mathsf{cust\_auth}), \epsilon \rangle, P(\mathsf{refuse\ request}) \rangle, \mathit{Main} \big\rangle, \\
& \big\langle \langle \mathsf{chooseSLA}, \mathsf{customer}, \epsilon, P(\langle \mathsf{sla1}, \mathsf{basic} \rangle\ \mathit{Or}\ \langle \mathsf{sla2}, \mathsf{bronze} \rangle) \rangle, \mathit{Main} \big\rangle, \\
& \big\langle \langle \mathsf{resp}, \epsilon, \#\mathsf{resp1}\ \mathit{And}\ \#\mathsf{resp2}, \top \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{resp1}, \mathsf{company}, \langle isDone(\mathsf{sla1}), t_{\mathsf{resp1}} < 24 \rangle, O(\mathsf{respond}), \#\mathsf{credit} \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{resp2}, \mathsf{company}, \langle isDone(\mathsf{sla2}), t_{\mathsf{resp2}} < 4 \rangle, O(\mathsf{respond}), \#\mathsf{credit} \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{credit}, \mathsf{company}, \langle isDone(\mathsf{reach}) \wedge isDone(\mathsf{monthly}), \epsilon \rangle, O(\mathsf{give\ credit}), \top \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{monthly}, \mathsf{customer}, \epsilon, P(\mathsf{pay\ monthly}) \rangle, \mathit{Main} \big\rangle, \\
& \big\langle \langle \mathsf{reach}, \mathsf{customer}, \epsilon, O(\mathsf{be\ reachable}), \top \rangle, \mathit{Main} \big\rangle \} \\
C_2 = {} & \langle \mathsf{ri1}, \mathsf{service\ type} \rangle\ \mathit{And}\ \langle \mathsf{ri2}, \mathsf{contact\ details} \rangle\ \mathit{And}\ \langle \mathsf{ri3}, \mathsf{problem\ description} \rangle
\end{aligned}
$$

Figure 4.8: The *C-O Diagram* model of the case study (Figure 4.7) in formal language syntax.

we model as a sequence of clauses governing the initiation of the request (req_type), the details required (req_info), and the response obligations from the company (resp).

The response time targets for dealing with customer requests are described in clause 1.4. Each SLA level is treated individually (resp1 and resp2), both depending on which level has been chosen by the customer in chooseSLA, using the $isDone$ predicate as a guard. These targets are encoded as intervals, e.g. $t_{\mathsf{resp1}} < 24$ enforcing that the response is completed within 24 hours.

Clause 1.5 says that the customer is entitled to credit when the company fails to respond within their target time. This is a typical example of a reparation. We model this as the clause credit, which is given as the reparation for both resp1 and resp2. The guards in this reparation restrict the situations in which credit can be given. Finally, the requirement for the customer to be reachable (clause 1.6) is encoding as a standalone obligation reach, which is also given as a guard to the credit clause.

The model described here contains 2 agents, 12 actions and 12 boxes (clauses). The UPPAAL system produced from its translation consists of 13 processes, 113 locations, and 122 transitions. It uses 15 channels, 30 clocks, and 116 boolean variables.

4.5.2 Syntactic analysis

Missing reparations. We can identify clauses in our contract with potentially problematic characteristics by inspecting the model syntactically. For example, the following query returns all clauses with no reparation:

$$\mathcal{Q}(\mathit{repair}(\top), K) = \{\mathsf{request}, \mathsf{req\_info}, \mathsf{resp}, \mathsf{cust\_auth}, \mathsf{credit}, \mathsf{reach}\}$$

As $\top$ is the default reparation, it is no surprise that most clauses are returned here. However this is valuable first step in identifying clauses which can be violated without any repercussions. If the company doesn't honour its promise to credit the customer, for example, there should surely be some recourse for this. By taking the names in the query response and tracing them back to original text, we find that all clauses except 1.4 in fact contain under-specified reparations.

Unbounded obligations. As a second example, we may wish to list all obligations without an upper bound in their interval:

$$\mathcal{Q}(\mathit{isObl} \wedge \neg \mathit{hasUppBound}, K) = \{\mathsf{req\_info}, \mathsf{cust\_auth}, \mathsf{credit}, \mathsf{reach}\}$$

Even if the company may be obliged to credit the customer, without any time constraints they can effectively avoid doing this. Though it is common for normative documents such as this to contain clauses without specific time restrictions, this is a common source of problems when it comes to their formalisation. Even though the clause names in the response are different from those in the previous example, they still correspond to the same clauses from the original text. This is because the clauses in the model are more fine-grained than those in the text, where each clause contains significant information in multiple sentences.

Possible choices. As a final example, we may wish to filter out the clauses in the contract which provide a choice to the customer, using the following query:

$$\mathcal{Q}(\mathit{isOr} \wedge \mathit{agentOf}(\mathsf{customer}), K) = \{\mathsf{chooseSLA}\}$$

This query returns a single clause chooseSLA, indicating the customer's choice of service level, as described in clause 1.4. As we can see here, these predicates can be combined in multiple ways to produce different kinds of syntactic queries on our contract models. The execution of these queries is very quick, and linear in the size of the model.

4.5.3 Semantic analysis

Consider the last sentence of clause 1.3 in Figure 4.7. We would like to use verification to check whether it is possible for the company to refuse the request, even though the customer has successfully identified themselves. This can be expressed with the following query:

$$E\Diamond\ \mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{cust\_auth}) \land \mathit{isDone}(\mathsf{req\_refuse})$$

Running this on our model with the UPPAAL model checker returns SAT in under 1 second, together with an example trace. Despite the guard on the req_refuse clause, the trace shows that it is possible for the request to be refused *before* the customer has a chance to authorise. This shows how under-specification of the timing constraints between related clauses can lead to undesirable situations.

We can attempt to fix the contract by adding a window of 2 hours for customers to authenticate themselves. This can be encoded by adding the guard $t_{\mathsf{cust\_auth}} > 2$ to clause req_refuse. We then update the query accordingly:

$$E\Diamond\ \mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{cust\_auth}) \land \mathit{Clocks}[\mathsf{cust\_auth}] < 2 \land \mathit{isDone}(\mathsf{req\_refuse})$$

Running this query now takes considerable longer (approx. 3 mins), though we now get UNSAT which verifies that the problem case is no longer possible. The query can also be reformulated as an invariant, for which we get SAT within a similar amount of time:

$$A\Box\ \mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{cust\_auth}) \land \mathit{Clocks}[\mathsf{cust\_auth}] < 2 \implies \neg\mathit{isDone}(\mathsf{req\_refuse})$$

When it comes to giving service credit to the customer, we may want to verify that this can also only occur when we intend. Considering the basic support level, we come up with the following pair of queries:

$$\begin{aligned}A\Box\ &\mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{resp1}) \land \mathit{isDone}(\mathsf{monthly}) \land \mathit{isDone}(\mathsf{reach})\\ &\land \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] > 24 \implies \mathit{isDone}(\mathsf{credit})\end{aligned}$$

$$\begin{aligned}A\Box\ &\mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{resp1})\\ &\land \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] < 24 \implies \neg\mathit{isDone}(\mathsf{credit})\end{aligned}$$

Note how we used the difference between two clocks to determine the relative time at which the response occurred. Running both queries returns a Sat result as expected, taking around 4 mins each to complete.

Execution times. The model checking times presented here are rather low, because they refer to the small model snippet presented in Figure 4.8. These times increase dramatically however as we model increasingly larger fragments of the case study text.

4.6 Related work

C-O Diagrams were introduced by Martinez et al. in [51], and further refined in [26]. While our work is heavily based on their formalism, we make a number of extensions to it. Most significantly, Díaz et al. convert guards in *C-O Diagrams* quite literally into edge guards in the resulting NTA. When a guard is true, a *possible* path is enabled in the automaton, but there is nothing forcing this path to be taken. Our translation has a stricter interpretation of guards, ensuring that *if* a guard becomes true during the specified time frame, then the resulting path *must* be taken. Our other extensions to the formalism were introduced to make the modelling task more natural. The trace semantics defined in Section 4.2.3 is also completely new for the *C-O Diagram* formalism. We follow the approach of [30] where a trace semantics is defined for the $\mathcal{CL}$ language [69]. Also, unlike [51, 26] we have a full working implementation of the translation of *C-O Diagrams* into Uppaal (no implementation existed before) allowing us to perform model checking, as well as an implementation for performing syntactic queries.

Angelov et al. [4] introduce a similar framework AnaCon for the analysis of contracts, based on the contract logic $\mathcal{CL}$. Their system allows for the detection of contradictory clauses in normative texts using the CLAN tool [31]. In comparison, the underlying logical formalism we use includes timing aspects which provides a whole new dimension to the analysis. Besides this, our translation into Uppaal allows for checking more general properties, not only normative conflicts.

