Thesis for the Degree of Doctor of Philosophy

Contracts and Computation

Formal modelling and analysis for normative natural language

John J. Camilleri

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2017

Contracts and Computation — Formal modelling and analysis for normative natural language
John J. Camilleri

ISBN 978-91-982237-4-3
Technical report 145D
Department of Computer Science and Engineering
Research groups: Formal Methods and 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

Cover artwork by the author
Typeset in TeX Gyre Pagella using XeLaTeX
Printed at Chalmers reproservice
Gothenburg, Sweden 2017

Abstract

Whether we are aware of it or not, our digital lives are governed by contracts of various kinds, such as privacy policies, software licenses, service agreements, and regulations. At their essence, normative documents like these dictate the permissions, obligations, and prohibitions of two or more parties entering into an agreement, including the penalties which must be paid when someone breaks the rules. Such documents are often lengthy and hard to understand, and most people tend to agree to these legally binding contracts without ever reading them.

Our goal is to create tools which can take a natural language document as input and allow an end user to easily ask questions about its implications, getting back meaningful answers in natural language within a reasonable amount of time. We do this by bringing formal methods to the analysis of normative texts, investigating how they can be effectively modelled and the kinds of automatic processing that these models enable.

This thesis includes six research papers by the author which cover the various aspects of this approach: entity recognition and modality extraction from natural language, controlled natural languages and visual diagrams as interfaces for modelling, logical formalisms which can be used for contract representation, and analysis via syntactic filtering, trace evaluation, random testing, and model checking. These components are then combined into a prototype tool for end users, allowing for end-to-end analysis of normative texts in natural language.

to Claudia
and Gabriel

Funding

The work presented in this thesis has been funded by the Swedish Research Council as part of the REMU project — *Reliable Multilingual Digital Communication: Methods and Applications* (grant number 2012-5746).

In addition, Paper V was partially supported by the Latvian State Research Programme *NexIT*.

Acknowledgements

A Ph.D. is a journey; one which has a clearly defined goal, but whose route is largely uncharted. Everyone you know and meet along the way will affect the path you end up taking, whether in minor or major ways. This thesis would have looked quite different, or perhaps not existed at all, were it not for a number of people.

First and foremost, I must thank Gerardo Schneider for being such a dedicated, supportive and wise supervisor. He has taught me not only a great deal about research and academia, but also a little bit about life too. Equally deserving of my gratitude is Aarne Ranta, who has shaped my career in crucial ways and who has been one of my greatest mentors. I am also very grateful to have had Koen Claessen as co-supervisor, not least for the motivation and advice he has given me during my time here.

In addition to these exceptional advisers, I have also had the opportunity to collaborate with some inspiring people, including Krasimir Angelov, Gabriele Paganelli, Filippo Del Tedesco, María-Emilia Cambronero, Normunds Grūzītis, and Bjørnar Luteberget. The quality of this work has no doubt benefited thanks to my follow-up group, licentiate discussion leader, thesis opponent, grading committee, and all the anonymous reviewers of our various publications.

More generally, I am thankful for all the people I've gotten to know through my work: my colleagues in the REMU project, the GF community, everyone I've met through CLT and CLASP, teachers I've had the pleasure of working with, office mates, lunch buddies, running friends, and all the administrative staff at our department.

Finally, I want to thank my sister Nicola for so willingly proofreading this thesis, my parents Joe and Anna for giving me opportunities that I am still realising the extent of, and most of all my wife Claudia and our son Gabriel, because without them none of this would mean a thing.

Structure of this thesis

This thesis consists of an introduction chapter followed by six individual research papers, listed below. These papers appear in their original form, with only minor edits made for formatting purposes. Thus, it is normal that there is some overlap in the background content of each paper.

Paper I John J. Camilleri and Gerardo Schneider. "Modelling and Analysis of Normative Documents". In: *Logical and Algebraic Methods in Programming* 91 (2017), pp. 33–59. DOI: 10.1016/j.jlamp.2017.05.002

Paper II Runa Gulliksson and John J. Camilleri. "A Domain-Specific Language for Normative Texts with Timing Constraints". In: *International Symposium on Temporal Representation and Reasoning (TIME 2016)*. IEEE, 2016, pp. 60–69. DOI: 10.1109/TIME.2016.14

Paper III Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. "A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language". In: *Logic and Algebraic Programming* 82.5-7 (2013), pp. 216–240. DOI: 10.1016/j.jlap.2013.03.002

Paper IV John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. "A CNL for Contract-Oriented Diagrams". In: *International Workshop on Controlled Natural Language (CNL 2014)*. Vol. 8625. Lecture Notes in Computer Science. Springer, 2014, pp. 135–146. DOI: 10.1007/978-3-319-10223-8_13

Paper V John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider. "Extracting Formal Models from Normative Texts". In: *International Conference on Applications of Natural Language to Information Systems (NLDB 2016)*. Vol. 9612. Lecture Notes in Computer Science. Springer, 2016, pp. 403–408. DOI: 10.1007/978-3-319-41754-7_40

Paper VI John J. Camilleri, Mohammad Reza Haghshenas, and Gerardo Schneider. *A Web-Based Tool for Analysing Normative Documents in English*. 2017. arXiv: 1707.03997 [cs.CL]

Contents

Introduction

1 Motivation

1.1 Normative texts

This thesis is about using formal methods to model and analyse real-world documents which determine what people can, should, and shouldn't do. We encounter documents like these in various forms, including privacy policies, terms of service documents, software licenses, service-level agreements, and regulations. We refer to these kinds of documents as **normative texts** or simply **contracts**. As they are written by and for humans, these documents are generally written in **natural language**. But how can informal texts be analysed using formal techniques? This problem plays a central role in this thesis. While the works presented here only consider contracts written in the English language, the methods we describe can be analogously applied to documents in other natural languages too, such as Swedish, Spanish, etc. As our approach is based on formal interlingua, a large part of the work is in fact language-agnostic.

The term *contract* has slightly different interpretations in different fields. Legally, a *contract* is commonly defined 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 a different concept: a promise of payment between parties, whose value changes over time, and which can be traded as an asset. In the context of blockchain technology, a *smart contract* is a piece of code stored on a public distributed ledger which governs the exchange of assets between parties. And in software engineering, *design by contract* refers to the specification of software in terms of assertions and pre- and post-conditions. While these concepts may be related in a general sense, we use the term *contract* in this thesis to refer to the class of normative texts described above.

1.2 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"*[1] — can have a number of negative outcomes, including the potential for exploitation of users and the erosion of respect for contracts in general.

In an experiment organised by the security company F-Secure [34], a free public Wi-Fi hotspot was set up in London with 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 the fact that these documents are often too long and hard for users to understand is gaining more attention.[2] A number of projects already exist which try to tackle this problem. Terms of service documents play a central role in this, and some projects already exist which try to make 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*. 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.

In the domain of software licenses, the **Choose a License**[5] website aims to help developers choose an appropriate Open-Source Software license for their project. They do this by summaris-

[1] http://biggestlie.com/

[2] See for example *Growing Up Digital: A report of the Growing Up Digital Taskforce* [23], where a UK-based law firm rewrote Instagram's terms of service to make them more easily understandable for children.

[3] https://tosdr.org/

[4] https://disconnect.me/icons/

[5] http://choosealicense.com/

ing 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. Similarly, the **tl;dr Legal**[6] project provides summaries of popular software licenses in simple English, together with summaries of what users *can*, *cannot*, and *must* do when agreeing to a given license.

The different copyright licenses available from the **Creative Commons**[7] also use icons to indicate their main features, but their approach goes a little deeper. All licenses incorporate a three-layer design, consisting of:

(i) a traditional legal tool, in the kind of language and text formats that lawyers work with;

(ii) a non-technical human-readable format called the *Commons Deed*, summarising and expressing some of the most important terms and conditions;

(iii) a machine-readable version written in the **Rights Expression Language (CC REL)**[8], containing a summary of the key freedoms and obligations in a format that software systems, search engines, and other kinds of technology can process.

So 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 time, interest in how computers can be used in the 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 [61] believe that computers will take on an increasingly larger role and eventually replace humans in certain tasks (e.g. legal search, generation of documents, and predictive analytics). They hold that as lawyers continue to embrace computational 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 [84] notes that the benefits of *computable contracting* could include reduced transaction costs associated with the contracting process, new potential for analysis and prediction, and the possibility of autonomous *computer-to-computer* contracting.

The **Legalese**[9] open source project and startup company highlights how the drafting of legal contracts today is still a largely manual task which is stuck at the level of piecing together templates in natural language, with no means for automatically checking the final contract for errors. Yet they note that contract drafting shares many parallels with software development, including the concepts of template reuse, exception handling, dependency graphs, version control, and collaborative editing. Thus, the project's vision is for tomorrow's lawyers to draft legal documents

[6] https://tldrlegal.com/

[7] https://creativecommons.org/licenses/

[8] https://wiki.creativecommons.org/wiki/CC_REL

[9] http://legalese.com/

the way today's programmers develop software: drawing on open source libraries, generating different machine-readable and human-friendly outputs from a single semantic representation, and using formal methods to verify their contracts before releasing them.

1.3 Classification and summarisation

Most of the projects mentioned in the previous section are concerned with summarising normative documents and classifying or rating them according to some criteria, in order to make them easier to understand. The simplest way of achieving this is to do it manually: decide on a set of criteria, read the texts, and then summarise and classifying them. Doing this for a single terms of service document would probably require a couple of person-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 the **ToS;DR** and **tl;dr Legal** projects are 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 the motivation for automating this task is great. A software tool that could process normative texts and automatically summarise and classify them would benefit everyone, from the writers of the policies to the users that must decide whether to accept them or not.

These are in fact common tasks in the field of natural language processing (NLP), and tend to make good cases for the application of machine learning techniques, given the availability of a suitable corpus for use as training data. There is an active research community in the area of artificial intelligence (AI) in the legal domain [12], and applying machine learning and other AI techniques to legal texts is by no means new [83, 66, 43]. Research such as this is also very relevant in the commercial sphere. One startup company out of Carnegie Mellon University called **LegalSifter**[10] specifically uses machine learning techniques to sell contract analysis services. Other similar companies include **Kira Systems**[11] and **Seal Software**[12], whose software aims to help customers better understand legal documents through summarisation, classification and advanced search tools.

[10] https://www.legalsifter.com/
[11] https://kirasystems.com/
[12] https://www.seal-software.com/

1.4 Deeper analysis

Classification and summarisation are useful tasks which can help users to determine quickly whether a particular contract meets some criteria or not. But there exist many other questions about contracts which we may want to answer automatically. Prisacariu [75] lists a number of interesting contract-processing tasks that would benefit from automation. These are expanded upon below, where we separate such tasks into the following categories:

- **Visualisation** — can the inherent structure of a natural language contract and the dependencies between its clauses be represented graphically?
- **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?
- **Querying** — which clauses in a contract pertain to a given event or party?
- **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 included in this thesis focus on the tasks of *conflict detection*, *visualisation*, *simulation*, *querying*, and *property testing*. In the following section we describe our approach to performing these types of analysis.

2 Approach

Natural language is rife with ambiguity. Most sentences in English contain ambiguity of some kind, be it lexical, syntactic, or semantic. As humans, we are generally very good at resolving ambiguity by using context and world knowledge, and most of it goes unnoticed in our everyday use of language. However, this ambiguity poses an enormous problem when trying to process natural language using computers based on **formal languages** which are well-defined, limited

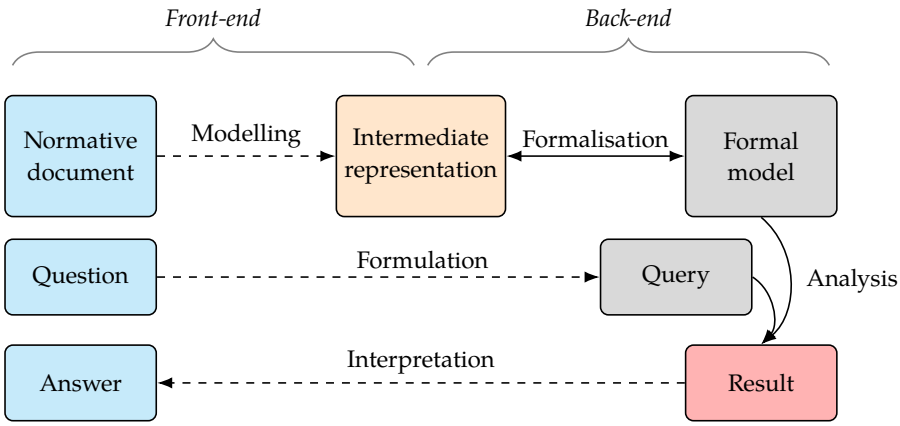


Figure 1: Overview of our approach to analysing normative texts. Dotted lines represent tasks which require manual effort by the user. Solid lines are fully automatic.

in expressivity, and unambiguous by design. As a result, it is not easy to apply the kinds of automatic analysis we are interested in directly on contracts written in natural language.

Instead, the approach we adopt is to first build a **model** of a contract in a formal language. By working with formal models instead of natural language text, the task of analysis becomes a concretely definable one which can be automated. A user would ideally like to have the benefit of both worlds: the familiarity of natural language combined with the power of formal methods, without having to be involved in the technical details of the latter. This is the ultimate goal which this thesis is interested in achieving. Our approach to contract analysis, which is outlined in Figure 1, can be divided into the following concerns (each of which is discussed in further detail in the following sections).

- **Modelling** — providing tools and interfaces for building and working with formal models of normative texts.
- **Formalisation** — designing a suitable formalism to use for modelling contracts.
- **Analysis** — processing contract models to detect conflicts, answer queries, or test if a property holds.

2.1 Modelling

Modelling is the process of building a formal object to represent and behave like some original informal object, in our case a normative text written in natural language. We see modelling as the *front-end* of our system. Apart from knowledge of the subject domain, modelling generally

also requires a level of expertise in the underlying formalism being used, and it is a task which the domain expert or end user may spend considerable time on.

It is important to remember that a model will always be a simplification or approximation of the real entity which it models. The structure of a model is limited by the syntactic constraints of the formalism, and its meaning depends on the semantics of that formalism, which may or may not match the original intended meaning. Thus, modelling is a process that can introduce errors or change meaning. The question of whether a given model is a faithful representation of its original text is not one that we directly address in this thesis. It is a problem which is inherent to the modelling approach; it can be mitigated but never solved completely. As observed by the statistician George Box, *"all models are wrong, but some are interesting."*[13]

Given that our goal is to build a contract analysis system for end users, we are interested in making the modelling process a user-friendly one. For this reason, we introduce an **intermediate representation** between the original text and the underlying formalism (as shown in Figure 1). The idea is that while models in this intermediate representation are directly translatable into models in the target formal language, they also have some properties which make them easier to work with for end users. To further explain this approach, we summarise below the main approaches to modelling considered in this thesis.

Controlled natural language

A *controlled natural language* (CNL) [93] 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 than that of its parent language, 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 [53].

A CNL which is designed to be close to a natural language which the user understands, but which is also a formal language that is automatically converted into the target formalism, can prove very useful as a high-level modelling language. Such a language is generally easier to read and understand for users who are not experts in the underlying formalism. Introducing a CNL is, however, not a replacement for modelling. Rather, it is a way of changing the level of abstraction at which the modelling process takes place, moving it away from the low-level logic of the formalism towards the higher-level concepts found in natural language.

Both of the CNLs discussed in this work have been implemented using the Grammatical

[13] https://en.wikipedia.org/wiki/All_models_are_wrong

Framework (GF) [79], 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 other representations, such as the concrete syntax of the formalism. As an example, consider the following sentence taken from the case study in Paper III:

> *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 paper, this clause could be modelled 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}
```

While this version of the clause is not entirely natural, it is straightforward to follow for anyone who understands English. However, whether or not it is an accurate representation of the original cannot be answered objectively; this ultimately depends on the semantics of the underlying formalism.

Graphical visualisation

As an alternative to text-based representations of contract models, we also consider the idea of having 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. [60]. 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.

Consider the example *C-O Diagram* in Figure 2. The main idea is that each box in the diagram represents a single norm, specifying a deontic modality over an agent and action. Boxes 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.

Tabular interface

Another approach to modelling considered in this work is the use of a tabular interface for model construction. An example of this can be seen in Figure 3. The idea is that the details and structure of a model can be completely specified by filling in the cells of a table. In the context of contracts,

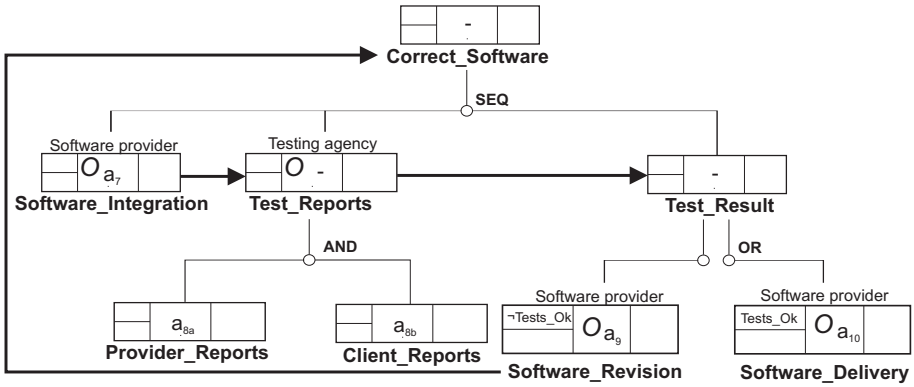


Figure 2: Full example of a *C-O Diagram*, modelling a software development process. [60]

Original sentence	No.	Modality	Subject	Verb	Object	Conditions
Students need to register for the course before the registration deadline, one week after the course has started.	1	O	student	register for	course	before 7
The deadline for the first assignment submission is on day 10.	2	O	student	submit	assignment	before 10
Graders should correct an assignment within one week of it being submitted.	3	OR[3.1,3.2]				isDone(#2) and within 7 of #2
–	3.1	O	grader	accept	assignment	
–	3.2	O	grader	reject	assignment	

Figure 3: Tabular interface for model construction, where each row corresponds to a clause in the final contract model. The contents of the cells are first filled in by an automatic text extraction process and can then be post-edited by the user, before finally exporting the table as a formal model.

each row corresponds to a clause while the columns separate to the various elements of each clause, such as action, agent and modality. Hierarchical structures such as conjunctions can be built by specifying refinement relationships between clauses via unique labels.

This interface was primarily conceived as the output format for an automatic clause extraction process from natural language text. As the output of this extraction inevitably requires some manual post-editing, the tabular format provides a familiar and intuitive interface for presenting the extracted information and allowing the user to make individual edits or insertions as needed. Once the user is done with the post-editing, the final contract model can be automatically exported from the tabular representation in a straightforward way.

2.2 Formalisms

What should a formal language for modelling contracts look like? On the one hand, a formalism should be expressive enough to facilitate the kinds of analysis desired, and as concise and modular as possible to simplify the task of processing it. On the other hand, it should also be defined at a high enough level of abstraction (that is, close enough to the source domain) so as to reduce the semantic gap between the original text and the model.

The formalisms used in this thesis for modelling contracts are based on deontic logic [62], containing at their core operators for **obligation**, **permission**, and **prohibition** of agents over actions. These are the primary modalities which we consider when modelling the kinds of normative documents we are interested in.[14]

Real-world contracts frequently contain deadlines, specifying enactment and expiry times for clauses as well as what should happen when deadlines are not met. Thus, in addition to the deontic modalities mentioned above, we are particularly interested in being able to model **temporal constraints**. This has been treated somewhat differently in the three formalisms considered in this thesis.

Contract Logic $\mathcal{CL}$

The formalism considered earliest in the works presented in this thesis is the contract language $\mathcal{CL}$ [78]. $\mathcal{CL}$ 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 [77, 76].

The formalism does not have a model of time or any operators for temporal constraints; actions can only be related to each other in terms of the order in which they occur, and specification of deadlines is not possible.

[14] There exist many other well-known concepts related to this area which differentiate themselves subtly from these modalities, such as the notion of power, right, duty, entitlement, etc. In choosing not to make these finer-grained distinctions in our work, we of course acknowledge that we can only approximate these concepts using our limited formalisms.

Contract-Oriented (C-O) Diagrams

Apart from the visual representation discussed in the previous section, *C-O Diagrams* are in fact based on a formal language with a well-defined syntax and semantics [28]. The majority of the works in this thesis make use of this formalism for modelling normative texts. As with $\mathcal{CL}$, the *C-O Diagram* language is based on the three main deontic modalities defined over complex actions. Individual clauses can be combined with refinement operators for conjunction, sequence, and choice.

The biggest novelty with *C-O Diagrams* is the ability to specify timing constraints for clauses. Time in this formalism is handled using *clocks*, a concept taken from the theory of Timed Automata (TA) [2]. Clocks can be thought of as real-valued variables which increment at a fixed rate, representing the passage of time in the wall-clock sense. A *C-O Diagram* has an implicit set of such clocks, corresponding to each clause in the model. 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).

Simplified Contract Language $\mathcal{SCL}$

$\mathcal{SCL}$ [42] is our own take on what a formalism for modelling normative texts should look like. It is *simplified* in the sense that each concept is separated out into its own constructor, such that guards and timing constraints are not defined as part of the main deontic operators (as in *C-O Diagrams*). Rather, the emphasis with $\mathcal{SCL}$ is on having a minimal set of combinators which are easily composable. This has a number of benefits when it comes to the back-end processing of the formalism. $\mathcal{SCL}$ is arguably at a lower level of abstraction than *C-O Diagrams*, and can be seen as a language which other higher-level formalisms are compiled into.

While it is based on the same deontic primitives as the other formalisms, $\mathcal{SCL}$ has a discrete model of time and comes with concrete operators for absolute and relative timing constraints. This avoids the need for clocks altogether and allows $\mathcal{SCL}$ to have a precise operational semantics defined in terms of time steps.

2.3 Analysis

We use the term *analysis* to refer to all types of automatic processing on contract models which can produce an answer to some question a user may ask about a contract. Answering different

kinds of questions requires different kinds of analysis techniques, but all generally consist of the following steps:

(i) formulating the question as a query or property in an associated query language;
(ii) running them against the model;
(iii) interpreting the results obtained in terms of the original query.

The methods for contract analysis covered in this thesis are summarised below.

Syntactic querying

Given a contract model, the simplest kind of analysis which can be performed on it involves traversing the model structurally and filtering out elements which match some criteria. As an example, a query to *"return all obligations of agent A"* could be represented as the predicate formula $\mathit{isObl} \wedge \mathit{agentOf}(\mathtt{A})$, which is then applied to all clauses found in the model. The list of matching clauses is then returned to the user, without the need for any complex processing. While relatively simple from a computational point of view, this kind of filtering can be very useful from an end user perspective. It is a functionality that comes *for free* once the trouble has been taken to build a model of a normative text.

Trace evaluation

Our contract models essentially describe groups of actions and the order in which they can and cannot appear. A natural question to ask of these contracts is thus, *"what happens if I do X followed by Y?"* Using the semantics of the formalism, this question can be answered via trace evaluation. Given an initial contract and a trace of events, we can return the residual contract which remains after the trace has been consumed (or an error, if the trace has violated the contract in a non-repairable way). This can be seen as a form of *simulation*.

Random testing

Trace evaluation can process a single concrete trace and return a corresponding residual contract. This can be extended and used to test large numbers of traces which are all different but which share some general property, such as containing a certain set of events in some order but without specifying the times at which they occur. This requires the right machinery for randomly generating traces according to some criteria, as well as a trace-evaluation function (as decribed in the previous section).

As this is ultimately a form of testing — which is weaker than verification — it cannot give guarantees about *all possible traces* through a system. However, this method is generally simpler

and faster than full verification, and can be quite useful for highlighting problems during the development phase of a contract without the need for writing large numbers of static test cases.

Conflict analysis

Contracts modelled in $\mathcal{CL}$ language can be checked for deontic conflicts using the CLAN analysis tool [32]. 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.

Model checking

The most advanced kind of contract analysis covered in this thesis is achieved through the technique of *model checking*, which is possible for contract models written in *C-O Diagrams* and $\mathcal{SCL}$ via translation into **networks of timed automata** (NTA) [13]. This kind of analysis is considerably more general than direct conflict detection. NTAs are amenable to model checking using the Uppaal [11] tool, which allows properties written in a subset of timed computation tree logic (TCTL) to be validated against the system. This allows us to test the following kinds of properties:

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

By using NTA to analyse our contract models, we are introducing yet another modelling language into the stack, which models in our previous formalism must now to be converted into. Apart from the potential problems with the correctness of this translation, another important issue is that NTAs are at a much lower level of abstraction. While our modelling formalisms are defined in terms of actions, clauses and refinement, NTAs are defined in terms of locations, edges and clock variables. This means that there is more work involved in converting a high-level query into a lower-level property in TCTL, as well as in interpreting any counter-examples returned by the model checker in terms of the original contract model.

3 Contributions

The body of this thesis is comprised of six individual research papers, which are included as separate chapters following this introduction. Brief descriptions of these papers are provided below, while Figure 4 gives a visual overview of the scope of each paper in terms of the topics covered. Note that the papers are grouped by subject, and are not necessarily presented in chronological order.

Paper I Modelling and analysis of normative documents with *C-O Diagrams*

John J. Camilleri and Gerardo Schneider. "Modelling and Analysis of Normative Documents". In: *Logical and Algebraic Methods in Programming* 91 (2017), pp. 33–59. DOI: 10.1016/j.jlamp.2017.05.002

Summary. This paper focuses on *C-O Diagrams* and the kinds of analysis possible on these models. Its novel contributions include extensions to the original formalism, the definition of a trace semantics, and an updated translation function from *C-O Diagrams* into NTA, complete with a fully-working implementation in Haskell. A small case study from a real-world contract is also included, demonstrating our methods for syntactic and semantic analysis of contract models.

Personal contribution. The changes to the formalism, the definition of the trace semantics, the entire implementation of the translation back-end, and the work on the case study.

Paper II $\mathcal{SCL}$: A domain-specific language for normative texts with timing constraints

Runa Gulliksson and John J. Camilleri. "A Domain-Specific Language for Normative Texts with Timing Constraints". In: *International Symposium on Temporal Representation and Reasoning (TIME 2016)*. IEEE, 2016, pp. 60–69. DOI: 10.1109/TIME.2016.14

Summary. This paper introduces the *Simplified Contract Language* $\mathcal{SCL}$, a domain-specific language for modelling contracts, inspired by *C-O Diagrams* but with a strong focus on compositionality. The language has a clearly defined operational semantics based on discrete time steps, and a highly modular translation to NTA, both of which are tested extensively using random test cases. The paper includes a case study showing how $\mathcal{SCL}$ can be used for the modelling, testing, simulation and verification of normative texts.

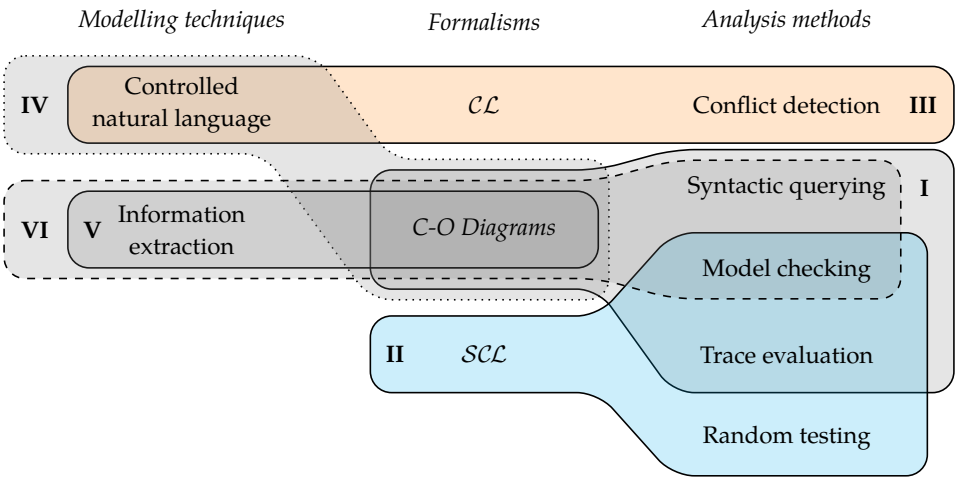


Figure 4: Overview of the topics covered in this thesis and the scope of each included paper.

Personal contribution. The design of the language and its semantics, implementation as an embedded DSL in Haskell, the overall method of the translation into NTA, and all work on random testing using QuickCheck.

Paper III AnaCon: A framework for conflict analysis of normative texts

Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. "A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language". In: *Logic and Algebraic Programming* 82.5-7 (2013), pp. 216–240. DOI: 10.1016/j.jlap.2013.03.002

Summary. This paper presents a CNL for the formal contract language $\mathcal{CL}$, implemented as a GF grammar, together with a basic tool for joining this CNL together with the conflict analysis tool CLAN. The novel contribution of this work is thus a simple framework for writing contracts in CNL and checking them automatically for conflicts, where any potential counter-examples are rendered back into CNL using the same grammar. The paper provides two case studies, demonstrating the iterative contract-modelling process using feedback from the conflict analyser.

Personal contribution. The implementation of the AnaCon tool, consultation on the design of the CNL, and all work related to the case studies.

Paper IV A CNL for *C-O Diagrams*

John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. "A CNL for Contract-Oriented Diagrams". In: *International Workshop on Controlled Natural Language (CNL 2014)*. Vol. 8625. Lecture Notes in Computer Science. Springer, 2014, pp. 135–146. DOI: 10.1007/978-3-319-10223-8_13

Summary. This paper presents another GF-based CNL for contracts, this time for the more expressive *C-O Diagram* formalism, which includes the ability to express timing constraints on clauses and comes with its own visual representation. Apart from the CNL itself, the contributions of this paper also include a web-based CNL editor with auto-completion and other helpful features, another web-based editor for visually manipulating *C-O Diagrams*, and a common interchange format between these representations.

Personal contribution. The complete design of the CNL and its implementation in GF, the building of the web-based CNL editing tool using standard web technologies, and all back-end conversion tools in Haskell.

Paper V Extracting formal models from normative texts

John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider. "Extracting Formal Models from Normative Texts". In: *International Conference on Applications of Natural Language to Information Systems (NLDB 2016)*. Vol. 9612. Lecture Notes in Computer Science. Springer, 2016, pp. 403–408. DOI: 10.1007/978-3-319-41754-7_40

Summary. This paper addresses the front-end task of contract modelling, using NLP methods to parse normative texts in English and build partial models of them in the *C-O Diagram* formalism. The idea is to provide a first-pass processing phase which could automatically extract some information from a contract and reduce some of the manual work required by modelling. The main contribution is a tool which, using the Stanford parser, extracts information from dependency trees using custom rules and heuristics, and outputs it in a custom tabular format for post-editing, before conversion into a final model. The paper includes a basic evaluation of this method, in terms of precision and recall over a small set of test sentences.

Personal contribution. Consultation on the system's heuristic rules, the design of the experiments, and carrying out the evaluation.

Paper VI *Contract Verifier*: A web-based tool for analysing normative documents in English

John J. Camilleri, Mohammad Reza Haghshenas, and Gerardo Schneider. *A Web-Based Tool for Analysing Normative Documents in English*. 2017. arXiv: 1707.03997 [cs.CL]

Summary. This paper ties together the tools developed in previous papers (I, IV & V) into a single web application, with the aim of bringing end-to-end analysis of normative texts in English to the end user. This is achieved by wrapping each individual tool as a web service with a corresponding API, and building a client-side web application which consumes these services. The novel contribution of this work is thus the web application itself, covering both the user interface and the supporting server architecture. The paper also includes a case study demonstrating the contract analysis workflow, starting from natural language and going all the way to verification of semantic queries.

Personal contribution. The design of the user workflow and overall system architecture, as well as the implementation of the back-end server in Haskell.

4 Related work

Each paper in this thesis includes its own *related work* section, listing other works that are more closely related to that specific paper. Here, we present some more generally related works on computational approaches to the legal domain.

Wyner [89] provides a thorough overview of the background, state-of-the-art, and future directions in the logical formalisation of legal texts in natural language. This includes the work by Wyner et al. [92], where the Attempto Controlled English (ACE) CNL is used for representing policy discussions, from which they can then be translated into first-order logic to support inference, consistency checking, and information extraction. As an alternative to using CNL for modelling, Wyner et al. [91] use the C&C/Boxer tool for parsing original legal texts and producing semantic representations of them using Discourse Representation Structures (DRS).

Logic-based formalisms for the legal domain come in various forms. In addition to the works mentioned below, we refer the reader to Hvitved [49, Chapter 1] for a wider overview of this area. One such language for formalising real-world contracts is the contract specification language **CSL**, introduced by Hvitved et al. [50]. The language supports a number of features, including conditional commitments, parametrised contract templates, relative and absolute temporal con-

straints, potentially infinite contracts, and in-place arithmetic expressions. CSL is designed as a concise modelling language, but also comes with an operational semantics mapping it into an abstract trace-based model for multiparty contracts with blame assignment. These semantics allow the derivation of a run-time monitor from a contract model, which in the case of noncompliance can specify exactly who is responsible for the breach of contract.

Azzopardi et al. [9] 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. The analysis in this work is limited to conflict detection, and the formalism has no support for temporal conditions.

Much like our own work, Gorín et al. [41] are also interested in applying temporal model checking to normative documents like regulations and contracts. They introduce the **FormaLex** toolset, which includes the LTL-based language together with tools that utilise model checking for finding normative incoherences (contradictions) in their models. Their input language is made up of a set of *rules*, which are LTL formulae with additional deontic operators aimed at capturing normative propositions, as well as a *background theory* which provides constructs for describing the class of models over which the rules predicate.

Abdelsadiq [1] also presents a toolkit for the verification of electronic contracts, using the Spin model checker for detecting conflicts in contract models. The Contractual Business-To-Business interaction model (**CB2B**) provides some high-level abstractions over the Promela language, allowing contract clauses to be encoded as event-condition-action (ECA) rules. The toolkit also includes a runtime monitor for contracts, which observes the interactions between parties and checks their contractual compliance with respect to the parties' sets of rights, obligations, and prohibitions.

Another take on a possible formalism for this domain is the **DynaLex** language, introduced by van Eijck and Ju [88]. This language uses dynamic logic for modelling legal relations, focusing on concepts like claims to rights, duties, privileges and liberties. Their conceptual model goes beyond the deontic modalities, allowing them to study the interplay of obligation, knowledge, and ignorance, and to model knowledge-based obligation.

Doesburg and Engers [29] make the case that the ideal formal representation for legal knowledge engineering should not be logic-based at all, but rather *frame-based*. This approach is based on classifying text fragments in natural language sources as elements of a semantic frame. By

using a semi-formal representation which is close to the original text, frames aim to preserve information of all potentially involved agents and their behavioural context. This task-agnostic representation can then be projected in different ways, depending on the processing task at hand.

Peyton Jones and Eber [73] describe a functional combinator language for modelling the complex financial contracts traded in derivative markets, implemented as a domain-specific language (DSL) embedded in Haskell.[15] These kinds of contracts are somewhat different from the normative documents we are concerned with in that they do not feature the deontic modalities; rather, they are treated as assets with a financial value which varies over time. For other work in this area refer to Szabo [85] and Bahr et al. [10].

Flood and Goodenough [33] also explore the representation of financial contracts, using finite-state automata where locations represent the states that a financial relationship can be in (such as *default* or *delinquency*), and 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). The authors 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.

There is also considerable work in the representation of contracts as knowledge bases or *ontologies*, rather than using logic-like languages. The **LegalRuleML** project [6] 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 legal texts in a machine-readable format in order to enable reasoning on them. It is part of the larger RuleML initiative [82], set up by 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 [40]. The **MetaLex** [14] language for legal and legislative resources has an even broader scope, aimed at allowing public administrations to link legal information between various levels of authority and different countries and languages.

The **Semantics of Business Vocabulary and Business Rules (SBVR)** [68] 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, 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 busi-

[15] https://web.archive.org/web/20130814194431/http://contracts.scheming.org

nesses. 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** [39] 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.

Paper I

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

John J. Camilleri and Gerardo Schneider

Abstract. We are interested in using formal methods to analyse *normative documents* or *contracts* 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 and including potential penalties resulting from the non-fulfilment of clauses. This is done using the *C-O Diagram* formalism, which we have extended syntactically and for which we have defined a new trace semantics. Models in this formalism can then be translated into networks of timed automata, and we have a complete working implementation of this translation. The network of automata is used as a specification of a normative document, making it amenable to verification against given properties. By applying this approach to a case study from a real-world contract, 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.

Contents

I.1 Introduction

We frequently encounter *normative documents* (or *contracts*) when subscribing to internet services and using software. These come in the form of terms of use, privacy policies, and service-level agreements, and we often accept these kinds of contractual agreements without really reading them. Though they are written using natural language, understanding the details of such documents often requires legal experts, and ambiguities in their interpretation are commonly disputed. Our goal is to model such texts formally in order to enable automatic querying and analysis of contracts, aimed at benefitting both authors of contracts and their users. To realise this, we are developing an end-to-end framework for the analysis of normative documents, combining natural language technology with formal methods. An outline of this framework is shown in Figure I.1.

Formal analysis requires a formal language: a given syntax together with a well-defined semantics and a state-space exploration technique. 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 texts. Instead, we choose to do this with a custom formalism based on the *deontic modalities* of **obligation**, **permission** and **prohibition**, and containing only the operators that are relevant to our domain. Specifically, we use the *Contract-Oriented (C-O) Diagram* formalism [28], 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, we are able to translate models in this formalism into networks of timed automata (NTA) [2] 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 can benefit greatly from the right tool support. In previous work [20] 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 (shown on the left-hand side of Figure I.1). 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 present work is concerned with the back-end of this system, focusing on the details of the modelling language and the different kinds of analysis that can be performed on these models.

Contributions and outline. The paper is laid out as follows. In Section I.2 we first present an extended definition of the *C-O Diagram* formalism, introducing an updated syntax and a novel trace semantics. Section I.3 then describes our own translation function from the extended *C-O*

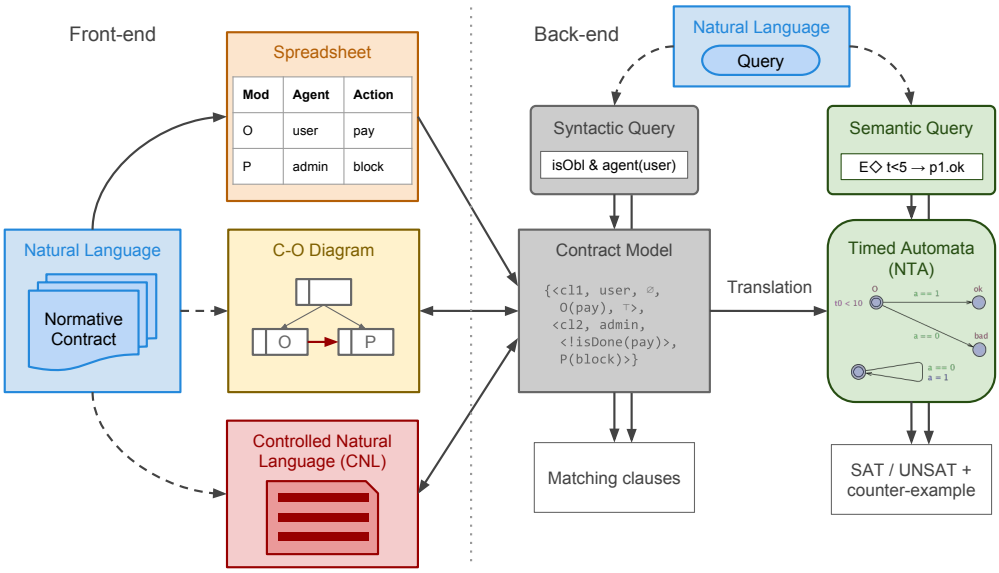


Figure I.1: Overview of our contract processing framework, separating the front-end concerns of model-building from the back-end tasks related to analysis. Dashed arrows represent manual interaction, while solid ones represent automatic steps.

Diagram formalism into Uppaal timed automata, which is more modular and fixes a number of issues with respect to the previous translation given in [28]. Our contribution includes the first fully-working implementation of this translation, written in Haskell. We also prove the correctness of this translation function with respect to our trace semantics. Section I.4 covers the analysis processes that we can perform on this formalism, discussing our methods for syntactic querying and semantic property checking of contract models. We demonstrate these methods by applying them to a case study from a real-world contract in Section I.5. Finally, we conclude with a comparison of some related work in Section I.6 and a final discussion in Section I.7.

Notation. Table I.1 below presents the symbols and function names used throughout the rest of this article.

Table I.1: Legend of symbols and functions used in this article. Where relevant, we have also included references to their definitions.

$\mathcal{N}$	set of names	ϕ, ψ	predicate name placeholders
$\mathcal{A}$	set of agents	ϵ	empty conditions
Σ	set of actions	$\emptyset$	empty guard/constraint list
$\mathcal{V}$	set of integer variables	ε	empty bound in interval
$\mathcal{C}$	set of clocks	Γ	environment
$\mathcal{B}$	set of Boolean flags	$get_{v/c/b}$	getters (I.17, I.18, I.19)
$\mathbb{N}$	type of natural numbers	$set_{v/b}$	setters (I.20, I.21)
$\mathbb{Z}$	type of integers	$reset_c$	reset clock (I.22)
$\mathbb{T}$	type of time stamps	*lookup*	lookup clause by name (I.23)
σ	event trace	τ	combine interval with constraints (I.24)
$\mathcal{T}$	set of event traces	*lst*	find lowest satisfying time stamp (I.28)
σ_U	UPPAAL timed trace	*check*	check a set of constraints (I.26)
$\mathcal{T}_U$	set of timed traces	*eval*	evaluate a constraint (I.27)
$\mathcal{T}_U^C$	set of timed traces corresponding to the satisfaction of C	*trf*	translate from *C-O Diagram* to UPPAAL model (Section I.3)
$\models$	*respects* relation between traces and contracts (Figure I.6)	*abstr*	translate from timed trace to event trace
		$\mathcal{Q}$	syntactic query function (I.53)

I.2 *C-O Diagram* formalism

C-O Diagrams were introduced by Martínez et al. [60] as a means for visualising normative texts involving obligation, permission and prohibition of agents over actions. The basic element in a *C-O Diagram* is the *box*, representing a simple clause (Figure I.2).

A box has four components:

(i) *guards* specify the conditions for enacting the clause;

(ii) an *interval* restricts the time during which the clause must be satisfied;

(iii) the box's propositional content specifies a *modality* applied over an *action*;

(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 is not fulfilled).

Each box also has an *agent* indicating the performer of the action, and a unique *name* for referencing purposes. Figure I.3 shows a completed example of such a box. Boxes can be expanded by using three kinds of refinement: *conjunction*, *choice*, and *sequence*, which allow complex clauses to be built out of simpler ones. Visually, complex clauses are represented as trees[1] where the child nodes signify the operands of the refinement, as shown in Figure I.4.

[1] Additional edges are sometimes included for visual clarity, making the diagrams technically graphs. However they do not change the model as such, and we still treat them structurally as trees.

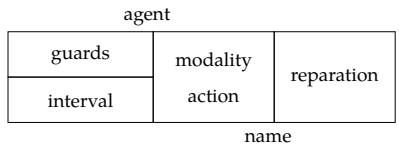
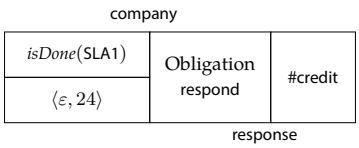


Figure I.2: The various components of a single *C-O Diagram* box.



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

Figure I.3: Example of a *C-O Diagram* box together with the natural language clause it models.

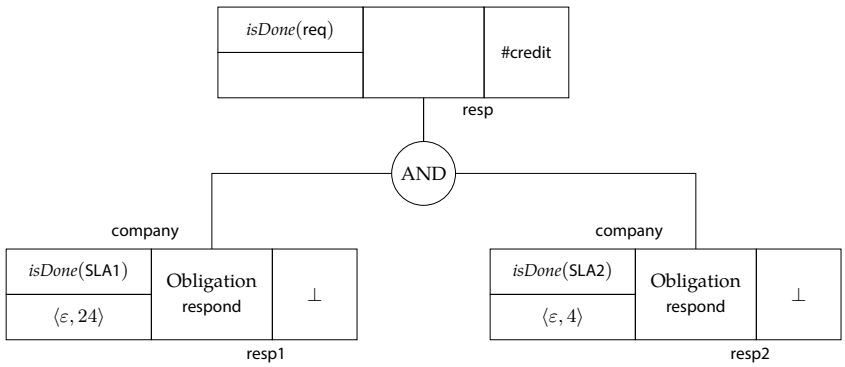
I.2.1 Formal syntax

Figure I.5 gives the formal grammar of our modelling language. It has been extended from its original definition given in [28]. These extensions are explained at the end of this section.