Earlier work by current authors discusses the front-end of this system [16]. This includes tools for working with contracts represented diagrammatically, and the definition of a controlled natural language (CNL) which can be used as both a source and a target interface for contracts modelled in this formalism.

In their work, Pace and Schapachnik introduce the Contract Automata formalism [62] for modelling interacting two-party systems. Their approach is similarly based on deontic norms, but with a strong focus on synchronous actions where a permission for one party is satisfied together with a corresponding obligation on the other party. Their formalism is limited to strictly two parties, and does not have any support for timing notions as *C-O Diagrams* do.

Much work on analysing texts in general exists in the Natural Language Processing community. However, this is limited to processing text for the sake of analysing linguistic aspects (e.g., co-reference resolution [11]) and thus not directly comparable with our approach. This kind of work would be useful to us as a pre-processing step, helping to bridge the gap between natural language text and our formal language. The exceptions to the above are maybe some specific works using passage retrieval techniques, and the work by the Attempto group with controlled natural languages, discussed below.

Rosso et al. [73] use passage retrieval techniques to analyse legal texts. Their analysis, however, cannot handle verification of timing constraints as we do, since their technique is limited to very specific kinds of (untimed) conflicts.

Attempto Controlled English (ACE) [34] is one of the best examples of a controlled natural language (CNL) designed for universal use. It comes with a parser to discourse representation structures (a syntactic variant of first-order logic) and a first-order reasoner RACE [33], and has been used in a variety of applications. The biggest distinction here is that our analysis is specifically concerned with timing constraints, whereas to our knowledge there is no such capability with RACE.

4.7 Conclusion

This work presents a number of extensions to the *C-O Diagrams* formalism for normative texts, together with a new translation to UPPAAL models and a fully working implementation in Haskell. We have provided a novel trace semantics for our language, defining what it means for a trace of events to respect a contract specification, and proved the correctness of the translation with respect to the trace semantics. We have also defined and implemented algorithms for performing both syntactic and semantic queries, the former being an *ad hoc* implementation while the latter uses the UPPAAL model checker.

Scalability. Space limitations prevent us from providing a larger case study, though it is significant enough to display the details of the approach. It is well-known that model checking may easily become intractable for non-trivial models, and verification time is very sensitive to the size of the automata and the use of channel synchronisations. Our translation is currently not optimised and may as such produce unnecessarily large/many automata. A thorough investigation of possible optimisations and their effect on performance is regarded as important future work, but outside the scope of the present paper.

It would also be relevant to work out which clauses in a model are independent of each other, as this could be used to exclude parts of the model when model checking and thus reduce verification time. We also point out that scalability is not an issue for the syntactic analysis, which is linear in the size of the model.

Future work. While we have only presented a single case study here, these methods have also been applied to various other real-world examples of normative texts to guide to design process.

We show here that analysis of normative texts is possible with the right formalisation and querying system. In our work so far, the formalisation of both contract and queries is still a manual task. Working towards a higher level of automation in this process reduces the workload for the user, and also indirectly creates a higher level of predictability.

This work forms the core of a larger toolkit for working with normative texts. On the front-end, we already have tools for building contract models graphically and using controlled natural language (CNL). We are currently working on applying NLP techniques for producing partial models from natural language documents, to ease the modelling burden on the user [18].

4.8 Appendix: Translation to NTA & proof of correctness

4.8.1 Outline

We prove here the correctness of our translation function to UPPAAL models with respect to the trace semantics for *C-O Diagrams* defined in Section 4.2.3. We do this by structural induction over the syntax in Figure 4.2, in each case considering the translated UPPAAL model obtained from the *trf* function and comparing the sets of traces which are allowed by our trace semantics and by UPPAAL's.

4.8.2 UPPAAL trace semantics

David et al. [22] give a formalisation for UPPAAL models, together with a definition of their trace semantics. We briefly repeat their definitions here.

Definition 3 (UPPAAL process). *An UPPAAL process A (single automaton) is a tuple $\langle L, T, Type, l^0 \rangle$, where*

1. *L is a set of locations,*
2. *T is a set of transitions between two locations, each containing optionally a guard g, synchronisation label s and assignment a,*
3. *$Type$ is a typing function which marks each location as ordinary, urgent or committed, and*
4. *$l^0 \in L$ is the initial location.*

Definition 4 (UPPAAL model). *An UPPAAL model M (network of automata) is a tuple $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$, where*

1. *$\vec{A}$ is a vector of processes $A_1, \ldots, A_n$;*
2. *$Vars$ is a set of variables,*
3. *$Clocks$ is a set of clocks,*
4. *$Chan$ is a set of synchronisation channels, and*
5. *$Type$ is a polymorphic typing function for locations, channels, and variables.*

Definition 5 (Configuration). *A configuration of an UPPAAL model is a triple $(\vec{l}, e, v)$, where*

1. *$\vec{l} = (l_1, \ldots, l_n)$ where $l_i \in L_i$ is a location of process A_i,*
2. *e is a valuation function mapping every variable to an integer value, and*
3. *v is a valuation function mapping every clock to a non-negative real number.*

Definition 6 (Simple action step). *For a configuration $(\vec{l}, e, v)$ a simple action step is enabled if there exists a transition $l \xrightarrow{g,a} l'$ such that*

1. *$l \in \vec{l}$,*
2. *its guards g evaluate to true given e, v,*
3. *the invariant on l' will hold after assignment a, and*
4. *if any other locations in $\vec{l}$ are committed, then l is also committed.*

Definition 7 (Synchronised action step). *For a configuration $(\vec{l}, e, v)$ a synchronised action step is enabled iff for a channel b there exist two transitions $l_i \xrightarrow{g_i, b!, a_i} l'_i$ and $l_j \xrightarrow{g_j, b?, a_j} l'_j$ such that*

1. *$l_i, l_j \in \vec{l}$ and $i \neq j$,*
2. *the guards $g_i \wedge g_j$ evaluate to true given e, v,*
3. *the invariants on l'_i and l'_j will hold after assignments a_i and a_j, and*
4. *if any other locations in $\vec{l}$ are committed, then l_i and/or l_j are also committed.*

Definition 8 (Delay step). *For a configuration $(\vec{l}, e, v)$ a delay step is enabled iff*

1. *none of the locations in $\vec{l}$ is urgent or committed,*
2. *no synchronised actions steps are enabled on channels marked as urgent, and*
3. *the invariants on all locations in $\vec{l}$ will still hold after the delay.*

Definition 9 (Timed trace). *A sequence of configurations $\{(\vec{l}, e, v)\}^K$ of length $K \in \mathbb{N} \cup \{\infty\}$ is a* timed trace *for a* UPPAAL *model M if*

1. *all locations in configuration 0 are the initial locations for their respective processes,*
2. *all variables and clocks evaluate to 0 in configuration 0,*
3. *if the sequence is finite, then at the last configuration no further steps are enabled (system is maximally extended/deadlocked),*
4. *if the trace is infinite, eventually every clock value exceeds every bound, and*
5. *every pair of consecutive configurations in the sequence are connected by a simple action step, synchronised action step, or delay step.*

4.8.3 Notes and notation

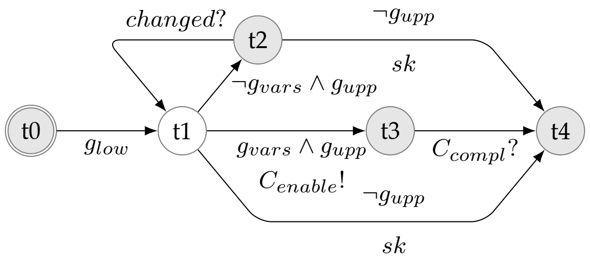
In each case of the proof, we present the automata resulting from the translation in graphical form, simply because they are more concise and easier to read than formulae. Similarly, details about variable and channel declarations are omitted for brevity. The following is a legend to the conventions we use:

1. Initial nodes are drawn with a double border.
2. Committed nodes are shown in white.

3. Dashed lines indicate edges that are to be filled in recursively.
4. A guard is split up into
   (a) lower-bound time constraints g_{low} (i.e. using the greater-than operator)
   (b) upper-bound time constraints g_{upp} (i.e. using the less-than operator)
   (c) non-temporal constraints g_{vars}
5. We use int to indicate the interval component of a set of conditions.
6. The symbol $\neg$ indicates the negation of constraints.
7. Constraints on a node indicate invariants.
8. The function calls $reset(name)$, $vio(name)$, $done(name)$, $sat(name)$, and $skip(name)$ are abbreviated to r, v, d, s, sk respectively, where $name$ is the name of the current box.
9. We use the term *end of time* to mean a time stamp value which is sufficiently large to be later than all events in the trace and all constraints in the model.

4.8.4 Thread automaton

All top level clauses (cases 4.2–4.8) may contain conditions which govern their enactment. As the translation of this logic into automata is identical for all clauses, we use a standard automaton model called the *thread* (shown below).