A **contract** specification K 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** (from the set $\mathcal{A}$) over an **action** (from the set Σ). Every clause is given a unique **name** (from the set of names $\mathcal{N}$) and optionally some **conditions** (described further below) which affect its applicability and expiration. A clause may have a **reparation** R, specifying another clause to be enacted if the main part of the clause is not satisfied. We use $\top$ for the trivially satisfied reparation, and $\bot$ for an unsatisfiable one. Instead of a modal statement, a clause can also be a refinement over sub-clauses using **conjunction** *And*, **sequence** *Seq* or **choice** *Or*. An action C_2 may be a single atomic action from the set Σ, or a complex one obtained by conjunction, choice or sequence. Finally, a clause may also be a simple named **reference** to another clause elsewhere in the contract.

Conditions are subdivided into guards and intervals. A **guard** is a conjunction of variable and timing constraints, which govern when a clause should be enacted. An **interval** is a tuple of an optional lower and upper bound on the time, which governs the window of time during which a clause is active and may be satisfied.[2] Given a finite set of integer variables $\mathcal{V}$, a **constraint over variables** is a Boolean formula comparing a variable against a constant ($v \sim z$) or against another variable ($v - w \sim z$). It can also be a predicate of the form $\phi(n)$. Similarly, given a finite set of

[2]The reason for the separation between guards and intervals is discussed on page 29.



When a request has been made, the company *must* respond *within 24 hours (in the case of* SLA1*) and within 4 hours (in the case of* SLA2*). In each case, if this target is not met, the customer is entitled to* credit.

Figure I.4: Example of refinement, where complex box resp is built from the conjunction of two simple boxes resp1 and resp2. The corresponding clause in natural language is also given.

clocks $\mathcal{C}$ — variables of abstract type $\mathbb{T}$ whose values increment at the same rate over time — a **timing constraint** is a Boolean formula comparing the absolute value of an individual clock ($c \sim t$) or the relative difference of two clocks ($c - d \sim t$).[3] By convention, we assume that for every name $n \in \mathcal{N}$ there is a clock in $\mathcal{C}$ with identifier t_n. The symbol ϵ is used as shorthand for the empty conditions $\langle \emptyset, \langle \varepsilon, \varepsilon \rangle \rangle$.

Well-formedness. Not all contracts which can be built from this grammar are considered valid. We define a *well-formed* 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, and
(v) clock names and predicates refer to existing boxes.

[3]Note that, for example, in the guard expression $\{x > 1, y < 5\}$ the elements of the set are syntactic objects denoting constraints; they are not part of a set definition.

$$K := \left\{ \langle C, \mathit{Type} \rangle^+ \right\} \text{ where } \mathit{Type} \in \{\mathit{Main}, \mathit{Aux}\} \tag{I.1}$$

$$C := \langle n, a, \mathit{Conditions}, O(C_2), R \rangle \tag{I.2}$$

$$\mid \langle n, a, \mathit{Conditions}, P(C_2) \rangle \tag{I.3}$$

$$\mid \langle n, a, \mathit{Conditions}, F(C_2), R \rangle \tag{I.4}$$

$$\mid \langle n, \mathit{Conditions}, C_1, R \rangle \tag{I.5}$$

$$\mid \mathit{Ref} \tag{I.6}$$

$$C_1 := C\ (\mathit{Seq}\ C)^+ \mid C\ (\mathit{And}\ C)^+ \mid C\ (\mathit{Or}\ C)^+ \tag{I.7}$$

$$C_2 := x \mid C_3\ (\mathit{Seq}\ C_3)^+ \mid C_3\ (\mathit{And}\ C_3)^+ \mid C_3\ (\mathit{Or}\ C_3)^+ \tag{I.8}$$

$$C_3 := \langle n, C_2 \rangle \tag{I.9}$$

$$R := \mathit{Ref} \mid \top \mid \bot \tag{I.10}$$

$$\mathit{Ref} := \#n \tag{I.11}$$

$$\mathit{Conditions} := \langle \mathit{Guard}, \mathit{Interval} \rangle \tag{I.12}$$

$$\mathit{Guard} := \{\mathit{Constraint}^*\} \tag{I.13}$$

$$\mathit{Constraint} := \phi(n) \mid v\ [-w] \sim z \mid c\ [-d\,] \sim t \tag{I.14}$$

$$\text{where } \phi \in \{\mathit{isDone}, \mathit{isComplete}, \mathit{isSat}, \mathit{isVio}, \mathit{isSkip}\}$$

$$v, w \in \mathcal{V},\quad c, d \in \mathcal{C},\quad \sim \in \{<, =, >\},\quad z : \mathbb{Z},\quad t : \mathbb{T}$$

$$\mathit{Interval} := \langle \varepsilon \mid t,\ \varepsilon \mid t \rangle \text{ where } t : \mathbb{T} \tag{I.15}$$

Figure I.5: Extended version of the *C-O Diagram* syntax [28] for contracts, where $n \in \mathcal{N}$, $a \in \mathcal{A}$ and $x \in \Sigma$. $\mathbb{T}$ represents the type of time stamps. Differences from the original include the top-level contract type K indicating *Main*/*Aux* clauses (I.1), the addition of cross-references (I.6), top/bottom as reparations (I.10), the distinction between guards and intervals (I.12), and the inclusion of predicates as constraints (I.14). In addition, our version of *C-O Diagrams* does not support repetition.

I.2.2 Extensions

The syntax presented here adds a number of extensions to the previous definition of *C-O Diagrams* given in [28]. These extensions were mainly introduced as a result of implementing the system as a runnable tool (see Section I.3.2) and to help the modeller by making common constructs easily expressible without requiring extra encoding. The extensions include:

1. *Re-structuring the top-level contract type as a forest of clause trees*
   Rather than modelling an entire contract as a single tree, it is more convenient to model groups of unrelated clauses in a list. This more closely matches the structure of natural language contracts. This structure also allows for more modularity and even re-use of

clauses, via the cross-referencing operator:

$$\Big\{\big\langle\langle \mathsf{m}, \epsilon, \#\mathsf{n}_1 \; \mathit{And} \; \#\mathsf{n}_2, \#\mathsf{n}_2\rangle, \mathit{Main}\big\rangle,$$
$$\langle\langle \mathsf{n}_1, \ldots\rangle, \mathit{Aux}\rangle,$$
$$\langle\langle \mathsf{n}_2, \ldots\rangle, \mathit{Aux}\rangle\Big\}$$

The example above shows how clause n_2 is defined once but referenced twice: in the main conjunction of m, as well as in its reparation. In the original language this kind of structure is only achievable by inlining, meaning that entire sub-clauses may need to appear multiple times in different parts of the same model.

2. *Distinguishing between* top ($\top$) *and* bottom ($\bot$) *reparations*
The previous version of the language only has a single type of null reparation (ϵ), whereas we want to be able to differentiate between one which is trivially satisfied and one which cannot be satisfied (making the parent clause irreparable). Consider the following two sequences of clauses:

$$\langle \mathsf{n}_1, \mathsf{agent}, \epsilon, O(\mathsf{action}_1), \top\rangle \;\; \mathit{Seq} \;\; \langle \mathsf{n}_2, \mathsf{agent}, \epsilon, O(\mathsf{action}_2), \bot\rangle$$
$$\langle \mathsf{n}_3, \mathsf{agent}, \epsilon, O(\mathsf{action}_1), \bot\rangle \;\; \mathit{Seq} \;\; \langle \mathsf{n}_4, \mathsf{agent}, \epsilon, O(\mathsf{action}_2), \bot\rangle$$

In the former sequence, the first clause (n_1) states that agent is obliged to perform action_1. However even if this first clause is violated, the second obligation n_2 will still be enabled as the reparation of n_1 is $\top$. This provides a kind of *soft violation*, which can be checked by constraints in other clauses. In the latter sequence, if n_3 is violated, then the entire sequence will be violated and never even reach n_4, as it impossible to repair $\bot$. These alternative kinds of reparation allow us to make such a distinction, which is not possible in the previous version of the language.

3. *Separating conditions into* guards *and* intervals
In the original language, it is impossible to specify whether a time constraint dictates the window during which the variable constraints should be checked, or the window within which the clause should be satisfied. This distinction is significant when the reparation of a clause has, in turn, its own timing constraints. Our revised structure for clause conditions allows timing constraints to be specified in two separate places: in guards (for enabling

clauses) and as intervals (for defining expiration). Consider the following two clauses:

$$\langle \mathsf{n}_1, \mathsf{agent}, \langle x = 1 \wedge t_0 < 5, \langle \varepsilon, \varepsilon \rangle \rangle, O(\mathsf{action}), \bot \rangle$$
$$\langle \mathsf{n}_2, \mathsf{agent}, \langle x = 1 \qquad\quad , \langle \varepsilon, 5 \rangle \rangle, O(\mathsf{action}), \bot \rangle$$

The former states that if $x = 1$ before clock t_0 reaches 5, then the obligation to perform action is enacted (otherwise it is skipped altogether). By contrast, the latter states that when $x = 1$ (at any time), then agent will be obliged to perform action within 5 time units from the enactment of the clause. This distinction is impossible to express in the previous syntax.

4. *Expressing guards as predicates rather than as variable comparisons*
   This was added to improve clarity during the modelling process, as querying the status of another clause is the most common kind of constraint. This can be seen as a purely syntactic change, allowing us to rewrite a constraint like $Sat_{\mathsf{clause}} = 1$ as $isSat(\mathsf{clause})$.

Furthermore, our version of *C-O Diagrams* does not include support for the repetition of clauses. This concept turned out to be quite problematic to define clearly, was deemed of lower utility, and thus removed altogether from our formalism.

For the remainder of this article we will use the term *C-O Diagrams* to refer to our extended version of the formalism (unless otherwise specified).

I.2.3 Trace semantics

Previous work defines the semantics of *C-O Diagrams* via translation to timed automata [28]. This translation is a complex operation with many individual cases to be handled, making it difficult to tell whether the semantics faithfully captures the intuition behind the constructs of the language. Thus, we define a completely new semantics for the formalism which is entirely independent from the translation function to timed automata. This allows us to compare the definition of the translation to the semantics, and argue that the former is correct with respect to the latter. It also allows a completely different back-end for *C-O Diagrams* to be verified for correctness without needing to compare it with the timed automata representation. The trace semantics may also be useful for deciding whether two contracts are equivalent, and thus proving the correctness of contract-rewriting rules.

We define a semantics in terms of *traces*. The intuition is that given a contract model, we want to know whether a sequence of actions respects or violates it. We choose to define a trace semantics so that the correctness of our translation function (Section I.3) may be proven by comparison with the semantics of UPPAAL automata, also defined in terms of traces [27].

Our trace semantics treats time as an abstract ordered type; a **time stamp** of type $\mathbb{T}$ is some value indicating a point in time, which can be directly compared with other time stamps. The examples used in this article represent time stamps as natural numbers.

We begin 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* $t : \mathbb{T}$*. The projection functions* $agent(e)$*,* $action(e)$ *and* $time(e)$ *extract the respective parts from an event.*

Traces can be referred to as follows: $\sigma(i)$ denotes the event at position i in trace σ, $\sigma(i..)$ denotes the finite sub-trace starting at event in position i until the end of the trace, and $\sigma(..j)$ is the sub-trace from the beginning of the trace to event $\sigma(j-1)$. Finally, $\sigma(i..j)$ is the sub-trace between indices i and j. The events in a trace are ordered by non-descending time stamp value (earliest events first) and indexed from 0 onwards. We say that a trace σ of length n is *well-formed* iff $\forall i, j \cdot (0 \leq i < n) \wedge (i < j < n) \implies time(\sigma(i)) \leq time(\sigma(j))$. We assume all our traces are well-formed.

The trace semantics of our language is defined via the *respects* relation ($\models$) between traces and contracts:

Definition 2. *We write* $\sigma \models C$ *to mean that* trace σ respects contract C *and* $\sigma \not\models C$ *for* trace σ does not respect (violates) contract C*. This relation is extended to clauses, where it is parametrised by a set of timing constraints and a starting time stamp (written* $\models_t^c$*). It is also extended to actions, where it is further parametrised by an agent (written* $\models_t^{c,a}$*). The set of all traces which respect a contract, indicated* $\mathcal{T}(C)$*, defines its* trace semantics.

We begin here by covering the concepts necessary for understanding our trace semantics. The rules defining the *respects* relation are then given in Figure I.6 on page 35.

Environment. The evaluation of constraints requires an environment $\Gamma : Env$ of Integer variables ($\mathcal{V}$), clocks ($\mathcal{C}$) and Boolean flags ($\mathcal{B}$), whose values may change over time. An environment can thus be seen as a function from a time stamp to a set of valuations (I.16). Clocks can be seen as variables of type $\mathbb{T}$ whose values automatically increase with the progression of time. All variables and clocks are initialised to 0, and all flags to *False*. We use $\Gamma_{\mathcal{V}}$, $\Gamma_{\mathcal{C}}$ and $\Gamma_{\mathcal{B}}$ to project the

respective parts of the environment.

$$Env = \mathbb{T} \to \langle \mathcal{V} \mapsto \mathbb{Z},\ \mathcal{C} \mapsto \mathbb{T},\ \mathcal{B} \mapsto Boolean \rangle \tag{I.16}$$

$$get_v\ :\ Env \to \mathbb{T} \to \mathcal{V} \to \mathbb{Z} \tag{I.17}$$

$$get_c\ :\ Env \to \mathbb{T} \to \mathcal{C} \to \mathbb{T} \tag{I.18}$$

$$get_b\ :\ Env \to \mathbb{T} \to \mathcal{B} \to Boolean \tag{I.19}$$

$$set_v\ :\ Env \to \mathbb{T} \to \mathcal{V} \to \mathbb{Z} \to Env \tag{I.20}$$

$$set_b\ :\ Env \to \mathbb{T} \to \mathcal{B} \to \mathbb{Z} \to Env \tag{I.21}$$

$$reset_c\ :\ Env \to \mathbb{T} \to \mathcal{C} \to Env \tag{I.22}$$

The environment can be queried via the *get* functions (I.17–I.19). Integer and Boolean variables can be updated using the *set* functions (I.20 and I.21), while clocks can be reset to 0 with $reset_c$ (I.22). The clock t_0 is used to indicate the current time, i.e. it is a clock which is never reset. An update affects all valuations from the given time stamp onwards. The set of Boolean flag variables is used to represent the status of boxes and actions in the contract model, for example whether an action has been completed or a clause has been violated. Guards expressed as predicates are encoded as comparisons involving these variables.

As a *respects* relation is applied and a contract evolves, the environment needs to be updated so that the state of each clause is kept up-to-date and clocks are reset as needed. For clarity however, these updates are not explicitly marked in the rules in Figure I.6. The environment itself does not appear in the rules either, as it is implicitly globally accessible. Updates to the environment are made in the following cases:

(i) when a clause n is enabled, clock t_n is reset;
(ii) when a clause n is satisfied (including via reparation), Sat_n is set to *True* and clock t_n is reset;
(iii) when a clause n is violated, Vio_n is set to *True*;
(iv) when the guard for clause n expires, $Skip_n$ is set to *True*;
(v) when an action x is performed by agent a, $Done_{a.x}$ is set to *True* and clock $t_{a.x}$ is reset, while $Done_n$ is also set to *True* for the parent clause n.

The *lookup* function (I.23) is used for resolving named cross-references between clauses. This function searches recursively over the structure of the contract model, returning the matching

clause or $\perp$ if none is found.

$$\mathit{lookup} : K \to \mathcal{N} \to C \tag{I.23}$$

Constraint satisfaction. Intervals are passed down from parent clauses by adding them to the set of timing constraints. We use the function τ (I.24) for combining an interval with an existing set of constraints (where empty bounds ε are ignored). The related function τ' (I.25) is used for combining the expired upper bound of a given interval with a set of constraints.

$$\begin{aligned} &\tau : \{\mathit{Constraint}^*\} \to \mathcal{N} \to \mathit{Interval} \to \{\mathit{Constraint}^*\} \\ &\tau(c, n, \langle i_{low}, i_{upp}\rangle) = c \cup \{t_n > i_{low}, t_n < i_{upp}\} \end{aligned} \tag{I.24}$$

$$\begin{aligned} &\tau' : \{\mathit{Constraint}^*\} \to \mathcal{N} \to \mathit{Interval} \to \{\mathit{Constraint}^*\} \\ &\tau'(c, n, \langle \_, i_{upp}\rangle) = c \cup \{t_n \geq i_{upp}\} \end{aligned} \tag{I.25}$$

Checking constraints from both guards and intervals is done with the *check* function (I.26). This function looks up the state of the environment Γ at time t and returns the conjunction of the results of evaluating each of its Boolean expressions with the *eval* function (I.27).

$$\mathit{check} : \{\mathit{Constraint}^*\} \to \mathbb{T} \to \mathit{Boolean} \tag{I.26}$$

$$\mathit{check}\big(\{c_1, \ldots, c_n\}, t\big) = \begin{cases} \mathit{True} & \text{if } n = 0 \\ \bigwedge_{1 \leq j \leq n} \mathit{eval}(c_j, t) & \text{otherwise} \end{cases}$$

$$\mathit{eval} : \mathit{Constraint} \to \mathbb{T} \to \mathit{Boolean} \tag{I.27}$$

$$\mathit{eval}\big(x \sim n, t\big) = \mathit{get}(\Gamma, t, x) \sim n$$

$$\mathit{eval}\big(x - y \sim n, t\big) = \mathit{get}(\Gamma, t, x) - \mathit{get}(\Gamma, t, y) \sim n$$

$$\mathit{eval}\big(\mathit{is}\phi(n), t\big) = \begin{cases} \mathit{eval}(\mathit{isSat}(n), t) \vee \mathit{eval}(\mathit{isSkip}(n), t) \\ \qquad \text{if } \phi = \mathit{Complete} \\ \mathit{get}_b(\Gamma, t, \phi_n) \\ \qquad \text{if } \phi \in \{\mathit{Done}, \mathit{Sat}, \mathit{Vio}, \mathit{Skip}\} \end{cases}$$

In order to determine the moment at which a clause will become enabled, we define the notion of *lowest satisfying time stamp*. Given a guard, we want to know the earliest time stamp later than t for which that guard becomes *True* in the environment. This is captured in the *lst* function (I.28), which is a partial function as the guard may in fact never be satisfied.

$$lst : Guard \to \mathbb{T} \to \mathbb{T} \tag{I.28}$$

$$lst(g,t) = \begin{cases} t' & \text{if } \exists t' : \mathbb{T} \cdot t' = \min_{\forall u \geq t} \left[check(g,u) = True \right] \\ undefined & \text{otherwise} \end{cases}$$

Rules. The rules defining the *respects* relation are given in Figure I.6. These rules work by recursing over the structure of the contract specification rather than iterating through the trace. In other words, an action is not consumed from the trace when it satisfies a particular clause. Each rule searches for the earliest event that satisfies it. Rules for sequential refinement (I.34 and I.38) are the only ones that divide a trace into sub-traces, as they enforce order. For a more detailed explanation of how each rule works, please refer to Appendix I.A.5.

I.2.4 Example

Consider the contract model shown earlier in Figure I.4. This can be represented in our formal syntax as follows:

$$C = \left\{ \left\langle \langle \mathsf{resp}, \langle isDone(\mathsf{req}), \langle \varepsilon, \varepsilon \rangle \rangle, C' \; And \; C'', \#\mathsf{credit} \rangle, Main \right\rangle \right\}$$

where

$$C' = \langle \mathsf{resp1}, \mathsf{company}, \langle isDone(\mathsf{SLA1}), \langle \varepsilon, 24 \rangle \rangle, O(\mathsf{respond}), \bot \rangle$$

$$C'' = \langle \mathsf{resp2}, \mathsf{company}, \langle isDone(\mathsf{SLA2}), \langle \varepsilon, 4 \rangle \rangle, O(\mathsf{respond}), \bot \rangle$$

We wish to use the rules defined in Section I.2.3 to determine whether a given trace of events satisfies the contract or not. Take the following trace as an example:

$$\sigma = \left[\langle \mathsf{company}, \mathsf{respond}, 5 \rangle \right]$$

This contains a single event, which is the agent company performing the respond action at time stamp 5. To determine whether this trace respects the given contract, we need to find a derivation for $\sigma \vDash C$. To do this, we also need an environment containing information about the status of

Contract

$$\sigma \vDash \quad \{\langle C^1, T^1\rangle, \ldots, \langle C^n, T^n\rangle\} \text{ iff } \bigwedge_{\substack{1\leq i\leq n,\\ T^i = \mathit{Main}}} \sigma \vDash^{\epsilon}_{0} C^i \tag{I.29}$$

Deontic operators

$$\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i\rangle, O(C_2), R\rangle \tag{I.30}$$

$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \vDash^{\tau(c,n,i),a}_{t} C_2 \text{ or } \sigma \vDash^{\tau'(c,n,i)}_{t} R\big)$$

$$\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i\rangle, P(C_2)\,\rangle \tag{I.31}$$

$$\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i\rangle, F(C_2), R\rangle \tag{I.32}$$

$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \vDash^{\tau(c,n,i),a}_{t} C_2 \text{ implies } \sigma \vDash^{c}_{t} R\big)$$

Refinement

$$\sigma \vDash^{c}_{t_0} \langle n, \langle g, i\rangle, C_1, R\rangle \tag{I.33}$$

$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \vDash^{\tau(c,n,i)}_{t} C_1 \text{ or } \sigma \vDash^{\tau'(c,n,i)}_{t} R\big)$$

$$\sigma \vDash^{c}_{t_0} C' \mathit{\ Seq\ } C'' \tag{I.34}$$

$$\text{iff } \exists j : \mathbb{N} \cdot \big(0 \leq j \leq \mathit{length}(\sigma) \wedge \sigma(..j) \vDash^{c}_{t_0} C' \wedge \sigma(j..) \vDash^{c}_{t_0} C''\big)$$

$$\sigma \vDash^{c}_{t_0} C' \mathit{\ And\ } C'' \text{ iff } \sigma \vDash^{c}_{t_0} C' \text{ and } \sigma \vDash^{c}_{t_0} C'' \tag{I.35}$$

$$\sigma \vDash^{c}_{t_0} C' \mathit{\ Or\ } C'' \text{ iff either } \sigma \vDash^{c}_{t_0} C' \text{ or } \sigma \vDash^{c}_{t_0} C'' \tag{I.36}$$

Actions

$$\sigma \vDash^{c,a}_{t_0} x \text{ iff } \exists j : \mathbb{N} \cdot \big(0 \leq j < \mathit{length}(\sigma) \wedge \langle a, x, t\rangle = \sigma(j) \wedge t_0 \leq t \wedge \mathit{check}(c, t)\big) \tag{I.37}$$

$$\sigma \vDash^{c,a}_{t_0} C_3' \mathit{\ Seq\ } C_3'' \tag{I.38}$$

$$\text{iff } \exists j : \mathbb{N} \cdot \big(0 < j < \mathit{length}(\sigma) \wedge \sigma(..j) \vDash^{c,a}_{t_0} C_3' \wedge \sigma(j..) \vDash^{c,a}_{t_0} C_3''\big)$$

$$\sigma \vDash^{c,a}_{t_0} C_3' \mathit{\ And\ } C_3'' \text{ iff } \sigma \vDash^{c,a}_{t_0} (C_3' \mathit{\ Seq\ } C_3'') \mathit{\ Or\ } (C_3'' \mathit{\ Seq\ } C_3') \tag{I.39}$$

$$\sigma \vDash^{c,a}_{t_0} C_3' \mathit{\ Or\ } C_3'' \text{ iff either } \sigma \vDash^{c,a}_{t_0} C_3' \text{ or } \sigma \vDash^{c,a}_{t_0} C_3'' \tag{I.40}$$

$$\sigma \vDash^{c,a}_{t_0} \langle n, C_2\rangle \text{ iff } \sigma \vDash^{c,a}_{t_0} C_2 \tag{I.41}$$

Reparation

$$\sigma \vDash^{c}_{t_0} \top \tag{I.42}$$

$$\sigma \nvDash^{c}_{t_0} \bot \tag{I.43}$$

$$\sigma \vDash^{c}_{t_0} \#n \text{ iff } \sigma \vDash^{c}_{t_0} \mathit{lookup}(n) \tag{I.44}$$

Figure I.6: Definition of the *respects* relation ($\vDash$) between traces and contracts (type K in Figure I.5), clauses (types C and C_1) and actions (types C_2 and C_3).

the external clauses referenced in our example (SLA1, SLA2 and req):

$$\Gamma_{\mathcal{B}}(0..2) = \{Done_{\mathsf{SLA1}} \mapsto \mathit{False},\ Done_{\mathsf{SLA2}} \mapsto \mathit{True},\ Done_{\mathsf{req}} \mapsto \mathit{False}\}$$
$$\Gamma_{\mathcal{B}}(3) = \{Done_{\mathsf{SLA1}} \mapsto \mathit{False},\ Done_{\mathsf{SLA2}} \mapsto \mathit{True},\ Done_{\mathsf{req}} \mapsto \mathit{True}\}$$

Note that at time stamp 3, the value of $Done_{\mathsf{req}}$ changes to *True*. The environment will also contain clocks and further flags pertaining to the clauses in our example (resp, resp1 and resp2). Only those parts of the environment which are relevant to the example are discussed here.

To begin our derivation, we first apply rule (I.29) which says that we must satisfy each of the *Main* clauses in the contract with the empty constraints and from time stamp 0. This gives us:

$$\sigma \vDash_0^{\epsilon} \langle \mathsf{resp}, \langle \mathit{isDone}(\mathsf{req}), \langle \varepsilon, \varepsilon \rangle \rangle, C' \mathit{And}\ C'', \#\mathsf{credit} \rangle \tag{I.45}$$

By rule (I.33), we then try to find the lowest satisfying time stamp (*lst*) which satisfies the given guard, i.e. $\mathit{lst}(\mathit{isDone}(\mathsf{req}), 0)$ which from the environment is 3 — the point at which $Done_{\mathsf{req}}$ becomes *True*. Thus we have either that the main clause is satisfied:

$$\sigma \vDash_3^{\tau(\epsilon, req, \langle \varepsilon, \varepsilon \rangle)} C' \mathit{And}\ C'' \tag{I.46}$$

or that the clause is repaired like so:

$$\sigma \vDash_3^{\epsilon} \#\mathsf{credit} \tag{I.47}$$

Trying to satisfy the main clause first, we apply the *And* rule (I.35) to line I.46, giving us the following sub-formulas:

$$\sigma \vDash_3^{\epsilon} \langle \mathsf{resp1}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{SLA1}), \langle \varepsilon, 24 \rangle \rangle, O(\mathsf{respond}), \bot \rangle \tag{I.48}$$

and

$$\sigma \vDash_3^{\epsilon} \langle \mathsf{resp2}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{SLA2}), \langle \varepsilon, 4 \rangle \rangle, O(\mathsf{respond}), \bot \rangle \tag{I.49}$$

We then apply the rule for obligation clauses (I.30) to both of these cases. The guard in the first case (line I.48), namely $\mathit{isDone}(\mathsf{SLA1})$, will never be true in the environment Γ. The expression $\mathit{lst}(\mathit{isDone}(\mathsf{SLA1}), 3)$ is thus undefined, the antecedent of the implication is false, and the sub-clause is trivially satisfied (we can think of the clause being *skipped*). In the second case (line I.49),

$lst(isDone(\mathsf{SLA2}), 3) = 3$, meaning that we now need to either satisfy the inner action:

$$\sigma \vDash_3^{\tau(\epsilon,\mathsf{resp2},\langle\varepsilon,4\rangle),\mathsf{company}} \mathsf{respond} \tag{I.50}$$

or the reparation of the clause:

$$\sigma \vDash_3^{\epsilon} \bot \tag{I.51}$$

The reparation $\bot$ can of course never be satisfied (rule I.43). We thus apply rule (I.37) to line I.50, which looks for a matching action in the trace which satisfies the given constraints:

$$\sigma \vDash_3^{t_{\mathsf{resp2}}<4,\mathsf{company}} \mathsf{respond} \tag{I.52}$$
$$\text{iff } \exists i : \mathbb{N} \cdot \big(0 \leq i < \mathit{length}(\sigma) \wedge \langle \mathsf{company}, \mathsf{respond}, t\rangle = \sigma(i) \wedge 3 \leq t \wedge \mathit{check}(t_{\mathsf{resp2}} < 4, t)\big)$$

At the point when the obligation clause is enabled (time stamp 3), the clock t_{resp2} is reset to 0, effectively meaning that the satisfying action needs to occur in the trace with a time stamp in the range $3 \leq t < 3 + 4 = 7$. The only event in our trace, $\sigma(0) = \langle \mathsf{company}, \mathsf{respond}, 5\rangle$, does indeed satisfy these requirements, as determined by evaluating $\mathit{check}(t_{\mathsf{resp2}} < 4, 5)$. This completes the derivation for $\sigma \vDash C$.

Had the trace not contained a satisfying action (either its time stamp was outside of the range, or it was missing from the trace altogether), we would have to backtrack to repairing the top-level clause (line I.47). Here, #credit is an example of a cross-reference which needs to be looked up in the model. The example considered here does not include this clause; evaluating it would involve the same procedure followed above.

I.3 Translation to timed automata

I.3.1 Timed automata

In order to enable property-based analysis on contract models, Díaz et al. [28] define a translation from *C-O Diagrams* into *networks of timed automata* (NTAs). A *timed automaton* (TA) [2] is a finite automaton extended with clock variables which increase in value as time elapses, all at the same rate. The model also includes clock constraints, allowing clocks to be used in guards on transitions and invariants on locations, in order to restrict the behaviour of the automaton. Clocks can be reset to zero during the execution of a transition. A *network of timed automata* (NTA) is a set of TAs which run in parallel, sharing the same set of clocks. The definition of NTA also includes a

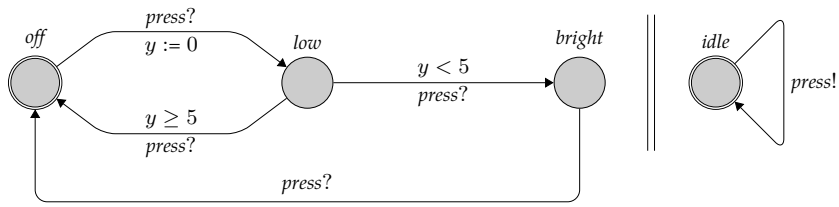


Figure I.7: Timed automata modelling a lamp (left) and a user (right), where y is a clock and *press* is a channel. Double circles indicate initial locations. Taken from [11].

set of channels which allow automata to synchronise.

Figure I.7 shows an example modelling the operation of a simple lamp. The lamp itself has three states, represented as locations *off*, *low*, and *bright*. The user automaton has just a single location, and can synchronise with the lamp via the *press* channel. The first time the user presses a button, the lamp is turned on to *low* and clock y is reset to 0. When the user presses the button again, one of two things may happen. If the user is fast and presses shortly after the first one, the lamp transitions to the *bright* location. Otherwise, if the second press is some time after the first, the lamp turns off. Clock y is used to determine if the user was fast ($y < 5$) or slow ($y \geq 5$). Note that the value of y increases (time passes) while in the *low* location, irrespective of any transitions being taken. The user can press the button randomly at any time or even not press the button at all.

UPPAAL [56] is a tool for the modelling, simulation and verification of real-time systems. It is appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and shared variables — as such making it an ideal tool for working with NTA models. The modelling language used in UPPAAL extends timed automata with a number of features [11], amongst them the concepts of *urgent* and *committed* locations. Put simply, the system does not allow time to elapse when it is in an urgent location. They are semantically equivalent to adding an extra clock x that is reset on all incoming transitions, and having an invariant $x \leq 0$ on that location. Committed locations are an even more restrictive variant on urgent locations. When any of the locations in the current state is committed, the system cannot delay and the next transition must involve an outgoing transition of at least one of the committed locations. UPPAAL also introduces the idea of *broadcast* channels, which allow one sender to synchronise with an arbitrary number of receivers. Any receiver that can synchronise in the current state must do so, but the send can still be executed if there are no receivers (i.e. broadcast sending is never blocking).

The translation of [28] is described in terms of abstract NTA, followed by explanations of how

$$C = \langle n, a, \langle g, i \rangle, O(C_2), R \rangle$$
where $g = g_{low} \wedge g_{upp} \wedge g_{vars}$ and $i = \langle i_{low}, i_{upp} \rangle$

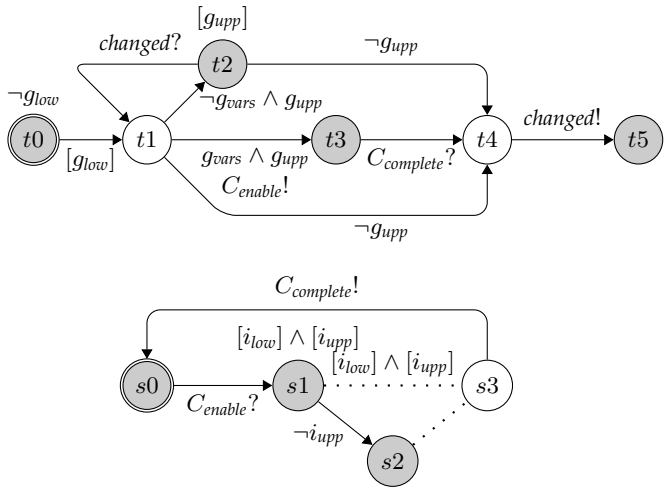


Figure I.8: Translation of an obligation clause (top) into two timed automata: the *thread* (middle) and *main* automaton (bottom). The dotted lines $s1 \cdots s3$ and $s2 \cdots s3$ are replaced with the translations of the complex action C_2, and of the reparation R, respectively. Square brackets indicate inclusive versions of a bound: $[t < 5] = t \leq 5$. Negation of a bound works as expected: $\neg(t < 5) = t \geq 5$. White nodes indicate committed locations.

these can then be encoded in UPPAAL. However despite the similarity of these two domains, there are certain aspects of the NTA of Díaz et al. which cannot be directly implemented in UPPAAL, such as the encoding of urgent edges. Thus, in this work we present a completely revised translation function *trf* from *C-O Diagrams* directly into UPPAAL automata. As there is no difference in abstraction level between NTA and UPPAAL models, we skip the intermediary abstract NTA representation altogether. Our translation avoids the problems present in the previous version, and allows us to take advantage of certain UPPAAL features which are not strictly part of NTA, such as shared integer variables and broadcast channels.