The thread starts the main automaton corresponding to the original clause via channel synchronisation on C_{enable}. Its structure ensures that the main automaton is guaranteed to be activated if and when the guard g_{vars} becomes true within the time frame specified by g_{low} and g_{upp}. When any of these is missing, it is replaced with a trivial condition $true$. Each time a variable in the system is updated, there is a synchronisation action on the broadcast channel *changed*, which causes all waiting threads to re-check their guards. If the time window expires without the guards becoming true, the main automaton is never enacted but instead skipped. There are various cases to consider here:

(a) g_{low} is false: Wait until g_{low} is true (must happen eventually).

(b) g_{vars} is immediately true: Transition to $t3$, activating main automaton.
(c) g_{vars} becomes true before g_{upp} expires: Wait in $t2$ until g_{vars} changes, then transition to $t1$ and then to $t3$, activating main automaton.
(d) g_{vars} becomes true, but after g_{upp} expires: Wait in $t2$ until g_{vars} changes, then transition to $t1$ and then to $t4$, skipping main automaton.
(e) g_{vars} never becomes true: Wait in $t2$ until g_{upp} expires, then transition to $t4$, skipping main automaton.

4.8.5 Case analysis

Note that the case numbers here correspond to the rules in the trace semantics in Section 4.2.3 (Figure 4.3).

Case 4.1: Contract.

$$\sigma \vDash \{Cl^1, \ldots, Cl^n\} \text{ iff } \bigwedge_{1 \leq i \leq n} \sigma \vDash^{\emptyset} C^i \text{ where } Cl^i = \langle C^i, \mathit{Main} \rangle$$

Event traces. Traces respecting this formula must respect each of the individual clauses independently.

Translation. Each *main* clause in a contract is translated into an automaton which is instantiated as a process in the UPPAAL model.

UPPAAL traces. Traces satisfying this model must contain configuration steps that take each individual process representing clause $name$ from its initial state to one in which no further steps are possible, and in which $isComplete(name)$ is true.

Argument. In both formalisms it is required that the trace must satisfy all clauses individually and concurrently.

Case 4.2: Obligation.

$$\sigma \vDash^{c} \langle n, a, c', O(C_2), R \rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash_{a}^{c \cup c'} C_2 \text{ or } \sigma \vDash^{c} R\right)$$

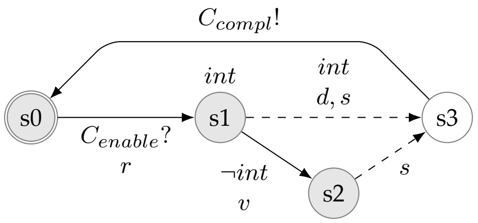
Event traces. Traces must contain events respecting C_2 while the conditions c' hold, or respecting R. We consider the following cases:
(a) Guards from combined conditions $c \cup c'$ are never true in the trace: the obligation is not enacted and thus trivially respected.

(b) Guards from combined conditions $c \cup c'$ become true in the trace: the obligation is enacted and can be respected in one of two ways:

   i. The actions in C_2 are performed by agent a at times which satisfy combined conditions $c \cup c'$.

   ii. The entire reparation clause R is completed while the inherited conditions c hold.

Translation. All automata from the translation of R, one thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s1 \rightarrow s3$ is filled with the translation of C_2 and $s2 \rightarrow s3$ is filled with the thread from the translation of R.



UPPAAL traces. In order to reach a state where the obligation is complete, a transition marked with s (satisfied) or sk (skipped) must be taken. This may happen in the following ways:

(a) The thread automaton ends up in $t4$ by skipping the main automaton.

(b) The thread automaton enables the main automaton, one of the following occurs:

   i. Transition $s1 \rightarrow s3$ is taken while interval int holds, respecting the translation of C_2.

   ii. Interval int expires and $s3$ is reached via $s2$, respecting the translation of R.

   Finally both automata synchronise on C_{compl} reaching maximally extended states.

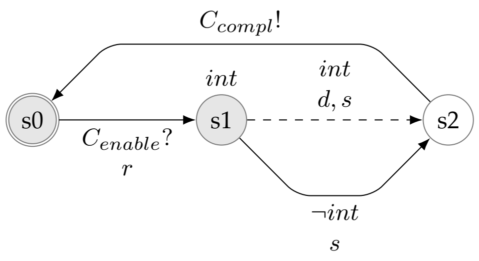
Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then either C_2 is respected while the conditions c' hold, or R is respected.

Case 4.3: Permission.

$$\sigma \vDash^c \langle n, a, c', P(C_2) \rangle$$

Event traces. Any trace will respect a permission.

Translation. One thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s1 \rightarrow s2$ is filled with the translation of C_2.



UPPAAL traces. In order to reach a state where the permission is complete, a transition marked with s (satisfied) or sk (skipped) must be taken. This may happen in the following ways:

(a) The thread automaton ends up in $t4$ by skipping the main automaton.

(b) The thread automaton enables the main automaton, one of the following occurs:

  i. Transition $s1 \to s2$ is taken while interval int holds, respecting the translation of C_2.

  ii. Interval int expires and $s2$ is reached via the lower transition. If no interval exists, the automaton will take this transition at the *end of time*.

  Finally both automata synchronise on C_{compl} reaching maximally extended states.

Argument. As any event trace is accepted, so is any UPPAAL trace which satisfies our basic conditions for completion.

Case 4.4: Prohibition.

$$\sigma \vDash^{c} \langle n, a, c', F(C_2), R\rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash_a^{c\cup c'} C_2 \text{ implies } \sigma \vDash^{c} R\right)$$

Event traces. If traces contain events respecting C_2 while the conditions c' hold then they must also respect R. We consider the following cases:

(a) Guards from combined conditions $c \cup c'$ are never true in the trace: the prohibition is not enacted and thus trivially respected.

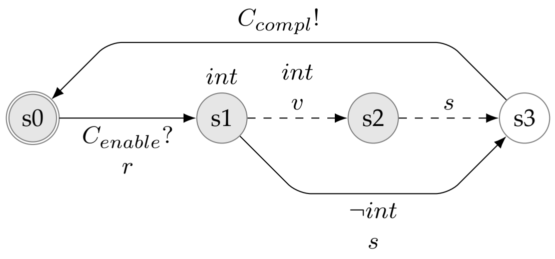
(b) Guards from combined conditions $c \cup c'$ become true in the trace: the prohibition is enacted and can be respected in one of two ways:

  i. The actions in C_2 are performed by agent a at times which satisfy combined conditions $c \cup c'$, followed by reparation clause R being completed while the inherited conditions c hold.

  ii. No action is taken until the conditions expire.

Translation. All automata from the translation of R, one thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s1 \to s2$ is filled with the translation of C_2 and

$s2 \to s3$ is filled with the thread from the translation of R.



UPPAAL traces. In order to reach a state where the prohibition is complete, a transition marked with s (satisfied) or sk (skipped) must be taken. This may happen in the following ways:

(a) The thread automaton ends up in $t4$ by skipping the main automaton.

(b) The thread automaton enables the main automaton, one of the following occurs:

i. Transition $s1 \to s2$ is taken while interval int holds, respecting the translation of C_2, followed by transition $s2 \to s3$, respecting the translation of R,

ii. Interval int expires and transition $s1 \to s3$ is taken.

Finally both automata synchronise on C_{compl} reaching maximally extended states.

Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then when C_2 is respected while the conditions c' hold, then R must necessarily be respected too.

Case 4.5: Refinement.

$$\sigma \vDash^{c} \langle n, c', C_1, R\rangle \text{ iff } \mathit{check}_g(c \cup c') \text{ implies } \left(\sigma \vDash^{c\cup c'} C_1 \text{ or } \sigma \vDash^{c} R\right)$$

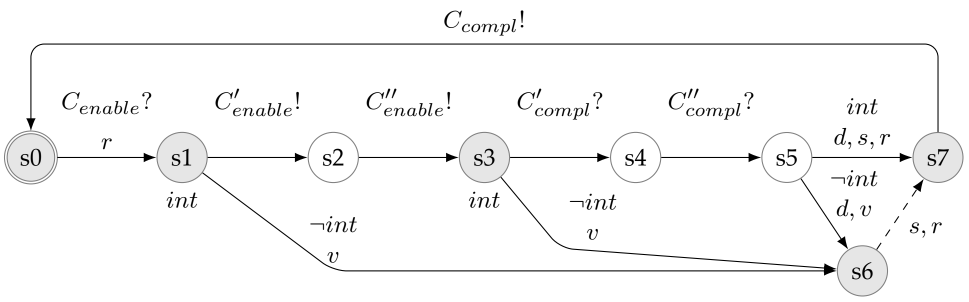
Argument. Refinement is covered in cases 4.6–4.8 below.

Case 4.6: Conjunction.

$$\sigma \vDash^{c} C' \mathit{ And } C'' \text{ iff } \sigma \vDash^{c} C' \text{ and } \sigma \vDash^{c} C''$$

Event traces. Traces must respect both C' and C'' individually while the conditions c' hold, or respect the reparation R.

Translation. All automata from the translations of C', C'' and R, one thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s6 \to s7$ is filled with the thread from the translation of R.



UPPAAL traces. The thread ensures that the main automaton is only enacted if and when the variable and time constraints in c' are met. The sub-automata for C' and C'' are both enacted (the order is not significant since the intermediate node is committed) ensuring that a trace of configurations must either satisfy both of these while the conditions c' hold, or the translation of R. The synchronisation with C_{compl} means that both thread and main automaton should reach a maximally extended state together.

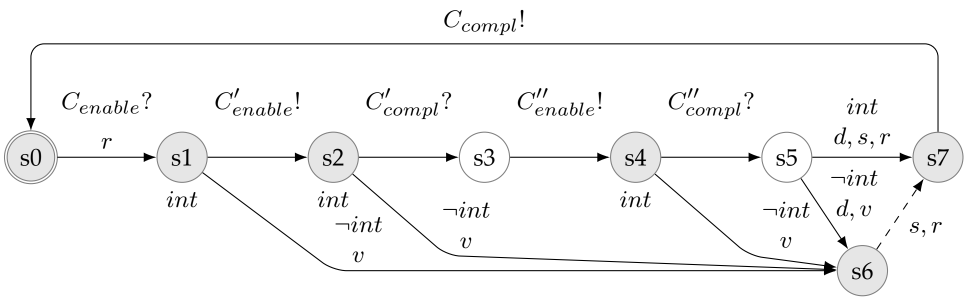
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in any order, while the conditions c' hold, or that the reparation is respected.

Case 4.7: Sequence.

$$\sigma \vDash^c C' \ Seq \ C'' \text{ iff } \exists i : 0 < i < length(\sigma) \text{ and } \sigma(..i) \vDash^c C' \text{ and } \sigma(i..) \vDash^c C''$$

Event traces. Traces can be divided in two, such that first subtrace respects C' and the second subtrace respects C'' while conditions c' hold, or the whole trace respects R.

Translation. All automata from the translations of C', C'' and R, one thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s6 \rightarrow s7$ is filled with the thread from the translation of R.



UPPAAL traces. The thread ensures that the main automaton is only enacted if and when the variable and time constraints in c' are met. The sub-automata for C' and C'' are enacted in sequence, such that C' must be complete before C'' is enacted. A trace of configurations must either satisfy both of these in order, while the conditions c' hold, or the translation of R. The synchronisation with C_{compl} means that both thread and main automaton should reach a maximally extended state together.

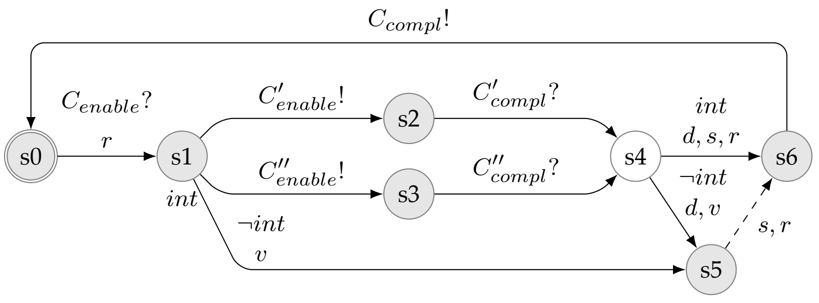
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in order, while the conditions c' hold, or that the reparation is respected.

Case 4.8: Choice.

$$\sigma \vDash^c C' \; \mathit{Or} \; C'' \text{ iff either } \sigma \vDash^c C' \text{ or } \sigma \vDash^c C''$$

Event traces. Traces must respect either C' or C'' while the conditions c' hold, or respect the reparation R.

Translation. All automata from the translations of C', C'' and R, one thread automaton (see Section 4.8.4) and one main automaton as follows, where edge $s5 \rightarrow s6$ is filled with the thread from the translation of R.



UPPAAL traces. The thread ensures that the main automaton is only enacted if and when the variable and time constraints in c' are met. Only one of the sub-automata for C' and C'' can be enacted, introducing non-determinism at node $s1$. A trace of configurations must either satisfy one of these while the conditions c' hold, or the translation of R. The synchronisation with C_{compl} means that both thread and main automaton should reach a maximally extended state together.

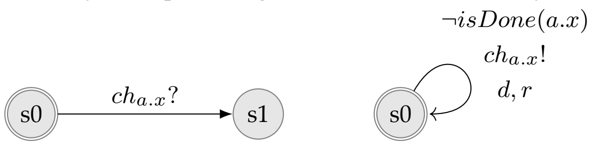
Argument. Both sets of traces require that either only one of the clauses in the refinement is respected, while the conditions c' hold, or that the reparation is respected.

Case 4.9: Simple action.

$$\sigma \vDash_a^c x \text{ iff } \exists i : 0 < i < \mathit{length}(\sigma) \text{ and } \langle a, x, t\rangle = \sigma(i) \text{ and } \mathit{check}_i(c, t)$$

Event traces. Traces must contain an event involving agent a and action x with a time stamp that complies with interval in c.

Translation. An action is simply an edge which synchronises on a broadcast channel $ch_{a.x}$ dedicated for that agent/action combination (below left). Each action also gets a corresponding *doer* automaton which broadcasts on this channel to represent the action being performed (below right). This can happen at any time, providing the action has not already been performed.



Time constraints do not appear at this level, however this simple automaton is always embedded within a larger one which would enforce such constraints (this is true of all the following *action* cases).

UPPAAL traces. Traces must contain a synchronisation on the $ch_{a.x}$ channel.

Argument. Both sets of traces require that the action is performed within a certain frame.

Case 4.10: Action conjunction.

$$\sigma \vDash_a^c C_3' \ \mathit{And}\ C_3'' \text{ iff } \sigma \vDash_a^c (C_3' \ \mathit{Seq}\ C_3'') \ \mathit{Or}\ (C_3'' \ \mathit{Seq}\ C_3')$$

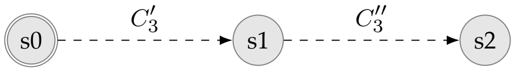
Argument. In both the trace semantics and the translation to UPPAAL, the *And* refinement is defined in terms of *Seq* and *Or*, and thus needs no special treatment here.

Case 4.11: Action sequence.

$$\sigma \vDash_a^c C_3' \ \mathit{Seq}\ C_3'' \text{ iff } \exists i : 0 < i < \mathit{length}(\sigma) \text{ and } \sigma(..i) \vDash_a^c C_3' \text{ and } \sigma(i..) \vDash_a^c C_3''$$

Event traces. Traces can be divided in two, such that first subtrace respects C_3' and the second subtrace respects C_3''.

Translation. The following automaton fragment, where each dashed edge is filled in with the translation of its label.



UPPAAL traces. A satisfying sequence of configurations must satisfy the translations C_3' and C_3'', strictly in that order.

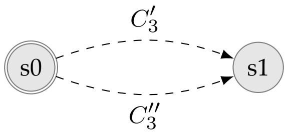
Argument. Both sets of traces ensure that both sub clauses are respected, in order.

Case 4.12: Action choice.

$$\sigma \vDash_a^c C_3' \mathit{Or}\ C_3'' \text{ iff either } \sigma \vDash_a^c C_3' \text{ or } \sigma \vDash_a^c C_3''$$

Event traces. Traces must respect either C_3' or C_3''.

Translation. The following automaton fragment, where each dashed edge is filled in with the translation of its label.



UPPAAL traces. A satisfying sequence of configurations must satisfy either the translation of C_3' or that of C_3'', introducing non-determinism at node $s0$.

Argument. Both sets of traces require that only one of the sub clauses is respected.

Case 4.13: Action naming.

$$\sigma \vDash_a^c \langle n, C_2 \rangle \text{ iff } \sigma \vDash_a^c C_2$$

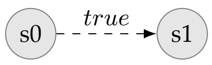
Argument. This case is simply handled recursively by considering the inner C_2 element.

Case 4.14: Top.

$$\sigma \vDash^c \top$$

Event traces. Any event trace respects *top*.

Translation. The following automaton fragment.



UPPAAL traces. This automaton is trivially satisfied by any trace.

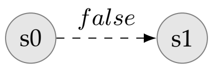
Argument. Both sets of traces are maximally inclusive.

Case 4.15: Bottom.

$$\sigma \not\models^c \ \bot$$

Event traces. No event trace respects *bottom*.

Translation. The following automaton fragment.



UPPAAL traces. This automaton is satisfied by no trace.

Argument. Both sets of traces are empty.