I.3.2 Description

This section highlights the main features of our translation: (i) how guards affect the enactment of a clause, and (ii) how channel synchronisations are used to produce a modular system of automata. We do not go through each case in the translation here; more details can be found in

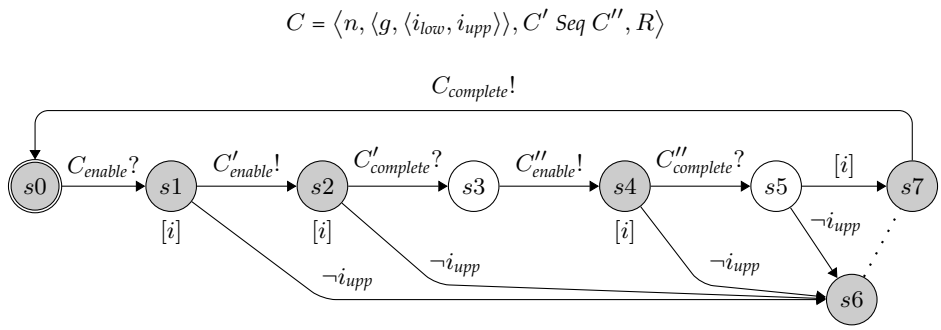


Figure I.9: Main automaton from the translation of a *Seq* refinement (thread automaton not shown here). Inner clauses C' and C'' are translated separately (not shown here) and controlled via their respective *enable* and *complete* channels. The automaton for reparation R is inserted between locations $s6 \cdots s7$. $[i]$ is used as shorthand for the expression $[i_{low}] \wedge [i_{upp}]$.

Appendix I.A.

Figure I.8 shows a generic obligation clause together with the UPPAAL automata produced from its translation. Informally, this is interpreted as follows: when guards g become true, agent a is obliged to do action C_2 within the time frame described by interval i. If the agent does not do action C_2 in time, the reparation clause R will come into effect.

Our translation splits 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 the middle of Figure I.8. The guard from the original clause is separated into lower and upper bound timing constraints (g_{low} and g_{upp}, respectively) and variable constraints (g_{vars}). First the lower bounds must be satisfied in order to progress in the automaton. The variable constraints 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* locations ($t1$ and $t2$, respectively). The *check* location is committed, meaning that no time can elapse while in this location. Each time a clause reaches a completed state, a broadcast signal is sent on the channel *changed* 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. This automaton, representing the inner obligation, is shown in Figure I.8 (bottom). 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 location (where it could possibly be re-triggered), while the thread automaton reaches a final end location.

As a further example, Figure I.9 shows the main automaton produced from translating a clause containing a *Seq* refinement. This demonstrates the use of modularity in the translation, where each sub-clause is activated using channel synchronisation rather than in-lining all the automata together. This has benefits not only for the modelling process but also when it comes to analysis.

Simulating actions. The function of the automata described above is to model clauses which essentially wait for actions to occur, and then react accordingly. In order to simulate the firing of such actions, a simple non-deterministic automaton is created for each action in the set Σ which can randomly fire at any point (given that the action has not already fired). For more about this, see case I.37 in Appendix I.A.5.

Implementation. A complete implementation of this translation has been built using Haskell.[4] It includes a definition of a data type for our extended *C-O Diagrams*, the ability to check whether a given *C-O Diagram* is well-formed, and a working translation function which produces a Uppaal-readable XML file as output. This tool was used in the application of our method to a case study, covered in Section I.5.

I.3.3 Correctness of the translation

The previous section informally describes the translation function *trf*, which converts a *C-O Diagram* into a Uppaal model. In order to trust any analysis performed on this translated model, we want to be certain that the translation itself is correct with respect to the trace semantics defined in Section I.2.3. We approach this by relating our trace semantics for *C-O Diagrams* with that of Uppaal.

David et al. [27] define a trace of a Uppaal model as a sequence of *configurations*, where a configuration describes the current locations 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 (or *deadlocked*, i.e. where no further transitions are possible), where each consecutive configuration can be reached from its previous one in a single step. These definitions have been reproduced in I.A.2.

[4] Full source code of this implementation can be found at the URL below:
http://remu.grammaticalframework.org/contracts/jlamp-nwpt2015/

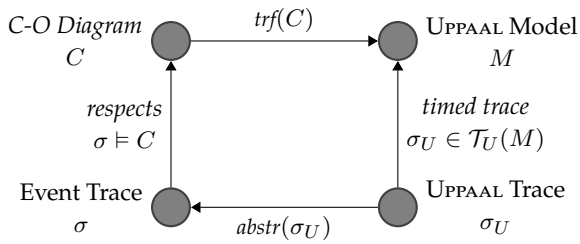


Figure I.10: Relations between *C-O Diagram* and UPPAAL model and trace representations.

Let $\mathcal{T}_U(M)$ denote the set of *timed traces* for a UPPAAL model M. This set includes all timed traces which are either infinite or maximally extended. 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 C. We shall indicate this set with $\mathcal{T}_U^C(M)$. Let us assume an abstraction function $\mathit{abstr} : \mathcal{T}_U \to \mathcal{T}$, which transforms a UPPAAL trace σ_U into an event trace σ by extracting the time stamps at which each action was performed. With these elements in place, visualised in Figure I.10, we state the following theorem relating our trace semantics for *C-O Diagrams* with UPPAAL model traces:

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

$$\sigma \vDash C \text{ iff } \exists \sigma_U \in \mathcal{T}_U^C(M) \cdot \sigma = \mathit{abstr}(\sigma_U)$$

Proof Sketch. The proof is performed by structural induction over the *C-O Diagram* syntax (see Figure I.5). For each case, we consider the translation into a UPPAAL model by the *trf* function. Using the formalisation of UPPAAL models and their trace semantics given by David et al. [27], 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 Appendix I.A. □

I.4 Analysis

The purpose of formalising normative documents as models is to enable automated analysis, by which we mean running queries of different kinds against our model. Needless to say, this task can only be meaningful if one pre-supposes that the contract model is an accurate representation

Table I.2: Predicates used in syntactic analysis.

Predicate	Holds when
isObl/*isFor*/*isPer*	clause is an obligation/prohibition/permission
isAnd/*isOr*/*isSeq*	clause contains conjunction/choice/sequence
hasUpperBound	clause has an upper bound in its interval
hasRep	clause has a reparation which is not $\top$
agentOf(a)	agent a is responsible for clause
hasAction(x)	action x appears in the body of clause

of the original text. Addressing this concern is beyond the scope of the current work. We separate the kinds of analysis possible into two main classes, which differ based on the method used to process queries of that type.

For examples of these types of analysis in use, see the case study in Section I.5.

I.4.1 Syntactic analysis

Certain kinds of queries can be checked by traversing the structure of a contract model, such as listing the permissions for a particular agent or identifying obligations without constraints or reparations. We refer to these as *syntactic* queries as they can be computed purely from the syntactic structure of the model.

We begin by introducing *predicates* over single clauses. For example, the predicate *isObl* holds if a given clause is an obligation. Predicates may also take additional arguments, such as *agentOf*(a), which is true if agent a is responsible for a clause. Table I.2 lists the basic predicates defined over clauses. These can be combined using the standard propositional operators to build a general property language over clauses. 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 of a given agent contained in a contract. We refer to these as *queries*, since they are the result of querying a contract with clause properties. The query function $\mathcal{Q}$ (I.53) returns the set of all clauses in the contract that satisfy the predicate provided as the first argument. The query function has also been implemented as a command-line tool in Haskell, together with the translation function from the previous section.

$$\mathcal{Q} : (C \to \mathit{Boolean}) \to K \to \mathcal{P}(C) \tag{I.53}$$

$$\mathcal{Q}\big(\psi, \{\langle C^1, T^1\rangle, \ldots, \langle C^n, T^n\rangle\}\big) = \{C^i \mid 1 \leq i \leq n,\ \psi \text{ holds w.r.t. } C^i\}$$

I.4.2 Semantic analysis

Other kinds of queries cannot be answered simply by looking at the structure of the model — for example, checking whether performing a given action at a particular time will satisfy a contract. Determining this must take into consideration not only the constraints of a single clause, but the evolution of the contract as whole as other actions are performed, new clauses are enabled and others expire. We refer to such queries as *semantic* because verifying them requires taking into account the operational behaviour of a contract model, rather than just its structure. This is done by converting a contract model into a network of timed automata in UPPAAL (as described in Section I.3) and using model checking techniques.

This approach requires that the query itself is encoded as a property in a suitable temporal logic which the model checker can process. In the case of UPPAAL, the property specification language is a subset of TCTL [11]. We shall look here at the aspects of this language which are relevant to the analysis of our contract models.

The automata systems produced by our translation are never infinite, in the sense that we have a clear definition of when we consider the contract to have reached a final state.[5] Thus, the main temporal operators that are of interest to us are those for *possibility* and *invariance*.[6]

Possibility. The property $\exists \Diamond \psi$ is satisfied if there exists some trace through the system of automata for which the expression ψ holds at some point in the sequence. Such a property can be used to test whether under a given contract, it is possible for a certain action to be performed or state of affairs to occur.

Invariance. On the other hand, the property $\forall \Box \psi$ will be satisfied if for every possible trace, the expression ψ can be shown to hold at all configurations in the sequence. This is the typical way to describe safety properties.

The expression part of a property consists of a predicate over the current configuration of the system — values of variables, comparison over clocks, and the current locations of the automata. The automaton representation is at a lower level of abstraction than the original contract model, and encoding a contract query as a temporal property may require an understanding of the translation function and resulting network of automata. To mitigate this, the guard predicates from the *C-O Diagram* syntax (Figure I.5, rule I.14) are also implemented as functions in the

[5] Any configuration where all the main contract clauses satisfy the *isComplete* predicate.

[6] Conceptually these can be expressed in terms of one another, i.e. $\forall \Box \psi \equiv \neg \exists \Diamond \neg \psi$. However the UPPAAL specification language does not allow negation of arbitrary queries, which is why they exist as separate operators.

translated Uppaal model. Clause names and action identifiers are also defined as global variables in the system, making them available for use within properties, e.g. *isComplete*(clause) or *isDone*(agent.action). To refer to the clocks associated with clauses and actions, these identifiers can be used as indices into a special array of clocks, e.g. *Clocks*[clause] or *Clocks*[agent.action]. Clock values can be subtracted from each other and compared with constant integer values to form a valid expression. Simple Boolean expressions can be combined with propositional operators to form complex ones.

These are the main components necessary for constructing semantic queries relevant to our contract models. Expressions involving template locations or comparisons with any other state variables should not be needed, in the sense that such low-level information on the state of NTA would not correspond to anything meaningful in terms of the original contract.

I.5 Case study

As a case study for demonstrating our approach to contract analysis, we have chosen a service level agreement (SLA) from the hosting company LeaseWeb USA, Inc.[7] The original agreement is a 6-page document, divided into 12 sections with a total of 59 clauses, most of which consisting of multiple sentences. For demonstration purposes, we here focus on one of the chapters from the full agreement, which we have abridged into 4 clauses (see Figure I.11). This has been done in the interest of conciseness, so that the example is not made unnecessarily long by overly verbose sentences or unrelated clauses. More details about the original document, together with the unabridged version of the chapter considered here, can be found in Appendix I.B.

I.5.1 Model

Building a *C-O Diagram* model from this example requires each sentence in the original text to be encoded as a formal clause. While one natural language sentence often corresponds to a single clause in the model, there can be many exceptions to this. Cases involving choice or specifying multiple actions must often be broken down into sub-clauses using refinement. Sequence is often something that is not explicitly expressed in a contract, and the modeller must identify the implicit sequence that may exist between clauses. Special care is also required when modelling guards and timing constraints, because of the various indirect ways in which they may appear. In short, the modelling task is a non-trivial one which requires a proper understanding of the original text, as well as solid knowledge of the formalism being used.

[7] The authors have no connection with LeaseWeb USA, Inc.

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 Credit shall be due if the Customer is not reachable.

Figure I.11: Abridged chapter from the SLA from LeaseWeb USA, Inc. covering hosting services (see Appendix I.B), which serves as the original contract in this case study.

$$
\begin{aligned}
C = \{ & \\
& \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 \mathit{isDone}(\mathsf{cust\_auth}), \langle \varepsilon, \varepsilon \rangle \rangle, P(\mathsf{refuse}) \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 \mathit{isDone}(\mathsf{sla1}), \langle \varepsilon, 24 \rangle \rangle, O(\mathsf{respond}), \#\mathsf{credit} \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{resp2}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{sla2}), \langle \varepsilon, 4 \rangle \rangle, O(\mathsf{respond}), \#\mathsf{credit} \rangle, \mathit{Aux} \big\rangle, \\
& \big\langle \langle \mathsf{credit}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{reach}), \langle \varepsilon, \varepsilon \rangle \rangle, O(\mathsf{give\ credit}), \top \rangle, \mathit{Aux} \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 I.12: Contract model for the normative text shown in Figure I.11.

When done completely manually, this can require a significant effort on the part of the modeller. However, suitable tool support can be of a great help in this regard. In our own previous work we introduce some such front-end tools ([20], [17]) for facilitating the modelling process. The construction of the contract model for the current case study was thus carried out using these tools. As a first step, we apply the ConPar extraction tool, giving us an initial list of clauses in a tabular format, separated into agent, action and modality, and including some refinements. This representation is then manually post-edited to fix the parts of the contract which were incorrectly parsed. From here, we can automatically generate a *C-O Diagram* from our tabular representation, visualising the hierarchical structure of the model and allowing us to make further adjustments, before finally exporting the formal model which we use below.

Figure I.12 shows the case study model we are concerned with, presented as an expression in our language described earlier. The contract is built from *main* and *auxiliary* clauses, linked together using cross-referencing (#). The primary clause is request, which 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 (Figure I.11). In our model each SLA level is treated individually, as in the obligation clauses (resp1 and resp2). Both are dependent on the level which has been chosen by the customer in chooseSLA, using the *isDone* predicate as a guard, making them mutually exclusive. The response time targets are then encoded as intervals on the corresponding obligations, e.g. $\langle \varepsilon, 24 \rangle$ enforces that the response to a basic-level SLA request is completed within 24 hours.

Clause 1.5 dictates 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, namely that the customer has fulfilled its obligation to be reachable (Clause 1.6). This is encoded as a standalone obligation reach, whose completion is given as a guard in the credit clause.

Size. The model described here contains 2 agents, 11 actions and 11 top-level clauses. The UPPAAL system produced from its translation consists of 33 templates and corresponding processes, with a total of 160 locations and 172 transitions. It uses 35 channels, 28 clocks, and 108 Boolean variables. A simple optimisation pass is then applied which removes transitions without any labels and merges the respective source and target locations. After removing a total of 30 such transitions, the resulting minimised system contains 130 locations and 142 transitions.

I.5.2 Syntactic analysis

To begin with, we can inspect the model syntactically to identify clauses in our contract with potentially problematic characteristics.

Missing reparations. For example, the following query returns all clauses with no reparation:

$$\mathcal{Q}(\neg \mathit{hasRep}, C) = \{\mathsf{request}, \mathsf{req\_info}, \mathsf{cust\_auth}, \mathsf{resp}, \mathsf{credit}, \mathsf{reach}\}$$

When building the model, we treat $\top$ as the default reparation when none is specified. It is not surprising that almost all the clauses are returned here, however this can be a useful first step in identifying clauses in the contract which can be violated without any repercussions. By taking the names in the query response and tracing them back to the original text, we find that all clauses except 1.4 do not in fact specify any reparations. These may be intended to be handled by a catch-all clause covering the violation of any part of the contract, or they may be intentionally left under-specified for legal reasons.

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

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

Consider the obligation clause credit. Even if the company may be obliged to credit the customer, without any time constraints they can effectively avoid doing this. It is quite common for normative documents such as this to contain clauses without specific time restrictions, but this often leads to problems when it comes to formalising them. As in the previous case, a query such as this can help the modeller to be more specific about acceptable time frames for clause satisfaction.

Note that even though the clause names returned here are different from those in the previous example, they still correspond to the same natural language 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. Finally, we may wish to search for the clauses in the contract which provide a choice to the customer. This can be done as follows:

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

This query returns a single clause chooseSLA, indicating the customer's choice of service level as described in Clause 1.4.

As demonstrated here, these simple predicates over clauses can be combined in various ways to produce different kinds of useful queries on our contract models. This method can be used to quickly highlight or filter out clauses having certain characteristics. Moreover, the execution of these kinds of queries is negligibly quick and linear in the size of the model.

I.5.3 Semantic analysis

We next show some examples of how we can analyse our case study contract by verifying temporal properties against the translated version of the model in UPPAAL.

Consider the bits of information required to make a request, as listed in Clause 1.3 (Figure I.11). We would like to verify whether it is possible for a customer to create a request without providing all the necessary information. This can be expressed with the following property:

$$\exists \Diamond \mathit{isComplete}(\mathsf{request}) \wedge \neg \mathit{isDone}(\mathsf{customer.contact\_details}) \tag{I.54}$$

Note how we are using predicates over the status of both clauses and actions. Verifying this in UPPAAL gives a result of SAT — essentially saying that it *is* in fact possible for a request to be completed even when the customer does not provide its contact details. This is not what we expect, so we consult the symbolic trace provided by the model checker as a counter-example.

A symbolic trace describes the sequence of transitions taken through a system of automata, together with the constraints on its variables and clocks at each point. By carefully stepping through the trace provided and following the automata transitions one by one,[8] we discover that the req_info clause can still reach a final location when its actions aren't completed, because of an unguarded transition corresponding to the $\top$ reparation. This points to a problem with the model. If we change the reparation for the clause to $\bot$, re-run the translation and then re-verify the property, we then get the expected result of UNSAT. This property could also be rewritten as an invariant:

$$\forall \Box \mathit{isComplete}(\mathsf{request}) \implies \mathit{isDone}(\mathsf{customer.contact\_details}) \tag{I.55}$$

In this case we obtain the opposite result, i.e. SAT. There is negligible difference in the time and

[8] The trace produced in this case consists of 12 transitions and 13 states, each of which describing the current location of 33 processes, 28 clock constraints and 108 variable valuations. Reproducing this trace here would take up a lot of space and would not be conducive to explaining the example. The interface provided by the UPPAAL tool makes stepping through traces a lot more manageable than just looking at the raw data.

space required to verify this version of the property.

Let us consider another example. When it comes to giving service credit to the customer (Clause 1.5), we may wish to verify that this is only given when the correct criteria are met. We come up with the following pair of queries which test this with respect to the basic support level:

$$\forall \Box\, \mathit{isComplete}(\mathsf{request}) \wedge \mathit{isDone}(\mathsf{resp1}) \wedge \mathit{isDone}(\mathsf{reach}) \\ \wedge\, \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] > 24 \\ \implies \mathit{isDone}(\mathsf{credit}) \tag{I.56}$$

$$\forall \Box\, \mathit{isComplete}(\mathsf{request}) \wedge \mathit{isDone}(\mathsf{resp1}) \\ \wedge\, \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] < 24 \\ \implies \neg \mathit{isDone}(\mathsf{credit}) \tag{I.57}$$

Note that we used the difference between two clocks to determine the relative time at which the response occurred. These properties check that credit is *always* given when the response time exceeds 24 hours, and that it is *never* given when the response time is less than 24 hours. Running both queries returns a SAT result as expected.

Execution Times. As is typical with model checking, the time and space required for verifying properties can be a potential problem. Table I.3 shows the space and time requirements for the verification of the properties described here. One can see that even for this small case study, verification time is in the order of tens of minutes when an exploration of the entire search space is required (counter-examples are generally found a lot quicker).

In an attempt to improve on this, we re-verified the same properties a second time with a slightly reduced version of the system, where the processes for some unrelated clauses were deactivated (those pertaining to the clauses cust_auth and req_refuse). As shown in Table I.3, the improvement obtained was dramatic. By reducing the number of running processes from 33 to 27 (18% decrease), we observed a decrease of over 99% for verification time and a decrease of over 98% for memory usage. These results indicate that deactivating parts of the translated contract model which are not relevant to the current property can have an enormous effect on the verification. We discuss this further in our conclusions in Section I.7.

Table I.3: Resources required for verifying the properties in Section I.5.3, for the full system of automata and for the reduced version of it, respectively. *States* is the number of states explored during the verification; *time* is given in the format MM:SS, and *space* is given in MiB.

Property	Result	Full system			Reduced system		
		states	*time*	*space*	*states*	*time*	*space*
(I.54)	UNSAT	87,353,719	25:56	5,273	770,023	00:10	87
(I.55)	SAT	87,353,719	26:15	5,273	770,023	00:11	89
(I.56)	SAT	119,371,443	45:22	7,455	770,023	00:22	89
(I.57)	SAT	119,371,443	45:19	7,455	770,023	00:22	89

I.6 Related work

C-O Diagrams were introduced by Martínez et al. in [60], and further refined in [28]. Our work is heavily based on their formalism, yet we have made significant contributions to their work. Building a fully working implementation of the translation from *C-O Diagrams* into UPPAAL automata has led us to modify their definition in various ways (as described in Section I.2.2). In particular, our translation has a stricter interpretation of guards, ensuring that *if* a guard becomes true during the specified time frame, then the corresponding transition *must* be taken.

The trace semantics defined in Section I.2.3 is completely new for the *C-O Diagram* formalism, intentionally creating a separation between the *intended* interpretation of a contract model and its actual behaviour when translated into timed automata. We follow the approach of [32] where a trace semantics is defined for the contract language $\mathcal{CL}$ [78]. The major difference in our work is that *C-O Diagrams* includes the concept of time, whereas $\mathcal{CL}$ does not. Because of this, the rules in our trace semantics cannot simply consume elements of a trace sequentially as in [32], but must search through the entire trace looking for events which satisfy the given conditions.

Llana et al. [57] re-use the visual model of *C-O Diagrams* for a different language for describing contract relationships. Their language, based on process algebra, includes an operational semantics and the definition of a simulation relation, in order to be able to determine whether an implementation of a system follows the rules established by a given contract. These semantics do not deal with event traces as in our work, and their focus is not on query-based contract analysis.

Our ultimate goal is to produce a usable end-to-end system for performing contract analysis. To this end, we also refer the reader to Camilleri et al. [20], where we focus on front-end aspects of working with *C-O Diagrams*. This includes going into the issues around modelling, introducing a tool for building 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.

AnaCon [4] is a similar framework for the analysis of contracts, based on the contract logic $\mathcal{CL}$ [78], which allows for the detection of contradictory clauses in normative texts using the CLAN tool [32]. By 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.

Pace and Schapachnik [71] introduce the Contract Automata formalism 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.

In [58] Marjanovic and Milosevic also defend a deontic approach for formal modelling of contracts, paying special attention to temporal aspects. They distinguish between three different kinds of time: absolute, relative and repetitive. The two first kinds are supported by *C-O Diagrams*, but repetition in general is not a part of our formalism. They also introduce visualisation concepts such as *role windows* and *time maps* and describe how they could be used as decision support tools during contract negotiation.

Wyner [90] 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. The tool is designed as an abstract, flexible framework in which alternative definitions of the deontic concepts can be expressed and exercised. However its high level of abstraction and lack of temporal operators make it limited in its application to processing concrete contracts. In particular, the work is focused on logic design issues and avoiding deontic paradoxes, and there is no treatment of query-based analysis as in our work.

There is also considerable work in the representation of contracts as knowledge bases or *ontologies*. The *LegalRuleML* project [6] embodies one of the largest efforts in this area by providing a rule interchange format for the legal domain, allowing the contents of the legal texts to be represented in a machine-readable format. The format aims to enable modelling and reasoning that let users evaluate and compare legal arguments constructed using their rule representation tools.

A similar project with a broader scope is the *CEN MetaLex* language [14], 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)* [68] uses a CNL to provide a fixed vocabulary and syntactic rules for expressing terminology, facts, and rules for business documents. As with most CNLs, the goal is to allow natural descriptions of the conceptual structure and operational controls of a business, which at the same time can be represented in predicate logic and converted to machine-executable form. SBVR is geared towards business rules, and not specifically at the kinds of normative texts in which we are interested.

I.7 Conclusion

This work presents a number of extensions to the *C-O Diagrams* formalism for normative texts, together with a revised translation to Uppaal automata, and a new 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 argue for the correctness of the translation with respect to the trace semantics. We also take a detailed look at the kinds of analysis possible on these models, distinguishing between queries which can be answered by syntactic means, and semantic queries which rely on the Uppaal model checker. These methods are then applied to a small case study taken from a real-world normative document.

Scalability

It is well-known that model checking may easily become intractable for non-trivial models, and the time and memory demands of verification can be very sensitive to the size of the automata, the number of clocks, and the use of channel synchronisations. The optimisations described in Section I.5.1 are currently performed manually, and as such our translation algorithm does not optimise the automata it produces. The result is that a translated system may contain unnecessarily many locations and/or transitions which negatively affect the verification time by increasing the number of states which need to be explored. A thorough investigation of possible optimisations and their effect on performance is regarded as important future work.

Another highly relevant method for reducing verification time is to identify the parts of the NTA which are irrelevant to the current query, and temporarily disable them before running the model checker. The reduced version of our case study in Section I.5.3 shows that even a modest reduction in the size of the system can yield dramatic improvements on the time and memory requirements of verification. While this may be hard to do for NTA in general, as *C-O Diagrams* are domain-specific and represent a higher level of abstraction, it should be much easier to identify independent clauses in the contract model and disable them *before* the translation to