Case 4.16: Reference.

$$\sigma \models^c \#\mathit{name} \text{ iff } \sigma \models^c C' \text{ where } C' = \mathit{lookup}(\mathit{name})$$

Translation. A reference is translated by looking up the clause in the contract with name *name* and making a copy of its translation.

Chapter 5

ConPar: a tool for automatically building partial *C-O Diagrams*

John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider

Abstract. Our goal is to analyse *normative texts* by converting natural language into models in the *C-O Diagram* formalism. We present an experimental tool to help automate this modelling task. Using dependency structures obtained from the Stanford Parser and applying our own extraction rules, we produce a table which can then be converted into a *C-O Diagram*. We perform experiments of the tool on several documents from different domains, providing an initial evaluation of our approach.

Chapter contents

5.1 Introduction

Normative texts are natural language documents which are concerned with *deontic norms*: what **must**, **may be**, and **should not be** done. These may include legal contracts, terms of usage and service level agreements. We can analyse such texts by modelling them within a formalism that allows us to perform complex queries and verify properties about them. The formalism used for this task is *Contract-Oriented (C-O) Diagrams* [51, 26], which provides a language for visualising normative texts involving the modalities of **obligation**, **permission** and **prohibition** (indicated by the letters **O**, **P** and **F** respectively). These norms can be expressed over agents and actions, together with *reparations* which apply when obligations and prohibitions are violated.

Models in this formalism can be converted automatically into networks of timed automata (NTA) [1, 10], which are amenable to verification using the Uppaal tool [8]. Much work has gone into building an implementation of a framework for the modelling and analysis of *C-O Diagrams*. There is, however, a large semantic gap between texts in natural language and their formal representations. The task of modelling has thus far been completely manual, requiring a good knowledge of both domain and formalism.

Contributions. We aim here to address this front-end task by presenting a tool for processing normative texts and building partial models from them, by analysing their syntactic structure and extracting relevant information. Our method uses dependency structures obtained from the general-purpose the Stanford Parser [46, 77], which are then processed using custom rules and heuristics that we specified based on a small development corpus in order to produce a table of clauses (predicate candidates). This can be seen as a specific information extraction task. While this method may only produce a partial model requiring post-editing, our goal is to automate the most tedious part of the work so that the user (knowledge engineer) can focus better on formalisation details. We also discuss the application of this method to a small test corpus of unseen sentences, and report on the performance based on a simple precision-recall metric.

5.2 Extracting predicate candidates

The proposed approach is application-specific but domain-independent. We assume that normative texts tend to follow a certain restricted style of natural language, despite variations across and within domains. However, as we do not impose any grammatical or lexical restrictions on the input texts, we first apply a general-purpose statistical parser to obtain a syntactic dependency tree for each sentence. Provided the syntactic analysis does not contain significant errors,

Refin.	Mod.	Subject	Verb	Object	Adverbials
You must not, in the use of the Service, violate any laws in your jurisdiction (including copyright or trademark laws).					
-	F	User	violate	law	in User's jurisdiction
You will not upload viruses or other malicious code.					
	F	User	upload	virus	-
OR	F	User	upload	other malicious code	-
The examiner may, in consultation with the principal supervisor, assess whether the applicant has the capacity to successfully complete the doctoral programme.					
-	P	examiner	assess	applicant have capacity	in consultation with the principal ...
The RENTER shall pay all reasonable attorney and other fees, the expenses and costs incurred by OWNER in protection its rights under this rental agreement and for any action taken OWNER to collect any amounts due the OWNER under this rental agreement.					
-	O	renter	pay	reasonable attorney	under this rental agreement
AND	O	renter	pay	other fee	under this rental agreement
The equipment shall be delivered to RENTER and returned to OWNER at the RENTER's risk, cost and expense.					
-	O	equipment	[is] delivered [to]	renter	at the renter's risk, cost and expense
AND	O	equipment	[is] returned [to]	owner	at the renter's risk, cost and expense

Table 5.1: Sample input and partial output.

we then apply a number of interpretation rules and heuristics on the dependency structures, obtaining predicate candidates as shown in Table 5.1. More than one candidate is extracted in the case of explicit or implicit coordination of subjects, verbs, objects or main clauses.

In our experiment, we use the Stanford Parser whose accuracy on Penn Treebank (the WSJ section) is around 90% [77]. Using the Stanford dependency representation [24] allows for a more straightforward predicate extraction based on the syntactic relations, compared to using phrase structure. However, our approach is not restricted to a specific parser or dependency representation. The Stanford dependency representation [24] is being increasingly adapted to parsers for other languages as well, for instance, Chinese [19], Finnish [41] and Persian [76], and it is the basis for the Universal Dependencies project [25].

5.2.1 Expected input and intended output

The input text is pre-processed by splitting each sentence on a new line. In this experiment, we have manually selected only the relevant sentences, ignoring (sub)titles, introductory notes, etc. Automatic analysis of the document structure is a separate issue. We also expect that sentences do not contain grammatical errors that would considerably affect the syntactic analysis. The output of the tool is a table (in tab-separated format) where each line corresponds to a *C-O Diagram* box (clause) with the following fields:

- **Subject** — the agent of the clause;
- **Verb** — the verbal component of an action;
- **Object** — the object component of an action;
- **Modality** — obligation (O), permission (P), prohibition (F) or declaration (D) for clauses which only state facts;
- **Refinement** — whether a clause should be attached to the preceding clause by conjunction (AND), choice (OR) or sequence (SEQ);
- **Time** — adverbial phrases indicating temporality;
- **Adverbials** — other adverbial phrases that modify the action;
- **Conditions** — phrases indicating conditions on agents, actions or objects;
- **Notes** — other phrases that provide additional information (e.g. relative clauses), indicating the element (head word) they attach to.

Values of the *subject*, *verb* and *object* fields undergo certain normalisation: head words are lemmatised, Saxon genitives are converted to of-constructions if contextually possible, the preposition *"to"* is explicitly added to indirect objects, and definite and indefinite articles are omitted.

A complete document in this format can be converted automatically into a *C-O Diagram* model. Our tool however does not necessarily produce a complete table, in that fields may be left blank when we cannot determine what to use. Correctness of the output is also not guaranteed; certain clauses may be encoded in multiple ways, and, while all fields may be filled, the user may find it more desirable to change the encoding.

5.2.2 Rules

The **rules** in our system include everything that explicitly follows from the dependency relations and part-of-speech tags. For example, when present, the head of the subject noun phrase (NP) (labelled by `nsubj`), and the head of the direct object NP (`dobj`) are straightforwardly used to fill the *subject* and *object* fields (see Figure 5.1).

As another example, modal verbs and other auxiliaries of the main verb are labelled as `aux`

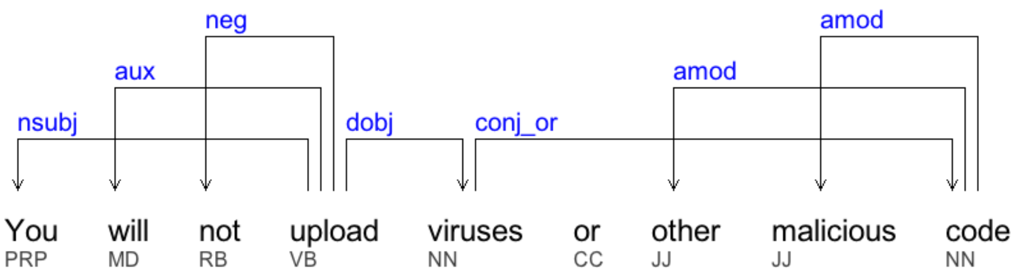


Figure 5.1: Sample dependency tree.

but words like *"may"* and *"must"* clearly indicate the respective modalities **P** and **O**. Auxiliaries can also be combined with other modifiers, such as the negation modifier *"not"* (`neg`), indicating **F**. In such cases, the rule is that obligation overrides permission, and prohibition overrides both obligation and permission.

In order to provide concise values for the *subject* and *object* fields, relative clauses (`rcmod`), verbal modifiers (`vmod`) and prepositional modifiers (`prep`) that modify heads of the subject and object NPs are separated in the *notes* field.

Adverbial modifiers (`advmod`), prepositional modifiers and adverbial clauses (`advcl`) that modify the main verb are separated, by default, in the *adverbials* field. If the main clause is expressed in the passive voice, and the agent is mentioned (expressed by the preposition *"by"*), the resulting predicate is converted to the active voice.

5.2.3 Heuristics

In addition to the obvious extraction rules, we apply a number of **heuristic** rules based on the development examples and our intuition about the application domains and of normative texts.

First of all, auxiliaries are compared and classified against extended lists of keywords; e.g. the modal verb *"can"* most likely indicates **P** while *"shall"* and *"will"* indicate **O**. We also consider the predicate itself (expressed by a verb, adjective or noun); e.g. words like *"responsible"*, *"liable"* and *"require"* most likely express **O**.

For prepositional phrases (PP) that are direct dependents of the verb, we first check if they reliably indicate a temporal modifier. The list of such prepositions include *"after"*, *"before"*, *"until"* etc. If not unambiguous, the head of the NP is checked to see if it bears a meaning of time. There is a relatively open list of such potential keywords, including *"day"*, *"week"*, *"month"* etc. Syntactic parsers often make PP-attachment errors, so if a PP attached to the object matches the above indicators, it is still put in the verb-dependent *Time* field.

The markers (`mark`) of adverbial clauses are also checked for indications of time (*"while"*, *"when"* etc.) and conditions (e.g. *"if"*). Adverbial modifiers are also checked against a list of irrelevant adverbs used for emphasis (e.g. *"very"*) or as gluing words (*"however"*, *"also"* etc.).

If there is no direct object in the sentence, or, in the case of the passive voice, no agent expressed by a prepositional phrase (using the preposition *"by"*), the first PP governed by the verb is treated as a prepositional object and thus is included in the *object* field. Finally, anaphoric references by personal pronouns are detected, normalised and tagged (e.g. *"we"*, *"our"* and *"us"* are all rewritten as *"<we>"*). In the case of terms of services, for instance, pronouns *"we"* and *"you"* are often used to refer to the service and the user respectively. The tool can be customised to carry out this simple but effective kind of anaphora resolution (see Table 5.1).

5.2.4 Post-editing

Our tool is not intended to completely replace a human knowledge engineer, and a certain amount of post-editing is often required. This can be sub-categorised into the following tasks, listed here in approximate order of effort required:

(i) filling in empty fields;
(ii) adding/removing adverbial information from subject and object;
(iii) changing verb/modality;
(iv) refinement into sub-clauses;
(v) complete paraphrasing.

5.3 Experiments

To test the potential and feasibility of the proposed approach, we have selected four normative texts from three different domains: a terms of service agreement, a rental agreement and research school regulations.

1. PhD regulations from our research school.[1]
2. Equipment rental agreement from RSO, Inc.[2]
3. Terms of service for GitHub, Inc.[3]
4. Terms of service for Facebook, Inc.[4]

[1] `http://document.chalmers.se/doc/fce9499c-feac-4152-809e-bfbbaf0fd8e9`, accessed 2015-06-15
[2] `http://www.rsoinc.com/pdfs/equip_rental_revb.pdf`, accessed 2015-06-15
[3] `https://github.com/site/terms`, accessed 2015-06-15
[4] `https://www.facebook.com/legal/terms`, accessed 2015-06-15

Document	**Rules only**			**Rules + heuristics**		
	Precision	*Recall*	F_1	*Precision*	*Recall*	F_1
PhD	0.66	0.73	0.69	0.82	0.90	0.86
Rental	0.75	0.67	0.71	0.71	0.66	0.69
GitHub	0.46	0.53	0.49	0.48	0.55	0.51
Facebook	0.43	0.54	0.48	0.43	0.57	0.49

Table 5.2: Evaluation results.

In the development stage we used 10 sentences from each document to help develop the tool heuristics. For the evaluation, we applied the tool to another 10 sentences of each document and manually evaluated the output for each.

5.3.1 Evaluation criteria

In our evaluation we use a simple precision/recall metric over the Subject, Verb, Object, and Modality fields. The other fields in the output were not included as they are too unstructured and always require some post-processing in order to be usable. The evaluation was performed twice: first on the output of the tool when using only the rules, and then again when using the rules and heuristics together. A summary of our experimental results can be found in Table 5.2, including the harmonic mean between precision and recall (F_1 scores).

Precision is concerned with rating the accuracy of the output. For each field in every row, one point is assigned when its value matches with our assessment of the correct value. When a single sentence results in multiple clauses, each of these is scored individually. **Recall** is a measure of how much information the tool was able to extract. For each sentence in the original text, we check whether the correct values have been extracted, scoring accordingly. When a sentence should result in multiple clauses, we score for each of these separately. The local scores for precision and recall are often identical, as one sentence in the original text generally corresponds to one clause. This does not hold when unnecessary refinements are added by the tool or, conversely, when co-ordinations in the text are not correctly added as refinements.

5.3.2 Observations

The large differences in the F_1 scores between documents (from 0.49 to 0.86) are mainly due to variations in language style. The heuristics do improve the scores, though the improvement is not equal across the different documents. Many sentence patterns handled in the heuristics do not in fact appear in the test sets. Of course the tiny corpus size is an issue here, and we cannot make any strong statements about the representative coverage of the test set.

Analysing the modal verb *shall* seems to be particularly difficult. It may either be an indication of an obligation when concerning an action, or a prescriptive construct as in *shall be* which is more indicative of a declaration.

5.3.3 Paraphrasing

The task of extracting the correct fields from each sentence can be seen as paraphrasing from the given sentence into one of the known patterns which can be handled by our rules. We give here some examples of errors encountered in the experiments, which can only currently be fixed by making non-trivial paraphrasings.

> *GitHub reserves the right at any time to modify or discontinue, temporarily or permanently, your access to the API (or any part thereof) with or without notice.*

For this sentence, our tool picks up *reserve* as verb and *right* as object, but this should really be realised as a permission with *modify* as the verb ad *access to API* as object. This could furthermore be refined as a permission of a choice of actions (modify, discontinue temporarily, discontinue permanently). Additional phrases such as *at any time* and *with or without notice* are actually not valuable here, as they reflect the default behaviour of the formalism (i.e. lack of constraints).

> *We require applications to respect your privacy, and your agreement with that application will control how the application can use, store, and transfer that content and information.*

Here we get an obligation with subject *we*, verb *require* and object *applications to respect your privacy*. The correct encoding however would be to make *applications* the subject, with *respect* as the verb and the object being *your privacy*.

> *When you publish content or information using the Public setting, it means that you are allowing everyone, including people off of Facebook, to access and use that information, and to associate it with you.*

In this case the tool fails completely, returning *it* as the subject, *mean* as verb and object *allowing everyone*, with the *declarative* modality (D). Here, the entire *when* clause should be treated as a condition. The phrase *you are allowing everyone* should more correctly be paraphrased as *everyone is allowed*, making this actually a permission with subject *everyone*, verb *access* and object *[that] information*.

To learn more about Platform, including how you can control what information other people may share with applications, read our Data Policy and Platform Page.

Other sentences such as this are generally unimportant for our purposes and should be skipped altogether, however we currently have no way identifying unhelpful sentences and ignoring them.

5.4 Formal analysis

The ultimate goal of this formalisation is automated analysis; by which we mean running queries of various kinds against a model. **Syntactic queries** are based on *predicates* defined over single clauses. The predicate $\mathit{isObl}(C)$ for example is true if clause C is an obligation. Such predicates are combined into larger queries over a model, such as checking if a model contains permissions for a particular agent or identifying obligations without constraints or reparations.

Queries dealing with timing constraints, possibility and invariance cannot be answered from syntactic model analysis alone. Such **semantic queries** are computed by converting our *C-O Diagram* model into a network of timed automata (NTA) and applying model checking using the UPPAAL requirement specification language, which is a subset of TCTL.

5.5 Related work

Information extraction is a large topic, where examples generally tend to consist of applying standard NLP techniques combined with customised rules to some specific domain and problem. Mercatali et al. [54] tackle the automatic translation of textual representations of laws to UML. This formalism is very different, primarily modelling the hierarchical structure of the documents rather than norms. Their method does not use dependency or phrase-structure trees but shallow syntactic chunks. Cheng et al. [20] also describe a system for extracting structured information for texts in a specific legal domain, combining surface-level methods like tagging and named entity recognition (NER) with semantic analysis rules which were hand-crafted for their domain.

Controlled natural language (CNLs) are often used to bridge the gap between natural and formal languages. This may be done using a general-purpose CNL such as Attempto Controlled English [35] which comes with a parser to discourse representation structures, or by writing a custom CNL specifically for *C-O Diagrams*, as in Camilleri et al. [16]. However a gap still exists between a natural language text and its CNL representation, which acts as a barrier to full automation. FrameNet-CNL by Barzdins [7] proposes an approach to the information extraction problem by combining CNL with FrameNet — a lexicographic database describing semantic frames and their syntactic realisation. This is a relevant approach to adapt and test on normative texts. This system encompasses a powerful abstract knowledge representation paradigm along with real-world information extraction system, based on frame-semantic parsing.

5.6 Summary

We have described our work-in-progress tool for processing normative texts in natural language and semi-automatically modelling them as *C-O Diagrams*. Our evaluation measures the accuracy of the tool in terms of precision and recall, but it would also be interesting to measure the time spent building a model from scratch versus post-editing the output from our tool.

It is common that some paraphrasing is needed during post-editing. This may purely syntactic (e.g. fixing adverbial attachment) but may also require using related or opposite concepts which simply cannot be determined without more processing on the semantic level.

Though the results of our experiments are indicative at best (because of the tiny corpus), we feel that applying the tool to the case studies reported here has undoubtedly eased the modelling task and warrants further work in this direction.

The *C-O Diagram* formalism is essentially *action-based*, where clauses prescribe what an agent should or shouldn't *do*. However in the texts from our experiments we have found that it is very common to describe what should or should not *be*, i.e. referring to states of affairs. Handling such sentences will require more effective paraphrasing patterns for these cases.

Glossary

API Application Programming Interface

AST Abstract Syntax Tree

$\mathcal{CL}$ Contract Logic [69]

CLAN $\mathcal{CL}$ Analyser [31]

CNL Controlled Natural Language

CSV Comma-Separated Values

CTD Contrary-to-Duty reparation

CTP Contrary-to-Prohibition reparation

C-O Contract-Oriented

DSL Domain-Specific Language

FL Formal Language

GF Grammatical Framework [72]

NTA Networks of Timed Automata [10]

NLP Natural Language Processing

PDL Propositional Dynamic Logic

RGL GF Resource Grammar Library [71]

SDL Standard Deontic Logic

TA Timed Automata [1]

TCTL Timed Computation Tree Logic

TSV Tab-Separated Values

UI User Interface

XML Extensible Markup Language

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*, pages 69–76. Association for Computational Linguistics, 2009.

[3] Krasimir Angelov and Aarne Ranta. Implementing controlled languages in GF. In *Workshop on Controlled Natural Language (CNL 2010)*, volume 5972 of *LNCS*, pages 82–101. Springer, 2010. ISBN 978-3-642-14417-2.

[4] Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. A framework for conflict analysis of normative texts written in controlled natural language. *Logic and Algebraic Programming*, 82(5-7):216–240, 2013. doi: 10.1016/j.jlap.2013.03.002.

[5] Tara Athan, Harold Boley, Guido Governatori, Monica Palmirani, Adrian Paschke, and Adam Wyner. OASIS LegalRuleML. In *International Conference on Artificial Intelligence and Law (ICAIL 2013)*, pages 3–12, 2013. doi: 10.1145/2514601.2514603.

[6] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. In *International Conference on Functional Programming (ICFP 2015)*, pages 315–327, New York, NY, USA, 2015. ACM. doi: 10.1145/2784731.2784747.

[7] Guntis Barzdins. FrameNet CNL: A knowledge representation and information extraction language. In *Controlled Natural Language*, volume 8625 of *LNCS*, pages 90–101. Springer, 2014. ISBN 978-3-319-10222-1. doi: 10.1007/978-3-319-10223-8_9.

[8] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL 4.0. Technical report, Department of Computer Science, Aalborg University, Denmark, 2006.

[9] Trevor Bench-Capon, Michał Araszkiewicz, Kevin Ashley, Katie Atkinson, Floris Bex, Filipe Borges, Daniele Bourcier, Paul Bourgine, Jack G. Conrad, Enrico Francesconi, Thomas F. Gordon, Guido Governatori, Jochen L. Leidner, David D. Lewis, Ronald P. Loui, L. Thorne Mccarty, Henry Prakken, Frank Schilder, Erich Schweighofer, Paul Thompson, Alex Tyrrell, Bart Verheij, Douglas N. Walton, and Adam Z. Wyner. A history of AI and law in 50 papers: 25 years of the international conference on AI and law. *Artificial Intelligence and Law*, 20(3): 215–319, September 2012. ISSN 0924-8463. doi: 10.1007/s10506-012-9131-x.

[10] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004. ISBN 978-3-540-22261-3. doi: 10.1007/b98282.

[11] Anders Björkelund and Pierre Nugues. Exploring lexicalized features for coreference resolution. In *CoCNLL 2011 Shared Task*, pages 45–50, Portland, Oregon, USA, 2011. Association for Computational Linguistics.

[12] Alexander Boer, Radboud Winkels, and Fabio Vitali. MetaLex XML and the Legal Knowledge Interchange Format. In Pompeu Casanovas, Giovanni Sartor, Núria Casellas, and Rossella Rubino, editors, *Computable Models of the Law*, pages 21–41. Springer, 2008. doi: 10.1007/978-3-540-85569-9_2.

[13] Lionel Briand. Capturing and analyzing legal requirements. Slide presentation, 2015. URL http://www.slideshare.net/briand_lionel/capturing-and-analyzing-legal-requirements. MoDRE keynote talk.

[14] D. Buscaldi, P. Rosso, J.M. Gómez-Soriano, and E. Sanchis. Answering questions with an n-gram based passage retrieval engine. *Intelligent Information Systems*, 34(2):113–134, 2009.

[15] John J. Camilleri, Gordon J. Pace, and Michael Rosner. Controlled Natural Language in a Game for Legal Assistance. In *Controlled Natural Language*, volume 7175 of *LNCS*, pages 137–153. Springer, June 2012.

[16] John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. A CNL for Contract-Oriented Diagrams. In *Workshop on Controlled Natural Language (CNL 2014)*, volume 8625 of *LNCS*, pages 135–146. Springer, 2014.

[17] John J. Camilleri, Filippo del Tedesco, and Gerardo Schneider. Modelling and analysis of normative texts. 2015. (Under submission).

[18] John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider. Extracting formal models from normative texts. 2015. (Under submission).

[19] Pi-Chuan Chang, Huihsin Tseng, Dan Jurafsky, and Christopher D. Manning. Discriminative reordering with Chinese grammatical relations features. In *Syntax and Structure in Statistical Translation (SSST 2009)*, pages 51–59, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. ISBN 978-1-932432-39-8.

[20] Tin Tin Cheng, J.L. Cua, M.D. Tan, K.G. Yao, and R.E. Roxas. Information extraction from legal documents. In *Symposium on Natural Language Processing (SNLP 2009)*, pages 157–162, 2009. doi: 10.1109/SNLP.2009.5340925.

[21] H. B. Curry. Some logical aspects of grammatical structure. In *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society, 1963.

[22] Alexandre David, M. Oliver Möller, and Wang Yi. Verification of UML statechart with real-time extensions. Technical report, Department of Information Technology, Uppsala University, Sweden, 2003.

[23] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *International Conference on Computer Aided Verificaiton (CAV 2011)*, pages 349–355, 2011.

[24] Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *COLING Workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8, 2008.

[25] Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D. Manning. Universal Stanford dependencies: A cross-linguistic typology. In *Language Resources and Evaluation Conference (LREC 2014)*, pages 4585–4592, 2014.

[26] Gregorio Díaz, Maria Emilia Cambronero, Enrique Martínez, and Gerardo Schneider. Specification and verification of normative texts using C-O Diagrams. *IEEE Transactions on Software Engineering*, 40(8):795–817, 2014. ISSN 0098-5589. doi: 10.1109/TSE.2013.54.

[27] Wei Dou, Domenico Bianculli, and Lionel Briand. OCLR: a more expressive, pattern-based temporal extension of OCL. Technical report, Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, 2014.

[28] ESTRELLA Project. The legal knowledge interchange format (LKIF). Technical report, 2008. URL http://www.estrellaproject.org/doc/Estrella-D4.1.pdf.

[29] F-Secure. Tainted love: How Wi-Fi betrays us, 2014. URL https://fsecureconsumer.files.wordpress.com/2014/09/wi-fi_report_2014_f-secure.pdf.

[30] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Automatic conflict detection on contracts. In *International Conference on Theoretical Aspects of Computing (ICTAC 2009)*, volume 5684 of *LNCS*, pages 200–214. Springer, 2009. ISBN 978-3-642-03465-7. doi: 10.1007/978-3-642-03466-4.

[31] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. CLAN: A tool for contract analysis and conflict discovery. In *Automated Technology for Verification and Analysis (ATVA 2009)*, volume 5799 of *LNCS*, pages 90–96. Springer, 2009. ISBN 978-3-642-04760-2.

[32] Mark D Flood and Oliver R Goodenough. Contract as automaton: The computational representation of financial agreements. *SSRN Electronic Journal*, 2015. ISSN 1556-5068. doi: 10.2139/ssrn.2538224.

[33] Norbert E Fuchs. First-order reasoning for Attempto controlled english. In *Workshop on Controlled Natural Language (CNL 2010)*, volume 7175, pages 73–94. Springer, 2012.

[34] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto controlled english (ACE) language manual, version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich, August 1999.

[35] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In *Reasoning Web*, volume 5224 of *LNCS*, pages 104–124. Springer, 2008. ISBN 978-3-540-85656-6.

[36] Thomas F. Gordon. An overview of the Carneades argumentation support system. In C. Reed, editor, *Dialectics, Dialogue and Argumentation - An Examination of Douglas Walton's Theories of Reasoning*, pages 145–156. King's College London, London, 2010. ISBN 978-1-84890-005-9.

[37] Ben Hachey and Claire Grover. Automatic legal text summarisation: Experiments with summary structuring. In *International Conference on Artificial Intelligence and Law (ICAIL 2005)*, pages 75–84, New York, NY, USA, 2005. ACM. ISBN 1-59593-081-7. doi: 10.1145/1165485.1165498.

[38] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.

[39] Yannis Haralambous, Julie Sauvage-Vincent, and John Puentes. A hybrid (visual/natural) controlled language. *Language Resources and Evaluation*, Special Issue: Controlled Natural Language, 2015. To appear.

[40] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *ACM*, 40:143–184, January 1993. ISSN 0004-5411.

[41] Katri Haverinen, Jenna Nyblom, Timo Viljanen, Veronika Laippala, Samuel Kohonen, Anna Missilä, Stina Ojala, Tapio Salakoski, and Filip Ginter. Building the essential resources for Finnish: the Turku dependency treebank. *Language Resources and Evaluation*, 48(3):493–531, 2014. ISSN 1574-020X. doi: 10.1007/s10579-013-9244-1.

[42] Hal Hodson. AI gets involved with the law. *New Scientist*, (2917), 2013. URL https://www.newscientist.com/article/mg21829175.900-ai-gets-involved-with-the-law/.

[43] Stefan Höfler. Legislative drafting guidelines: How different are they from controlled language rules for technical writing? In *Workshop on Controlled Natural Language (CNL 2012)*, number 7427 in LNCS, pages 138–151. Springer Verlag, 2012. CNL 2012.

[44] Stefan Höfler and Alexandra Bünzli. Designing a controlled natural language for the representation of legal norms. In *Workshop on Controlled Natural Language (CNL 2010)*, September 2010.

[45] Albert Sydney Hornby. *Oxford Advanced Learner's Dictionary of Current English, Third Edition*. Oxford University Press, 1974.

[46] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Annual Meeting of the Association for Computational Linguistics (ACL 2003)*, pages 423–430, 2003.

[47] Tobias Kuhn. An evaluation framework for controlled natural languages. In *Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *LNCS*, pages 1–20. Springer, 2010. ISBN 3-642-14417-9, 978-3-642-14417-2.

[48] Tobias Kuhn. *Controlled English for Knowledge Representation*. Doctoral thesis, Faculty of Economics, Business Administration and Information Technology, University of Zurich, 2010.

[49] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1), 2014.

[50] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. ISSN 1433-2779. doi: 10.1007/s100090050010.

[51] Enrique Martínez, Emilia Cambronero, Gregorio Díaz, and Gerardo Schneider. A model for visual specification of e-contracts. In *IEEE International Conference on Services Computing (IEEE SCC 2010)*, pages 1–8. IEEE Computer Society, 2010. ISBN 978-0-7695-4126-6. doi: 10.1109/SCC.2010.32.

[52] Jo McGinnis and Rg Pearce. The great disruption: How machine intelligence will transform the role of lawyers in the delivery of legal services. *Fordham Law Review*, 3041(14):3041–3066, 2014. ISSN 0015704X.

[53] Paul McNamara. Deontic logic. In D.M. Gabbay and J. Woods, editors, *Handbook of the History of Logic*, volume 7, pages 197–289. North-Holland Publishing, 2006.

[54] Pietro Mercatali, Francesco Romano, Luciano Boschi, and Emilio Spinicci. Automatic translation from textual representations of laws to formal models through UML. In *Conference on Legal Knowledge and Information Systems (JURIX 2005)*, pages 71–80. IOS Press, 2005.

[55] J.-J. Ch. Meyer, F.P.M. Dignum, and R.J. Wieringa. The paradoxes of deontic logic revisited: A computer science perspective. Technical report, Department of Computer Science, Utrecht University, Utrecht, 1994.

[56] Roger Mitton. A partial dictionary of English in computer-usable form. *Literary & Linguistic Computing*, 1:214–215, December 1986.

[57] Marie-Francine Moens, Erik Boiy, Raquel Mochales Palau, and Chris Reed. Automatic detection of arguments in legal texts. In *International Conference on Artificial Intelligence and Law (ICAIL 2007)*, pages 225–230, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-680-6. doi: 10.1145/1276318.1276362.

[58] Seyed M. Montazeri, Nivir Roy, and Gerardo Schneider. From contracts in structured English to CL specifications. In *Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2011)*, volume 68 of *EPTCS*, pages 55–69, 2011.

[59] Erika Doyle Navara, Silvia Pfeiffer, Robin Berjon, Steve Faulkner, Travis Leithead, and Edward O'Connor. HTML5. Candidate recommendation, W3C, 2014. URL http://www.w3.org/TR/2014/CR-html5-20140204/.

[60] Object Management Group (OMG). Semantics of business vocabulary and business rules (SBVR). Technical Report formal/2015-05-07, 2015. URL http://www.omg.org/spec/SBVR/1.3/PDF.

[61] Gordon J. Pace and Michael Rosner. A controlled language for the specification of contracts. In *Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *LNCS*, pages 226–245. Springer, 2010. ISBN 978-3-642-14417-2.

[62] Gordon J. Pace and Fernando Schapachnik. Contracts for interacting two-party systems. In *Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2012)*, 2012. doi: 10.4204/EPTCS.94.3.

[63] Gordon J. Pace and Gerardo Schneider. Challenges in the specification of full contracts. In *Integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 292–306, February 2009. ISBN 978-3-642-00254-0.

[64] Gordon J. Pace, Cristian Prisacariu, and Gerardo Schneider. Model checking contracts — a case study. In *Automated Technology for Verification and Analysis (ATVA 2007)*, volume 4762 of *LNCS*, pages 82–97. Springer-Verlag, 2007. ISBN 978-3-540-75595-1.

[65] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In Gibbons and de Moor, editor, *The Fun of Programming*, pages 105–129. Palgrave Macmillan, 2003. doi: 10.1.1.14.7885.

[66] Cristian Prisacariu. *A Dynamic Deontic Logic over Synchronous Actions*. PhD thesis, Department of Informatics, University of Oslo, 2010.

[67] Cristian Prisacariu. Logics for terms of services and their usefulness for automation. Slide presentation, 2013. URL https://frab.fscons.org/en/fscons13/public/events/27. FSCONS 2013.

[68] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2007)*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.

[69] Cristian Prisacariu and Gerardo Schneider. CL: An action-based logic for reasoning about contracts. In *Workshop on Logic, Language, Information and Computation (WOLLIC 2009)*, volume 5514 of *LNCS*, pages 335–349. Springer, 2009. ISBN 978-3-642-02260-9.

[70] Cristian Prisacariu and Gerardo Schneider. A dynamic deontic logic for complex contracts. *Logic and Algebraic Programming*, 81(4):458–490, 2012. ISSN 1567-8326.

[71] Aarne Ranta. The GF resource grammar library. *Linguistic Issues in Language Technology*, 2 (2), December 2009.

[72] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011.

[73] Paolo Rosso, Santiago Correa, and Davide Buscaldi. Passage retrieval in legal texts. *Logic and Algebraic Programming*, 80(3–5):139–153, 2011. ISSN 1567-8326.

[74] RuleML. Rule markup language initiative, 2015. URL http://wiki.ruleml.org/.

[75] Erich Schweighofer, Andreas Rauber, and Michael Dittenbach. Automatic text representation, classification and labeling in european law. In *International Conference on Artificial Intelligence and Law (ICAIL 2001)*, pages 78–87, New York, NY, USA, 2001. ACM. ISBN 1-58113-368-5. doi: 10.1145/383535.383544.

[76] Mojgan Seraji, Carina Jahani, Beáta Megyesi, and Joakim Nivre. A Persian treebank with Stanford typed dependencies. In *Language Resources and Evaluation Conference (LREC 2014)*, Reykjavik, Iceland, 2014. European Language Resources Association (ELRA). ISBN 978-2-9517408-8-4.

[77] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *Annual Meeting of the Association for Computational Linguistics (ACL 2013)*, pages 455–465, 2013.

[78] Harry Surden. Computable contracts. *UC Davis Law Review*, 46:629–700, 2012.

[79] Romuald Thion and Daniel Le Métayer. FLAVOR: A formal language for a posteriori verification of legal rules. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011)*, pages 1–8. IEEE Computer Society, 2011. ISBN 978-1-4244-9879-6.

[80] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Conference on Empirical Methods in Natural*

Language Processing (EMNLP 2000), volume 13, pages 63–70. Association for Computational Linguistics, 2000.

[81] Adam Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Höfler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, Martin Luts, Jonathan Pool, Mike Rosner, Rolf Schwitter, and John Sowa. On controlled natural languages: Properties and prospects. In *Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *LNCS/LNAI*, pages 281–289. Springer-Verlag, 2010. ISBN 3-642-14417-9, 978-3-642-14417-2.

[82] Adam Zachary Wyner. *Violations and Fulfillments in the Formal Representation of Contracts*. PhD thesis, Department of Computer Science, King's College London, 2008.