NTA. We see this as a promising method of avoiding the potential scalability problems with using model checking, and intend to explore how this can be incorporated into our contract analysis framework. We also point out that scalability is not an issue for the syntactic analysis, which is linear in the size of the model.

Our trace semantics in Section I.2.3 defines the *respects* relation between traces and contracts, however we do not provide a concrete algorithm for it which is independent of the translation to NTA. Thus we have no objective measure of the computational complexity of computing this relation, though we would not expect it to be expensive given that we are dealing with concrete traces and not considering the space of all possible traces, as is the case when verifying the NTA.

Future work

We show here that analysis of normative documents is possible with the right formalisation and querying system. The task of formalising a contract from a natural language text is not trivial, and increasing the level of automation in this process both reduces the workload for the user and creates a higher level of predictability. This work forms the core of a larger toolkit for working with contracts, which addresses other facets of this task not described in the current work.

The natural language aspects of contract modelling form an equally important part of our overall framework. We already have some prototype front-end tools for automatically producing partial models from natural language documents using entity extraction [17], as well as for building contract models graphically and using controlled natural language [20].

As with the semantic gap faced in the modelling process, a similar gap exists when it comes to constructing syntactic and semantic queries, as well as in the interpretation of their results. Thus another strand of current research involves the identification of query patterns and the definition of a CNL for analysis. This work will also cover the processing of symbolic traces returned by Uppaal and verbalising them back into natural language.

As these different strands of development progress towards maturity, our ultimate goal is to combine all elements of this work together into a user application specifically for the end-to-end analysis of normative documents. Further details about the case study and our tools, including source code, can be found at `http://remu.grammaticalframework.org/contracts/jlamp-nwpt2015/`.

Acknowledgements

We are very grateful to Gabriele Paganelli and Filippo Del Tedesco for their contributions to earlier versions of this work.

I.A Proof of correctness

I.A.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 I.2.3. We do this by structural induction over the syntax in Figure I.5, 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.

I.A.2 UPPAAL trace semantics

David et al. [27] give a formalisation for Uppaal models, together with a definition of their trace semantics. We briefly repeat their definitions here.

Definition 3 (UPPAAL process). *A 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). *A 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 a 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*

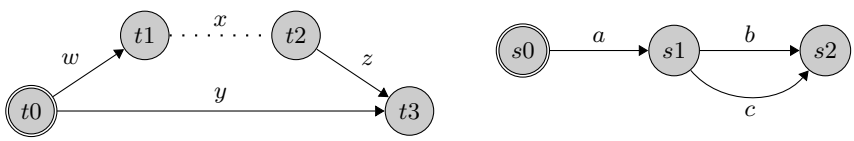
1. *all locations in the initial configuration are the initial locations for their respective processes,*
2. *all clocks in the initial configuration evaluate to 0,*
3. *if the sequence is finite, then at the last configuration no further steps are enabled (system is maximally extended/deadlocked),*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

I.A.3 Notes and notation

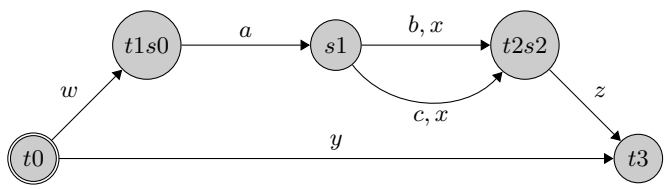
In each case of the proof, we present the automata resulting from the translation in graphical form, simply because they are more concise and easier to read than formulas. Similarly, details about variable and channel declarations are omitted for brevity. The following is a legend to the conventions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

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

3. A guard is split up into:
   (a) lower bound time constraints g_{low} (i.e. using $>$)
   (b) upper bound time constraints g_{upp} (i.e. using $<$)
   (c) non-temporal (variable) constraints g_{vars}
4. An interval i is composed of a lower and upper bound $\langle i_{low}, i_{upp} \rangle$.
5. Square brackets indicate inclusive versions of a bound: $[t < 5] = t \leq 5$.
6. $[i]$ is used as shorthand for the expression $[i_{low}] \wedge [i_{upp}]$.
7. The symbol $\neg$ indicates the negation of constraints: $\neg(t < i) = t \geq i$.
8. Constraints on a location indicate invariants.
9. The function calls $reset(n)$, $vio(n)$, $done(n)$, $sat(n)$, and $skip(n)$ are abbreviated to r, v, d, s, sk respectively, where n is the name of the clause being translated.
10. 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.
11. A dotted line indicates a placeholder where another automaton (obtained through translation of a sub-clause) should be inserted. For example, consider the following two automata:



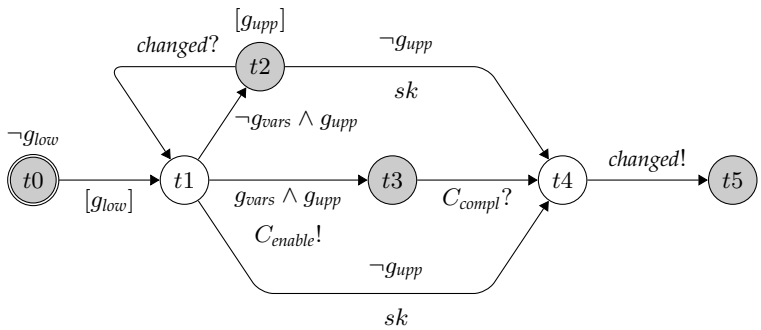
As the t automaton contains a dotted line $t1 \cdots t2$, the entire s automaton could be inserted between these two locations, resulting in the following:



All transition labels are preserved. The label on the dotted line is merged with all transitions in the sub-automaton which end in its final location. While UPPAAL automata cannot be marked with an end location per se, all the automata produced by our translation which are inserted as described here will have exactly one location which is clearly final (no outgoing transitions).

I.A.4 Thread automaton

All top level clauses (cases I.30–I.36, Figure I.6) 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 constraint *True*. Each time a clause reaches a completed state, 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 initially *False*: Wait in $t0$ until g_{low} is *True*, at which point the invariant on $t0$ will cause a transition to $t1$.

(b) g_{vars} is *True* upon reaching $t1$: Transition to $t3$ immediately, activating main automaton.

(c) g_{vars} is initially false but 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} never becomes *True* before g_{upp} expires: Wait in $t2$ until g_{upp} expires, then transition to $t4$, skipping main automaton.

(e) g_{upp} is already expired upon reaching $t1$: Transition to $t4$ immediately, skipping main automaton.

I.A.5 Case analysis

Note that the case numbers here correspond to the rules in the trace semantics in Section I.2.3 (Figure I.6).

Case I.29: Contract

$$\sigma \vDash \{\langle C^1, T^1\rangle, \ldots, \langle C^n, T^n\rangle\} \text{ iff } \bigwedge_{\substack{1\leq i\leq n,\\ T^i = Main}} \sigma \vDash_0^\epsilon C^i$$

Event traces. Traces respecting this formula must respect each of the individual *Main* 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 n from its initial state to one in which no further steps are possible, and in which *isComplete*(n) is *True*.

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

Case I.30: Obligation

$$\sigma \vDash_{t_0}^c \langle n, a, \langle g, i\rangle, O(C_2), R\rangle$$
$$\text{iff } \left(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\right) \text{ implies } \left(\sigma \vDash_t^{\tau(c,n,i),a} C_2 \text{ or } \sigma \vDash_t^{\tau'(c,n,i)} R\right)$$

Event traces. We consider the following cases:

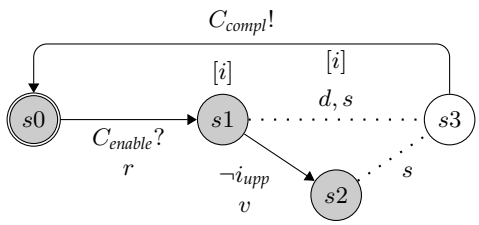
(a) Guards g are never *True* ($\nexists t$): the obligation is not enacted and thus trivially respected.

(b) Guards g become *True* ($\exists t$): 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 constraints $\tau(c, n, i)$.

(ii) The entire reparation clause R is completed after interval i has expired but while the other constraints still hold, $\tau'(c, n, i)$.

Translation. All automata from the translation of R, one thread automaton (see I.A.4) and one main automaton as follows, where dotted line $s1 \cdots s3$ is filled with the translation of C_2 and $s2 \cdots s3$ 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) The automaton progresses through $s1 \cdots s3$ while interval i holds, respecting the translation of C_2.

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

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

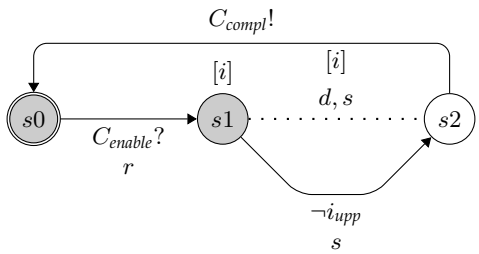
Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then either C_2 is respected within the interval i, or R is respected after i has expired.

Case I.31: Permission

$$\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i \rangle, P(C_2) \rangle$$

Event traces. Any trace will respect a permission.

Translation. One thread automaton (see I.A.4) and one main automaton as follows, where dotted line $s1 \cdots s2$ 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) The automaton progresses through $s1 \cdots s2$ while interval i holds, respecting the translation of C_2.

  (ii) Interval i expires and the transition $s1 \rightarrow s2$ is taken. 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 I.32: Prohibition

$$\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i\rangle, F(C_2), R\rangle$$
$$\text{iff } \left(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\right) \text{ implies } \left(\sigma \vDash^{\tau(c,n,i),a}_{t} C_2 \text{ implies } \sigma \vDash^{c}_{t} R\right)$$

Event traces. We consider the following cases:

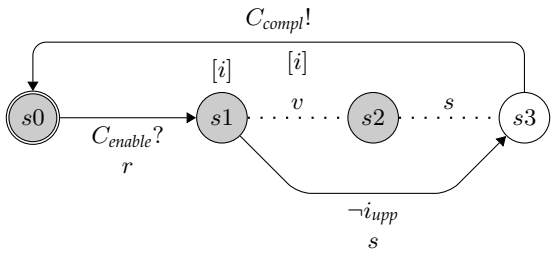
(a) Guards g are never *True* ($\nexists t$): the prohibition is not enacted and thus trivially respected.

(b) Guards g become *True* ($\exists t$): 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 constraints $\tau(c, n, i)$, followed by reparation clause R being completed while the inherited constraints c hold.

  (ii) The actions in C_2 are not performed while the combined constraints hold.

Translation. All automata from the translation of R, one thread automaton (see I.A.4) and one main automaton as follows, where dotted line $s1 \cdots s2$ is filled with the translation of C_2 and $s2 \cdots s3$ 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) The automaton progresses through $s1 \cdots s2$ while interval i holds, respecting the translation of C_2, followed by $s2 \cdots s3$, respecting the translation of R,

  (ii) Interval i expires and transition $s1 \rightarrow s3$ 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 within the interval i, then R must necessarily be respected too.

Case I.33: Refinement

$$\sigma \vDash^{c}_{t_0} \langle n, \langle g, i \rangle, C_1, R \rangle$$
$$\text{iff } \left(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\right) \text{ implies } \left(\sigma \vDash^{\tau(c,n,i)}_{t} C_1 \text{ or } \sigma \vDash^{\tau'(c,n,i)}_{t} R\right)$$

Event traces. We consider the following cases:

(a) Guards g are never *True* ($\nexists t$): the clause is not enacted and thus trivially respected.

(b) Guards g become *True* ($\exists t$): the clause can be respected in one of two ways:

  (i) The inner clause C_1 is respected while combined constraints $\tau(c, n, i)$ hold (as covered in cases I.35–I.36 below).

(ii) The inner clause C_1 is not respected and the reparation clause R is completed after interval i has expired but while the other constraints still hold, $\tau'(c, n, i)$.

Translation. All automata from the translation of R, one thread automaton (see I.A.4) and one main automaton as described in cases I.35–I.36 below, containing the thread from the translation of R.

UPPAAL traces. In order to reach a state where the clause 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, and one of the following occurs:
  (i) The main automaton completes by taking a transition labelled s while interval i holds.
  (ii) The main automaton takes a transition labelled v to a violation state, following by a reparation transition labelled s into a final state.

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

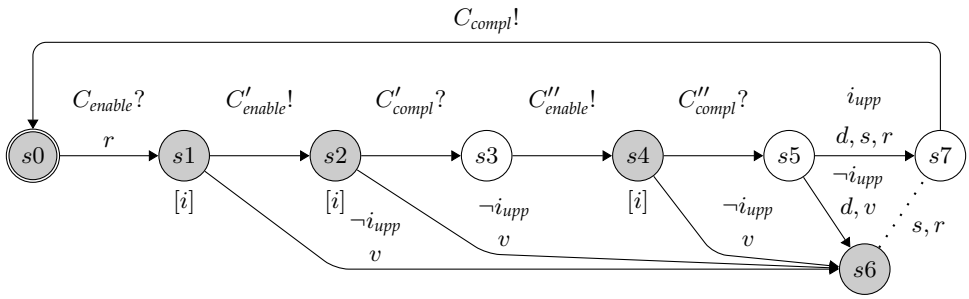
Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then either C_2 is respected within the interval i, or R is respected after i has expired.

Case I.34: Sequence

$$\sigma \vDash^{c}_{t_0} C' \mathit{\ Seq\ } C''$$
$$\text{iff } \exists j : \mathbb{N} \cdot \left(0 \leq j \leq \mathit{length}(\sigma) \wedge \sigma(..j) \vDash^{c}_{t_0} C' \wedge \sigma(j..) \vDash^{c}_{t_0} C''\right)$$

Event traces. Traces can be divided in two, such that first sub-trace respects C' and the second sub-trace respects C'' while constraints c hold.

Translation. All automata from the translations of C' and C'', and one main automaton as follows, where $s6 \cdots s7$ is filled with the thread from the translation of R (from parent clause) and $i = c$.



UPPAAL traces. The sub-automata for C' and C'' are enacted in sequence, such that C' must be completed before C'' is enacted. A trace of configurations must either satisfy both of these in order, within the interval i, or the translation of R after i has expired. The synchronisation with C_{compl} means that both thread and main automaton should reach a maximally extended state together.

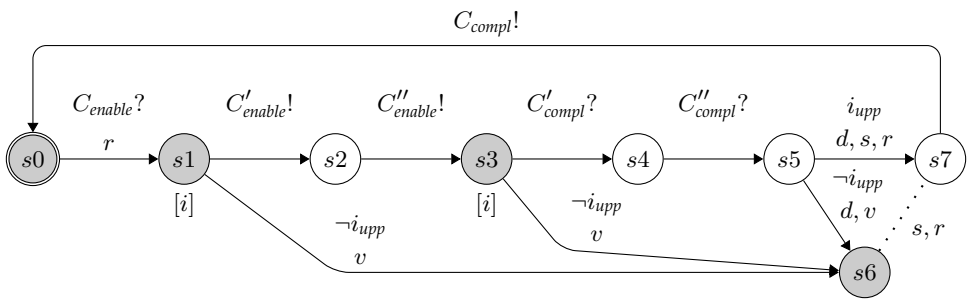
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in order, within the interval i, or that the reparation is respected.

Case I.35: Conjunction

$$\sigma \vDash^{c}_{t_0} C' \textit{ And } C'' \text{ iff } \sigma \vDash^{c}_{t_0} C' \text{ and } \sigma \vDash^{c}_{t_0} C''$$

Event traces. Traces must respect both C' and C'' individually while the constraints c hold.

Translation. All automata from the translations of C' and C'', and one main automaton as follows, where $s6 \cdots s7$ is filled with the thread from the translation of R (from parent clause) and $i = c$.



UPPAAL traces. The sub-automata for C' and C'' are both enacted (the order is not significant since the intermediate location $s2$ is committed) ensuring that a trace of configurations must either satisfy both of these within the interval i, or the translation of R after i has expired. The synchronisation with C_{compl} means that both thread and main automaton should reach a maximally extended state together.

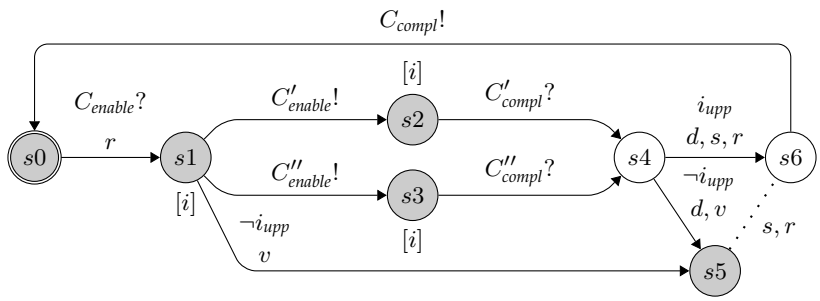
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in any order, within the interval i, or that the reparation is respected.

Case I.36: Choice

$$\sigma \vDash^{c}_{t_0} C' \mathit{Or}\ C'' \text{ iff either } \sigma \vDash^{c}_{t_0} C' \text{ or } \sigma \vDash^{c}_{t_0} C''$$

Event traces. Traces must respect either C' or C'' while the constraints c hold.

Translation. All automata from the translations of C' and C'', and one main automaton as follows, where $s5 \cdots s6$ is filled with the thread from the translation of R (from parent clause) and $i = c$.



UPPAAL traces. Only one of the sub-automata for C' and C'' can be enacted, introducing non-determinism at location $s1$. A trace of configurations must either satisfy one of these within the interval i, or the translation of R after i has expired. 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 within interval i, or that the reparation is respected.

Case I.37: Simple action

$$\sigma \vDash^{c,a}_{t_0} x \text{ iff } \exists j : \mathbb{N} \cdot \big(0 \leq j < \mathit{length}(\sigma) \wedge \langle a, x, t\rangle = \sigma(j) \wedge t_0 \leq t \wedge \mathit{check}(c, t)\big)$$

Event traces. Traces must contain an event involving agent a and action x with a time stamp that is later than or equal to t_0 and which complies with the constraints c.

Translation. An action is simply a transition which can only be taken when the corresponding action $a.x$ has been performed (below left). Each action also gets a corresponding *doer* automaton which sets the status of that action to *done* (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 the transition where the status of action $a.x$ is set to *done*.

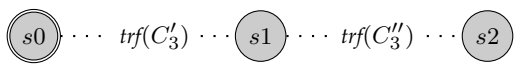
Argument. Both sets of traces require that the action is performed within a certain frame.

Case I.38: Action Sequence

$$\begin{aligned}&\sigma \vDash^{c,a}_{t_0} C'_3 \mathit{\ Seq\ } C''_3\\ &\qquad \text{iff } \exists j : \mathbb{N} \cdot \big(0 < j < \mathit{length}(\sigma) \wedge \sigma(..j) \vDash^{c,a}_{t_0} C'_3 \wedge \sigma(j..) \vDash^{c,a}_{t_0} C''_3\big)\end{aligned}$$

Event traces. Traces can be divided in two, such that first sub-trace respects C'_3 and the second sub-trace respects C''_3, given the agent a and constraints c.

Translation. The following automaton fragment, where each dotted line is replaced with the translations as marked.



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 I.39: Action Conjunction

$$\sigma \vDash_{t_0}^{c,a} C_3' \mathit{And} C_3'' \text{ iff } \sigma \vDash_{t_0}^{c,a} (C_3' \mathit{Seq} C_3'') \mathit{Or} (C_3'' \mathit{Seq} C_3')$$

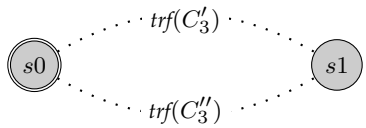
Argument. In both the trace semantics and the translation to UPPAAL, the *And* refinement is defined in terms of *Seq* and *Or*, and thus needs no special treatment here.

Case I.40: Action Choice

$$\sigma \vDash_{t_0}^{c,a} C_3' \mathit{Or} C_3'' \text{ iff either } \sigma \vDash_{t_0}^{c,a} C_3' \text{ or } \sigma \vDash_{t_0}^{c,a} C_3''$$

Event traces. Traces must respect either C_3' or C_3'', given the agent a and constraints c.

Translation. The following automaton fragment, where each dotted line replaced with the translations as marked.



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

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

Case I.41: Action Naming

$$\sigma \vDash_{t_0}^{c,a} \langle n, C_2 \rangle \text{ iff } \sigma \vDash_{t_0}^{c,a} C_2$$

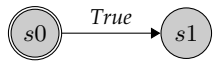
Argument. This case is simply handled recursively by considering the inner C_2 element.

Case I.42: Top

$$\sigma \vDash^{c}_{t_0} \top$$

Event traces. Any event trace respects *top*.

Translation. The following automaton fragment.



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

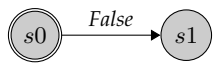
Argument. Both sets of traces are maximally inclusive.

Case I.43: Bottom

$$\sigma \nvDash^{c}_{t_0} \bot$$

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 I.44: Reference

$$\sigma \vDash^{c}_{t_0} \#n \text{ iff } \sigma \vDash^{c}_{t_0} \mathit{lookup}(n)$$

Argument. A reference is translated by looking up the clause in the contract with name n and making a copy of it, resulting in a clause with a structure matching one of the cases already seen earlier.

I.B Case study details

Support and Service Level Schedule for LeaseWeb USA, Inc. The case study below has been reproduced from: https://www.leaseweb.com/sites/default/files/US_ENG_B2B_v2014.1%20Support%20and%20Service%20Level%20Schedule_1.pdf

1.1 LeaseWeb shall provide an English-language customer support service. LeaseWeb will maintain support engineers actively on duty 24 hours per day, every day of the year.

1.2 LeaseWeb shall in no event be obliged to provide any support services to Customer's End Users.

1.3 Customer may initiate a request for Standard Support, Advanced Support or Remote Hands, or report a Service Disruption (a "Support Request") via the technical helpdesk via the Customer Portal, phone or e-mail. A Support Request must include the following information: (i) type of service, (ii) company name, (iii) name and number for immediate contact with the Customer, (iv) a clear, detailed and unambiguous description of Standard Support, Advanced Support or Remote Hands Services requested, and (v) a detailed description of the Service Disruption (if applicable). LeaseWeb may refuse a Support Request if it is not able to establish that the Support Request is made by the person authorised thereto in the Customer Portal.

1.4 The table below sets forth the Response Time (the "Response Time Target") for (a) any Service Disruptions that have been reported by Customer to LeaseWeb in accordance with Section 1.3 above, and (b) any request for Standard Support Service, Advanced Support Service or Remote Hands Service to be performed made in accordance with Section 1.3 above. The Response Time Target, depends (i) for Colocation Services, on the Remote Hands Package chosen by Customer, and (ii) for any other Services, on the SLA level that the Customer has chosen.

SLA level	**Remote hands**	**Response time target**
Basic	Basic	24 hours
Bronze	Bronze	4 hours
Silver	Silver	2 hours
Gold	Gold	1 hour
Platinum	Platinum	30 minutes

1.5 In the event LeaseWeb does not respond within the applicable Response Time Target, Customer shall be eligible to receive a Service Credit (the "Response Time Credit") for every one (1) hour in excess of the maximum Response Time Target equal to 2% of the Monthly Recurring SLA Charge or the Monthly Recurring Remote Hands Charge (as applicable) for the respective month for the Service or Equipment affected by the Service Disruption or for which Advanced Support Services/Remote Hands were requested (as applicable). If Customer does not pay a Monthly Recurring SLA Charge or Monthly Recurring Remote Hands Charge (as applicable), 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 on Customer's emergency number.

1.7 The maximum amount of Response Time Credits that a Customer may be eligible to in a particular month, shall be limited to 50% of the Monthly Recurring SLA Charge or the Monthly Recurring Remote Hands Charge (as applicable) for the respective month for the Customer's Service or Equipment affected by the Service Disruption or for which Advanced Support Services/Remote Hands were requested (as applicable).

Paper II

$\mathcal{SCL}$: A domain-specific language for normative texts with timing constraints

Runa Gulliksson and John J. Camilleri

Abstract. We are interested in the formal modelling and analysis of normative documents containing temporal restrictions. This paper presents a new language for this purpose, based on the deontic modalities of obligation, permission, and prohibition. It allows the specification of normative clauses over actions, which can be conditional on guards and timing constraints defined using absolute or relative discrete time. The language is compositional, where each feature is encoded as a separate operator. This allows for a straightforward operational semantics and a highly modular translation into timed automata. We demonstrate the use of the language by applying it to a case study and showing how this can be used for testing, simulation and verification of normative texts.

Contents

Note that Appendix II.A does not appear in the published version of this paper.

II.1 Introduction

It is more or less impossible today to use an application or service without first agreeing to a long terms and conditions document which you probably didn't read. We refer to these as *normative texts* (or *contracts*), that is, natural language documents prescribing the rights and obligations of different parties. Our goal is to be able to automatically analyse and query such documents, by combining natural language technologies with formal methods. The core of this approach is formalisation, i.e. building a formal model which represents a non-formal text. Well-known generic formalisms such as first-order logic or temporal logic would not provide the right level of abstraction for this domain-specific task. Instead, we design a custom language based on the *deontic modalities* of **obligation**, **permission** and **prohibition**, and containing only the operators that are relevant to our domain. This paper introduces such a language, which apart from these modalities also includes constructors for guards and temporal constraints.

The rest of the article is laid out as follows. Section II.2 introduce the syntax of our language, $\mathcal{SCL}$, together with its operational semantics. Section II.3 covers how contracts in this language are converted into Networks of Timed Automata (NTA). In Section II.4 we look at a small case study, showing how the language is used and the kinds of testing performed on the model. We then discuss some related work in this area (Section II.5) and end with a discussion of future work (Section II.6).

II.2 Language

We begin by introducing our domain-specific language (DSL) which we call *Simplified Contract Language*, or $\mathcal{SCL}$. It is "simple" in the sense that we have a separate constructor for each concept, such that guards and timing constraints are not defined as part of the main deontic operators. In addition, each constructor has a specific semantics and the idea is that complex constraints are constructed via composition.

This is in contrast to previous work (see Section II.5), where guards, timing constraints and reparations are all combined into monolithic constructors. The benefits for this at the modelling stage are not obvious, however it has a big payoff when defining our semantics (Section II.2.4) and in making our translation to Timed Automata more modular (Section II.3).

II.2.1 Syntax

$\mathcal{SCL}$ is an action-based language, describing what may and may not *be done* (as opposed to what *should be*). The core of the language is the atomic deontic operators for obligation *O*, permission *P* and prohibition *F*. Each of these is intended to describe the modality of an *agent* (e.g. *the student*) over a simple *act* (e.g. *submitting an assignment*). To simplify things, we use the term *action* to mean both agent and act together. In other words, given a set of agents $\mathcal{A}$ and a set of simple acts $\mathcal{X}$, the set of $\mathcal{SCL}$ actions Σ is defined as the Cartesian product $\Sigma = \mathcal{A} \times \mathcal{X}$. In addition to this set we also assume a set of integer variables $\mathcal{V}$ and a set of clause names $\mathcal{N}$ (where all above mentioned sets are disjoint).

We distinguish two kinds of temporal values in $\mathcal{SCL}$: those which are relative ($\mathbb{T}_r$), as in *5 days*, and those which are absolute ($\mathbb{T}_a$), as in *31st May 2014*. Again for simplicity, here we treat both relative and absolute temporal values as natural numbers. However these could just as easily be implemented as types representing real time units and calendar dates without any changes to the language.

The full syntax of $\mathcal{SCL}$ is shown in Figure II.1. We use the term *clause* (or *sub-clause*) to refer to anything of type *C*, while *contract* refers to a list of top-level clauses (type *Contract*). In brief, $\top$ is the trivially satisfied clause while $\bot$ is unsatisfiable and indicates irreparable violation. *O*, *P* and *F* are the basic deontic operators over actions described above. The declaration operator *D* allows variables to be updated using either literal values or other variables. A clause can be *Named* so that its status may be queried in guard expressions. Clauses can be refined into sub-clauses by conjunction *And*, choice *Or*, and sequence *Seq*. The reparation operator *Rep* specifies an alternative clause to be applied if the clause is violated.

The operators concerning timing constraints are as follows. *Wait* waits for a relative amount of time before enabling its clause, while *Within* ensures that the inner clause is satisfied within a given amount of time (failing with $\bot$ otherwise). *After* and *Before* are the absolute time versions of the previous two constructors. *In* and its counterpart *At* are similar to *Within* and *Before*, but they wait until their time constraint has expired before checking the status of their inner clause.

Guards over clauses are introduced with *When*, which will activate the inner clause when the guard condition is met (waiting forever otherwise). The expiring versions of *When* are *WhenWithin* and *WhenBefore*, for relative and absolute temporal values respectively.

Guards themselves can be seen as predicates over the current state as stored in the evaluation environment. Specifically, $\mathit{done}(a)$ checks whether the action a has been done and $\mathit{sat}(n)$ checks whether the named clause n has been satisfied. $\mathit{earlier}(\mathbb{T}_a)$ and $\mathit{later}(\mathbb{T}_a)$ query the current time. Variables can be compared with fixed values or other variables using $<$, $=$, and $>$. Guards can

$$
\begin{aligned}
\mathit{Contract} &:= [C] \\
C &:= \top \mid \bot \\
&\mid O\langle a\rangle \mid P\langle a\rangle \mid F\langle a\rangle \text{ where } a \in \Sigma \\
&\mid D\langle v, \mathit{Val}\rangle \text{ where } v \in \mathcal{V} \\
&\mid \mathit{Named}\langle n, C\rangle \text{ where } n \in \mathcal{N} \\
&\mid \mathit{And}\langle C, C\rangle \mid \mathit{Or}\langle C, C\rangle \mid \mathit{Seq}\langle C, C\rangle \mid \mathit{Rep}\langle C, C\rangle \\
&\mid \mathit{Wait}\langle \mathbb{T}_r, C\rangle \mid \mathit{After}\langle \mathbb{T}_a, C\rangle \\
&\mid \mathit{Within}\langle \mathbb{T}_r, C\rangle \mid \mathit{Before}\langle \mathbb{T}_a, C\rangle \\
&\mid \mathit{In}\langle \mathbb{T}_r, C\rangle \mid \mathit{At}\langle \mathbb{T}_a, C\rangle \\
&\mid \mathit{When}\langle G, C\rangle \\
&\mid \mathit{WhenWithin}\langle \mathbb{T}_r, G, C\rangle \\
&\mid \mathit{WhenBefore}\langle \mathbb{T}_a, G, C\rangle \\
G &:= \mathit{done}(a) \text{ where } a \in \Sigma \\
&\mid \mathit{sat}(n) \text{ where } n \in \mathcal{N} \\
&\mid \mathit{earlier}(\mathbb{T}_a) \mid \mathit{later}(\mathbb{T}_a) \\
&\mid \mathit{Val} < \mathit{Val} \mid \mathit{Val} = \mathit{Val} \mid \mathit{Val} > \mathit{Val} \\
&\mid \neg G \mid G \wedge G \mid G \vee G \\
\mathit{Val} &:= v \text{ where } v \in \mathcal{V} \\
&\mid i \text{ where } i \in \mathbb{Z}
\end{aligned}
$$

Figure II.1: $\mathcal{SCL}$ syntax.

be negated ($\neg$) and combined via conjunction ($\wedge$) or disjunction ($\vee$).

II.2.2 Example

As a running example we pick here a single clause from our larger case study (more details can be found in Section II.4). Let action submit stand for the student submitting a lab assignment. We can make this obligatory with $O\langle\mathsf{submit}\rangle$. To specify the submission deadline, we use the *At* constructor with a deadline of 11, giving $\mathit{At}\langle 11, O\langle\mathsf{submit}\rangle\rangle$. The submission should be followed by the a grader correcting it within 7 days of the deadline. Thus we combine *Seq* and *Within* to end up with $\mathit{Seq}\langle \mathit{At}\langle 11, O\langle\mathsf{submit}\rangle\rangle, \mathit{Within}\langle 7, O\langle\mathsf{accept}\rangle\rangle\rangle$. If the grader decides to reject the assignment, the student must resubmit before a second deadline and the grader will need to accept this new submission. This can be modelled as a reparation which applies when the first obligation to

accept the lab is violated. We also give a name to this entire clause, obtaining finally:

$$
\begin{aligned}
&\mathit{Named}\langle \mathsf{Lab}, \mathit{Seq}\langle \\
&\quad \mathit{At}\langle 11, O\langle \mathsf{submit} \rangle \rangle, \\
&\quad \mathit{Rep}\langle \\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 26, O\langle \mathsf{resubmit} \rangle \rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept} \rangle \rangle \rangle, \\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept} \rangle \rangle \rangle \rangle \rangle
\end{aligned}
$$

II.2.3 Implementation

$\mathcal{SCL}$ is implemented in Haskell [59] as an embedded domain-specific language (EDSL) [38]. This allows us to implement the language without the need to design a concrete syntax or build any compilation tools. Haskell makes a suitable host language because its algebraic data types allow for a clean and direct implementation of $\mathcal{SCL}$'s combinators, whilst its strong static type system can be leveraged to ensure that all constructed $\mathcal{SCL}$ terms are type-correct.

Another benefit of using an EDSL is that it allows the programmer to take advantage of the host language when working with contract models. For example, the conjunction operator *And* takes exactly two arguments, but we can easily build a conjunction of an arbitrarily long list of sub-clauses by folding:

```
foldr1 And [c1, c2, c3, c4] =
  And c1 (And c2 (And c3 c4))
```

Taking this idea further, entire sub-clauses can be encoded as parametrised Haskell functions, meaning that common clause patterns do not need to be written out explicitly each time by the programmer. This also helps with the readability of the source code. For example, we often want to specify a sequence of two sub-clauses c_1 and c_2, where if the first is not satisfied, then a reparation r is applied immediately without the second sub-clause being enabled. This behaviour can be encoded using the following pattern:

$$\mathit{Seq}\langle \mathit{Rep}\langle r, \mathit{Named}\langle n, c_1 \rangle \rangle, \mathit{When}\langle \mathit{GSat}(n), c_2 \rangle \rangle$$

We can encode this as a Haskell function `seqRep1` below:

```
seqRep1 :: C -> C -> C -> C
seqRep1 c1 c2 r =
  Seq (Rep r (Named n c1)) (When (GSat n) c2)
  where n = anonName c1
```

where `anonName` is a function that creates automatically generated names for internal use. We can then use this function to create a clause describing the publishing of course assignments. For each assignment, the description must be published by the teacher by a given date, after which the student has 5 days to submit their solution. If the teacher misses the publication deadline, then the student has the right to an automatic pass for that assignment. This behaviour is captured in the function `task`:

```
task :: Name -> AbsTime -> C
task n t =
  seqRep1
    (At t (O (Action ("publish "++n))))
    (Within 5 (O (Action ("submit "++n))))
    (D ("pass_"++n) (VInt 1))
```

Finally this can be applied to a list of three assignments *proposal*, *essay* and *review*, each with a different deadline:

```
foldr1 Seq $ map (uncurry task) [
  ("proposal",10), ("essay",22), ("review",26)
]
```

This produces the following clause:

```
Seq
  (Seq
    (Rep
      (D "pass_proposal" (VInt 1))
      (Named
        "8893"
        (At 10 (O (Action "publish proposal")))))
    (When
      (GSat "8893")
      (Within 5 (O (Action "submit proposal")))))
  (Seq
    (Seq
      (Rep
        (D "pass_essay" (VInt 1))
        (Named
          "5075"
          (At 22 (O (Action "publish essay")))))
      (When
        (GSat "5075")
        (Within 5 (O (Action "submit essay")))))
    (Seq
      (Rep
```

```
        (D "pass_review" (VInt 1))
        (Named
          "4783"
          (At 26 (O (Action "publish review")))))
      (When
        (GSat "4783")
        (Within 5 (O (Action "submit review"))))))
```

An existing contract can also be modified by writing a function which traverses its structure and makes certain changes, for example extending all deadlines by some amount. Filtering out clauses which meet a certain criteria, such as pertaining to a particular action, can also be done in a similar way.

II.2.4 Operational semantics

The semantics for $\mathcal{SCL}$ describe how a contract evolves as actions are performed and as time elapses. We begin by introducing the concept of a *trace*, which is a sequence of events ordered by the time stamp at which they occurred. An *event* may either be an *action* performed by an agent, or an update to a variable in the environment which we call an *observation*. For example, the student submitting the lab at time stamp 7 and the grader accepting it at time stamp 13 is represented by the trace:

$$[7 : \mathsf{submit}, 13 : \mathsf{accept}]$$

We are interested in the validity of this trace with respect to a given contract. Formally, we say that a contract is *satisfied* if all its top-level clauses have been reduced to $\top$ (accounting for *Named* clauses), and that a contract is *violated* if any of its top-level clauses have been reduced to $\bot$. These two concepts are **not** opposites, as a contract may be free from violations without being fully satisfied (e.g. if it contains obligations which still need to be fulfilled). We use the term *non-violated* to refer to this state. A trace is *valid* with respect to a given contract as long as it does not lead to a violation of that contract.

The operational semantics for $\mathcal{SCL}$ are defined as a residual function which given a contract and an event trace (and an initial environment) returns an updated contract and environment. We treat time as discrete and actions as instantaneous (they take no time to complete). An event trace is expanded into a sequence of *steps*, where a step is either: the doing of an action a (indicated $\xrightarrow{a}$), an update of a variable ($\xrightarrow{x=1}$), or a delay of one time unit ($\rightsquigarrow$). We use the arrow $\dashrightarrow$ to

$$Obl \frac{}{O\langle a\rangle \xrightarrow{x} \top} a = x \tag{II.1}$$

$$At_{Thru} \frac{C \dashrightarrow C'}{At\langle t, C\rangle \dashrightarrow At\langle t, C'\rangle} \Gamma[t_0] < t \tag{II.2}$$

$$At_{Top} \frac{At\langle t, C\rangle}{\top} \Gamma[t_0] \geq t, isTop(C) \tag{II.3}$$

$$Seq_{Thru} \frac{C_1 \dashrightarrow C_1'}{Seq\langle C_1, C_2\rangle \dashrightarrow Seq\langle C_1', C_2\rangle} \tag{II.4}$$

$$Seq_{Top} \frac{Seq\langle C_1, C_2\rangle}{C_2} isTop(C_1) \tag{II.5}$$

$$Within_{Del} \frac{C \rightsquigarrow C'}{Within\langle z, C\rangle \rightsquigarrow Within\langle z-1, C'\rangle} z \geq 1 \tag{II.6}$$

$$Within_{Top} \frac{Within\langle z, C\rangle}{\top} isTop(C) \tag{II.7}$$

$$Rep_{Top} \frac{Rep\langle C_r, C\rangle}{\top} isTop(C) \tag{II.8}$$

Figure II.2: Selection of $\mathcal{SCL}$ semantic rules. For the full set of rules, please refer to Appendix II.A.

mean a single step of any kind. The event trace above is thus expanded to the sequence of steps:

$$\Big[\underbrace{\rightsquigarrow, \ldots, \rightsquigarrow}_{7 \text{ times}}, \xrightarrow{\text{submit}}, \underbrace{\rightsquigarrow, \ldots, \rightsquigarrow}_{6 \text{ times}}, \xrightarrow{\text{accept}} \Big]$$

One could append an arbitrary or even an infinite number of delay steps to the end of this sequence, but for our purposes here the sequence is terminated by the last event from the trace.

Apart from a trace, the evaluation of a contract also requires an *environment* Γ which maps actions to time stamps ($\Sigma \mapsto \mathbb{T}_a$) for recording when they take place, names to their inner clauses ($\mathcal{N} \mapsto C$), and variables to their integer values ($\mathcal{V} \mapsto \mathbb{Z}$). It also contains a variable t_0 which indicates the current time in *ticks* (number of delay steps consumed). An action step sets the time stamp for the corresponding action to the current time ($\Gamma[a := t_0]$) while a delay step increments

the current time by one unit ($\Gamma[t_0 \mathrel{+}= 1]$). Guard predicates are evaluated within an environment as follows: $\mathit{done}(a)$ holds if action a has occurred ($\Gamma[a] > -1$); $\mathit{sat}(n)$ holds if name n is satisfied $\top$ ($\Gamma[n] = \top$); $\mathit{earlier}(t)$ holds if $\Gamma[t_0] < t$ and $\mathit{later}(t)$ holds if $\Gamma[t_0] > t$.

As an example, consider the clause given at the end of Section II.2.2 and the step sequence given above. The first 7 delay steps have no effect on the clause, other than updating the clock t_0. When consuming the action step submit, the obligation corresponding to it is eliminated by rule (II.1) in Figure II.2, leaving the following clause and environment:

$$\mathit{Named}\langle \mathsf{Lab}, \mathit{Seq}\langle \mathit{At}\langle 11, \top\rangle, \mathit{Rep}\langle \ldots\rangle\rangle\rangle$$
$$[\mathsf{submit} = 7,\ \mathsf{Lab} = \mathit{Seq}\langle \ldots\rangle,\ t_0 = 7]$$

After consuming 4 more delay steps, the *At* clause is replaced with $\top$ by rule (II.3), resulting in:

$$\mathit{Named}\langle \mathsf{Lab}, \mathit{Seq}\langle \top, \mathit{Rep}\langle \ldots\rangle\rangle\rangle$$
$$[\mathsf{submit} = 7,\ \mathsf{Lab} = \mathit{Seq}\langle \ldots\rangle,\ t_0 = 11]$$

The *Seq* operator can also be factored away by rule (II.5):

$$\begin{aligned}&\mathit{Named}\langle \mathsf{Lab}, \mathit{Rep}\langle\\ &\quad \mathit{Seq}\langle \mathit{Before}\langle 26, O\langle \mathsf{resubmit}\rangle\rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept}\rangle\rangle\rangle,\\ &\quad \mathit{Within}\langle 7, O\langle \mathsf{accept}\rangle\rangle\rangle\rangle\\ &[\mathsf{submit} = 7,\ \mathsf{Lab} = \mathit{Rep}\langle \ldots\rangle,\ t_0 = 11]\end{aligned}$$

Consuming the two remaining delay steps decrements the relative time value in the *Within* clause (rule (II.6)):

$$\mathit{Named}\langle \mathsf{Lab}, \mathit{Rep}\langle \ldots, \mathit{Within}\langle 5, O\langle \mathsf{accept}\rangle\rangle\rangle\rangle$$
$$[\mathsf{submit} = 7,\ \mathsf{Lab} = \mathit{Rep}\langle \ldots\rangle,\ t_0 = 13]$$

Finally, the step for the accept action is consumed and we have the following sequence of simplifications to the clause:

$$\begin{aligned}
&\mathit{Named}\langle \mathsf{Lab}, \mathit{Rep}\langle \ldots, \mathit{Within}\langle 5, \top\rangle\rangle\rangle && \text{by (II.1)}\\
&\mathit{Named}\langle \mathsf{Lab}, \mathit{Rep}\langle \ldots, \top\rangle\rangle && \text{by (II.7)}\\
&\mathit{Named}\langle \mathsf{Lab}, \top\rangle && \text{by (II.8)}
\end{aligned}$$

where the final state of the evaluation environment is:

$$[\mathsf{submit} = 7,\ \mathsf{accept} = 13,\ \mathsf{Lab} = \top,\ t_0 = 13]$$

These operational semantics have been implemented as a Haskell function which allows us to apply these rules automatically and programmatically determine the validity of a trace with respect to a contract.

II.3 Translation to timed automata

To enable property-based analysis on contract models, we define a translation from $\mathcal{SCL}$ into *networks of timed automata* (NTAs). A *timed automaton* (TA) [2] is a finite automaton extended with clock variables which increase their value as time elapses, all at the same rate. The model also includes clock constraints, allowing clocks to be used in guards on transitions and in invariants on locations, in order to restrict the behaviour of the automaton. Clocks can be reset to zero during the execution of a transition. An NTA is a set of TAs which are run in parallel, sharing the same set of clocks. The definition of NTA also includes a set of channels which allow synchronisation between independent automata.

UPPAAL [56] is a tool for the modelling, simulation and verification of real-time systems. The modelling language used in UPPAAL extends timed automata with a number of features, amongst them the concepts of *urgent* and *committed* locations which prevent time from elapsing when any process is in such a location. It also introduces the idea of *broadcast* channels, which allow one sender to synchronise with an arbitrary number of receivers. The query language of UPPAAL, which is used to define properties to be checked over a system of automata, is a subset of timed computation tree logic (TCTL) [11].

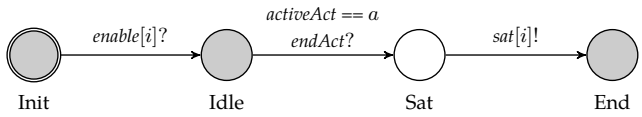


Figure II.3: Template for the obligation $O\langle \mathsf{a} \rangle$, where i is the index of the clause. White nodes indicate committed locations.

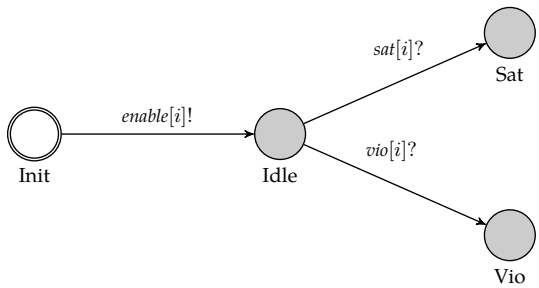


Figure II.4: A *Start* template, where i is the index of the clause being started.

II.3.1 Modularity

For each of the $\mathcal{SCL}$ constructors, we have designed a corresponding UPPAAL template that models its behaviour. Every template has a channel at the start that enables it and one or two response channels at the end in order to send either a satisfaction or a violation response. Translating a given contract into a UPPAAL system involves creating an instance of the corresponding templates for each clause and sub-clause. Template parameters are used to specify which channels each instance should synchronise on, thus linking them together. Figure II.3 shows the generic template for the obligation clause $O\langle \mathsf{a} \rangle$. After being enabled, the template waits in the Idle location while listening for a synchronisation corresponding to the action a. The template then immediately signals that it has been satisfied to its parent.

To enable the top-level clauses in a contract the *Start* template is used (Figure II.4). As the first location is committed, the first thing that happens (before time passes) is that the clause is enabled. The template then waits to receive a signal on one of the response channels.

As an example, the contract $\left[And\langle O\langle \mathsf{a} \rangle, O\langle \mathsf{b} \rangle\rangle\right]$ will require a total of four template instantiations (processes) in the corresponding UPPAAL system:

```
Start(0), And(0,1,2), O(a,1), O(b,2);
```

The *Start* process will send a synchronisation signal on the enabling channel with index 0, which is the index that the *And* process is listening to. This in turn enables processes with indexes 1

and 2, corresponding to $O\langle a \rangle$ and $O\langle b \rangle$ respectively, where `a` and `b` are constants referring to the indexes for those actions.

The only exception to this design are the templates involving guards, i.e. *When*, *WhenWithin* and *WhenBefore*. A new template must be built for every occurrence of these clauses in the contract, as UPPAAL does not make it possible to use guards as parameters to a template.

II.3.2 Step generation

$\mathcal{SCL}$ uses a discrete model of time where actions occur between clock ticks, and all parts of the contract are updated in lock-step when a time delay step occurs. This behaviour must be simulated in UPPAAL in a way that ensures that all processes in the system are updated and reach their correct waiting locations before the next delay or action step occurs. Depending on whether we want to simulate a particular event trace or not, this behaviour is modelled in one of two ways.

With a trace

An event trace in $\mathcal{SCL}$ is translated into a UPPAAL template along with the rest of the contract. In this case the order and timing of every step is predetermined, resulting in a long sequential template as shown in Figure II.5. Each stage in the template corresponds to either a time step, an action or an observation. The local clock `t` is used as a location invariant in order to trigger delay steps at the correct time. The global integer variable `ticks` is incremented during every time step to reflect the current discrete time.

Arbitrary order of actions

Without an available trace, we model the possibility of actions occurring and variables being updated at any time and in any order. This is done with two additional templates. The *Ticker* template (Figure II.6) handles the simulation of time, initiating a new delay step at regular intervals determined by the clock `t`. The global clock `t0` keeps track of the system time (it is never reset). The *Doer* template (Figure II.7) takes care of action and observation steps, generating any action in the contract or changing the value of any variable (by incrementing or decrementing it). This may occur any number of times during the same time unit. Arguments to the translation function control various aspects of the generation of these steps, such as the value by which variables are in/decremented, limits for how high or low variables can be set, an upper bound on the time, and whether or not an action can occur more than once.

Delay and action steps are encoded in UPPAAL as channel synchronisations between processes. These synchronisations are instantaneous, and if a process is not listening when the signal

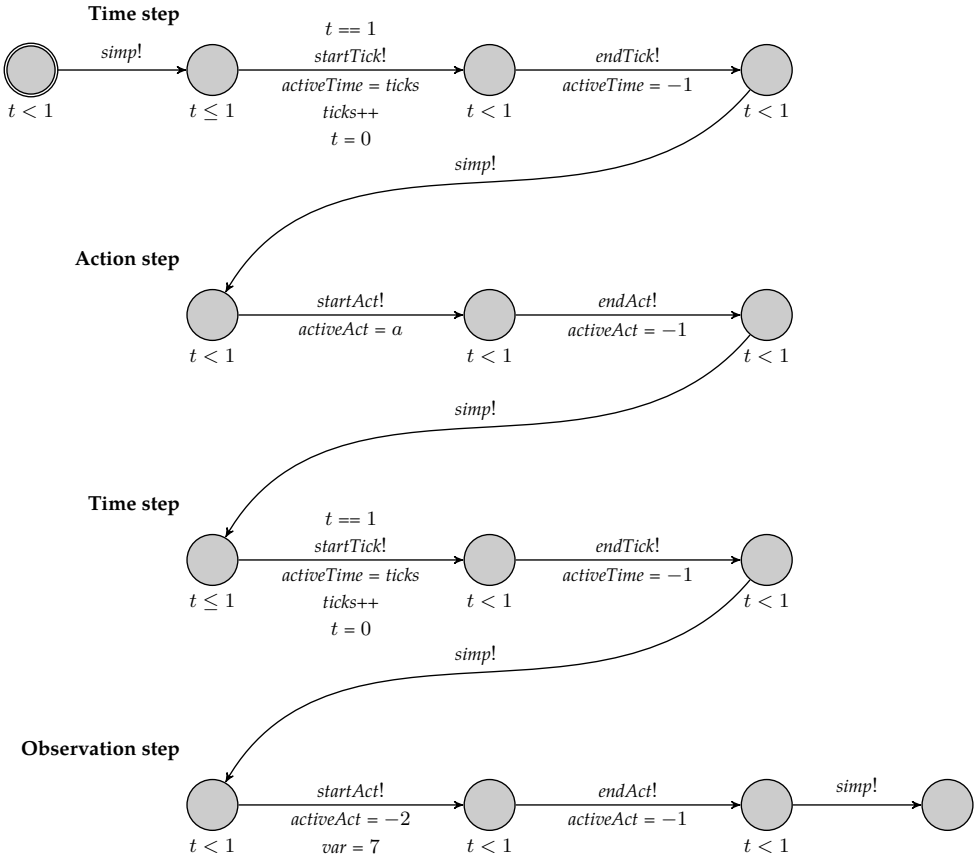


Figure II.5: Template for the trace $[1 : a,\ 2 : v = 7]$ (action a taking place at time 1 and variable v being updated to 7 at time 2).

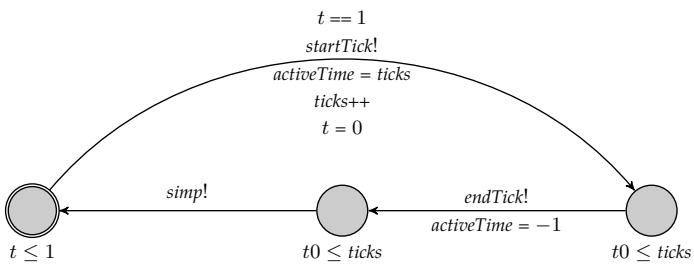


Figure II.6: *Ticker* template, which triggers delay steps in the system.

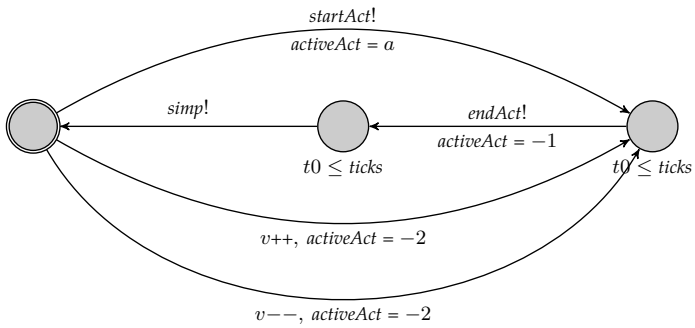


Figure II.7: *Doer* template, containing transitions for action a and variable v. Further actions can be added to the same template by replicating the top transition and replacing the relevant action index in place of a.

is sent then it will miss it. This turned out to be quite problematic for representing the behaviour of $\mathcal{SCL}$. Our solution to this involves the `ticks` variable together with variables indicating the active action and time steps (`activeAct` and `activeTime` respectively). The variable `activeAct` is -1 when there is no action step taking place. During an action step, it takes the value of the index for the action taking place. For observation steps a value of -2 is used, since variables do not have indexes. Similarly, `activeTime` is also set to -1 when inactive and to the current time when active.

Each step begins with a synchronisation on a *start* channel and ends with one on an *end* channel. These are listened for on edges in other templates in order to progress between locations at the right time. Since these channels are *broadcast*, they can be signalled on without requiring any another process to be listening.

In between every time or action step, a synchronisation is sent on a simplification channel `simp`. This is used to ensure that all processes progress to the location they need to be in before the next active state begins.

II.3.3 Automatic testing for correctness

In order to test the correctness of our translation to timed automata, we compare the behaviour of an $\mathcal{SCL}$ contract with that of its generated automaton in UPPAAL. Specifically, we want to ensure that both representations encode the same notion of contract satisfaction. In terms of the $\mathcal{SCL}$ semantics, a trace satisfies a contract when evaluation produces a contract with $\top$ in all the top-level clauses. In the UPPAAL representation, contract satisfaction can be checked for by verifying

that a satisfaction property of the following form holds:

$$E\Diamond\ C.status = SAT$$

where C is a process representing the top-level clause in the contract, and SAT is a constant indicating the satisfaction status of a process. As the event trace itself is also translated as an automaton, such a property will be satisfied if, and only if, the given trace leads to a state of contract satisfaction.

To thoroughly test our translation we repeat this process for many different contracts and traces which are generated randomly. To do this we use QuickCheck [24], a tool for testing Haskell programs automatically. By providing a specification of a program in the form of properties which its functions should satisfy, QuickCheck then tests that these properties hold in a large number of randomly generated cases. QuickCheck provides combinators for defining properties, building test data generators, shrinking counter-examples, and observing the distribution of test data.

Using QuickCheck, we define a property which takes an arbitrary contract and trace, converts them into UPPAAL using our translation and runs the verifier as an external program with the kind of query shown above. The result of this is then compared with the result of evaluating the same contract and trace using the pure $\mathcal{SCL}$ semantics, ensuring that the two match. This has been carried out on tens of thousands of random test cases without any failing examples.

II.4 Case study

To demonstrate the use of $\mathcal{SCL}$, we apply it to a small case study concerning the rules governing the running of a university course. A textual description of these rules can be found in Figure II.8. The corresponding $\mathcal{SCL}$ model for this case study is shown in Figure II.9. It is a contract with 6 top-level clauses, where each has been named for convenience. This model was built manually by the authors. While automating this modelling process is of great interest to us, it is beyond the scope of the present work. For more about this, see [16]. The rest of this section describes the various ways in which this contract model can be tested and verified.

Students need to register for the course before the registration deadline, 1 week after the course has started.
Students must sign up for the exam before before the deadline on day 45.
The first deadline for lab assignment 1 is on day 10. If the assignment is not accepted, the student will have until the final deadline on day 25 to re-submit it.
The first deadline for lab assignment 2 is on day 30. The final deadline is on day 48.
Graders have 7 days from a submission deadline to correct an assignment.
The exam will be held on day 60.
The examiner should correct the exams within 3 weeks.
To pass the course, a student needs to pass all assignments and get a passing grade on the exam. The grade needs to be registered before day 90.

Figure II.8: Textual description of the course case study. Integral numbers are used as absolute time stamps. The course is assumed to start on day 0.

$$
\begin{aligned}
&\big[\mathit{Named}\langle \mathsf{InCourse}, \mathit{Before}\langle 8, P\langle \mathsf{regCourse}\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{RegisteredExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Before}\langle 45, P\langle \mathsf{regExam}\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{Lab1}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle\\
&\quad \mathit{At}\langle 11, O\langle \mathsf{submit1}\rangle\rangle,\\
&\quad \mathit{Rep}\langle\\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 26, O\langle \mathsf{resubmit1}\rangle\rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept1}\rangle\rangle\rangle,\\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept1}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{Lab2}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle\\
&\quad \mathit{At}\langle 31, O\langle \mathsf{submit2}\rangle\rangle,\\
&\quad \mathit{Rep}\langle\\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 49, O\langle \mathsf{resubmit2}\rangle\rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept2}\rangle\rangle\rangle,\\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept2}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{RegisteredExam}),\\
&\quad \mathit{After}\langle 60, \mathit{Seq}\langle\\
&\qquad \mathit{Within}\langle 1, P\langle \mathsf{takeExam}\rangle\rangle,\\
&\qquad \mathit{Within}\langle 21, O\langle \mathsf{passExam}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassCourse}, \mathit{Before}\langle 90,\\
&\quad \mathit{When}\langle \mathsf{sat}(\mathsf{Lab1}) \wedge \mathsf{sat}(\mathsf{Lab2}) \wedge \mathsf{sat}(\mathsf{PassExam}), \top\rangle\rangle\rangle\big]
\end{aligned}
$$

Figure II.9: $\mathcal{SCL}$ contract for the course case study.

II.4.1 Trace-based unit testing

The first thing we can do with our model is to come up with various concrete event traces and evaluate the contract against them. This can be done either by (i) using the pure implementation of the $\mathcal{SCL}$ semantics (Section II.2.4), or (ii) by translating the model and trace into UPPAAL (Section II.3). If all we are concerned with is contract satisfaction, then it should make no difference which method is used, as both representations behave equivalently (as argued in Section II.3.3).

Using the operational semantics

The following is an example of a valid trace, where all obliged actions are performed within their respective time constraints:

$$\begin{aligned}&[7 : \mathsf{regCourse}, 8 : \mathsf{regExam}, 10 : \mathsf{submit1}, 17 : \mathsf{accept1},\\ &\ 30 : \mathsf{submit2}, 48 : \mathsf{resubmit2}, 54 : \mathsf{accept2},\\ &\ 60 : \mathsf{takeExam}, 80 : \mathsf{passExam}]\end{aligned}$$

When evaluated together with our contract model, this gives the fully satisfied contract below:

$$\begin{aligned}&\big[\mathit{Named}\langle \mathsf{InCourse}, \top\rangle, \mathit{Named}\langle \mathsf{Lab1}, \top\rangle, \mathit{Named}\langle \mathsf{Lab2}, \top\rangle,\\ &\ \mathit{Named}\langle \mathsf{RegisteredExam}, \top\rangle, \mathit{Named}\langle \mathsf{PassExam}, \top\rangle,\\ &\ \mathit{Named}\langle \mathsf{PassCourse}, \top\rangle\big]\end{aligned}$$

As a negative example, consider the same trace as above but with actions related to lab 2 missing:

$$\begin{aligned}&[7 : \mathsf{regCourse}, 8 : \mathsf{regExam}, 10 : \mathsf{submit1}, 17 : \mathsf{accept1},\\ &\ 60 : \mathsf{takeExam}, 80 : \mathsf{passExam}]\end{aligned}$$

This time, evaluation using the operational semantics gives the non-satisfied contract below. Note in particular that the clause Lab2 has evaluated to the unsatisfiable clause $\bot$:

$$\begin{aligned}&\big[\mathit{Named}\langle \mathsf{InCourse}, \top\rangle, \mathit{Named}\langle \mathsf{Lab1}, \top\rangle, \mathit{Named}\langle \mathsf{Lab2}, \bot\rangle,\\ &\ \mathit{Named}\langle \mathsf{RegisteredExam}, \top\rangle, \mathit{Named}\langle \mathsf{PassExam}, \top\rangle,\\ &\ \mathit{Named}\langle \mathsf{PassCourse}, \mathit{Before}\langle 90,\\ &\quad \mathit{When}\langle \mathsf{sat}(\mathsf{Lab1}) \wedge \mathsf{sat}(\mathsf{Lab2}) \wedge \mathsf{sat}(\mathsf{PassExam}), \top\rangle\rangle\rangle\big]\end{aligned}$$

Using the UPPAAL translation

With our timed automata representation of the case study, contract satisfaction can be checked by verifying the following query:

$$E\Diamond\ \mathsf{PassCourse}.\mathit{status} = \mathit{SAT}$$

However as UPPAAL allows us to verify general reachability properties against the system, these can be used to query not just the final status of a contract, but also the intermediate states of its clauses.

For example, given the valid trace introduced above, we can verify that the Lab2 clause only becomes satisfied when the second lab is accepted. This is done by verifying the query:

$$A\Box\ (\mathsf{Lab2}.\mathit{status} = \mathit{SAT}) \implies \mathit{isDone}(\mathsf{accept2})$$

where $\mathit{isDone}(\cdot)$ is a helper function defined in our UPPAAL system which returns a Boolean indicating whether the provided action has occurred or not. Alternatively, this query could also be expressed as a reachability property, trying to find if the clause can become satisfied before the time stamp associated with the accept2 action:

$$E\Diamond\ (\mathsf{Lab2}.\mathit{status} = \mathit{SAT}) \wedge (\mathit{ticks} < 54)$$

This query gives a result of unsatisfied. Verification of this kind, where a concrete trace is translated into an automaton together with the main contract, only takes a matter of milliseconds in UPPAAL.

II.4.2 Trace-based random testing

Apart from working with individual test cases as in the previous section, we can also use QuickCheck to randomly generate any number of traces for us. By supplying the random generation function with a set of constraints which the traces should meet, we can effectively test a whole class of traces which all meet the same criteria. These *trace constraints* describe the presence of actions within a trace, timing constraints over them, and their relative order. They are implemented as Haskell functions along with the rest of our framework.

```
allOf
  [ actionSeq [ regCourse, submit1, accept1 ]
  , actionAt submit1 (between 2 11)
  , negate (hasAction resubmit1)
  ]
```

This example is a conjunction of three sub-constraints, which state that:

(i) the trace contains the actions regCourse, submit1 and accept1 in that specific order;

(ii) action submit1 occurs between time stamps 2 and 11; and

(iii) the trace should not contain action resubmit1 at all.

Given these trace constraints, we use QuickCheck to randomly generate traces and test them against our contract model. As in the previous section, this could either mean evaluation using the pure $\mathcal{SCL}$ semantics, where we would also provide some criteria for validating the resulting contract and environment; or, we can translate the contract and trace pair to UPPAAL from within the QuickCheck property and verify a temporal query against the translated system.

This method is useful when we are not concerned with specific traces, but more generally with a class of traces which share some characteristics. As this is ultimately still testing and not verification, it is mainly suitable for uncovering counter-examples. An advantage of using QuickCheck is that when a failing example is found, it will be reduced to a minimal version of itself through shrinking. For example, if we use the trace constraints given above together with a property stating that the Lab1 clause is satisfied, then the following counter-example is found:

$$[5 : \mathsf{regCourse}, 7 : \mathsf{submit1}, 9 : \mathsf{accept1}, 77 : \mathsf{accept2},$$
$$101 : \mathsf{submit2}, 162 : \mathsf{resubmit2}, 185 : \mathsf{passExam}]$$

which is reduced to the minimal trace:

$$[0 : \mathsf{regCourse}, 2 : \mathsf{submit1}, 2 : \mathsf{accept1}]$$

The reason why this trace violates the contract is because the acceptance of lab assignments should come *after* the deadline (which for lab 1 is at time stamp 10). Note how the events after the accept1 action are in fact irrelevant and thus automatically removed in the minimal trace.

While the evaluation/verification time with this method should be no different than in the previous section, the time required for generating traces which match the supplied criteria may be an issue. This depends both on the number of constraints provided, as well as the implementation of the trace generation function.

II.4.3 Verification with temporal properties

Up until this point we have only considered using *concrete* traces when testing the behaviour of our contract. In order to verify properties which hold over *all possible event sequences*, we use the timed automaton representation of the contract together with the *Ticker* and *Doer* templates (as discussed in Section II.3.2). In this case, using the $\mathcal{SCL}$ operational semantics is no longer an option as they are only defined over concrete traces. As in the previous sections, verification of properties is done using the UPPAAL tool and query language.

As an example, we can check to see if it's possible to pass the course if missing the submission deadline for lab 2:

$$E\Diamond\ done[\mathsf{submit2}] > 30 \wedge \mathsf{PassCourse}.status = SAT$$

where *done* is an array containing the time stamps of performed actions (or -1 if the action is not performed). This query is not satisfied in UPPAAL, as we would expect. This could also be turned around to a query which verifies that if lab 2 is submitted before the deadline, then it should be possible to pass the lab:

$$E\Diamond\ done[\mathsf{submit2}] \leq 30 \wedge \mathsf{Lab2}.status \neq VIO$$

Furthermore, we can re-formulate the example from Section II.4.2 entirely as a query in UPPAAL like so:

$$\begin{aligned} A\Box\ \big(&isDone(\mathsf{regCourse}) \\ &\wedge done[\mathsf{submit1}] \geq done[\mathsf{regCourse}] \\ &\wedge done[\mathsf{accept1}] \geq done[\mathsf{submit1}] \\ &\wedge done[\mathsf{submit1}] \geq 2 \wedge done[\mathsf{submit1}] < 11 \\ &\wedge \neg isDone(\mathsf{resubmit1})\big) \implies \mathsf{Lab1}.status = SAT \end{aligned}$$

where the left-hand side of the implication models the trace constraints and the right-hand side is the property on the resulting contract state. Attempting to verify this query in UPPAAL gives a negative result, with a symbolic trace as a counter example corresponding to the event trace:

$$[2 : \mathsf{regCourse}, 2 : \mathsf{submit1}, 2 : \mathsf{accept1}]$$

Performance and compression

Though powerful, the limitation with this technique is that the time and memory required for verification may be prohibitively great. This is typically the case with safety properties which require the entire search space to be covered. This problem is of course well-known in model checking, and theoretically unavoidable. However we know empirically that small changes to a given system of automata, such as reducing the number of ticks between events, can have a significant effect on the verification time without altering the overall behaviour of the contract.

Thus, to mitigate performance issues, we propose *compressing* a contract model such that unnecessary gaps between deadlines (which translate into potentially many step transitions) are removed. This requires an algorithm to traverse the contract and pick up all its significant time stamps, which can then be used to re-scale the deadlines in the original contract to smaller values without otherwise altering its behaviour. For example, the Lab1 clause would be compressed into:

$$
\begin{aligned}
&\mathit{Named}\langle \mathsf{Lab1}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle \\
&\quad \mathit{At}\langle 2, O\langle \mathsf{submit1} \rangle\rangle, \\
&\quad \mathit{Rep}\langle \\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 4, O\langle \mathsf{resubmit1} \rangle\rangle, \mathit{Within}\langle 2, O\langle \mathsf{accept1} \rangle\rangle\rangle, \\
&\qquad \mathit{Within}\langle 2, O\langle \mathsf{accept1} \rangle\rangle\rangle\rangle\rangle\rangle
\end{aligned}
$$

Alternatively, rather than changing the values of the time stamps themselves, the translated automata could be modified to skip over multiple tick steps in one go for stretches where no other significant events occur. This option has the advantage that queries including specific temporal values will not need to be adjusted.

II.5 Related work

The $\mathcal{SCL}$ language is inspired by *C-O Diagrams*, introduced by Martínez et al. [60]. The idea of translation into NTA for analysis also comes from this work [28]. Our language is compositional, where each operator serves a single purpose, and thus is structurally quite different from the *C-O Diagram* language. However both languages cover very much the same concepts, and conversion from a *C-O Diagram* into an $\mathcal{SCL}$ term would be easy to do automatically (with the exception of clauses involving repetition).

Our ultimate goal is to produce a usable high-level system for end-to-end analysis of normative contracts. The present work continues on that of Camilleri [16], which describes the other

components and considerations such a system requires. These include not only the user interface and natural language processing aspects of modelling, but also a user query language which can abstract away from the different analysis methods discussed here.

The AnaCon [4] framework for contract analysis, based on the contract logic $\mathcal{CL}$ [78], has a similar goal but more limited scope than the current work. In particular, their underlying logical formalism contains no direct temporal notions other than sequencing, and the only kind of analysis possible is the detection of normative conflicts using the CLAN tool [32].

Pace and Schapachnik [71] introduce the *Contract Automata* formalism 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, which are key to our work.

Marjanovic and Milosevic [58] also defend a deontic approach for modelling of contracts. They pay special attention to temporal aspects, distinguishing between three different kinds of time: absolute, relative and repetitive. They also introduce visualisation concepts such as *role windows* and *time maps* and describe how they could be used as decision support tools during contract negotiation. Their ideas however do not seem to have been implemented as any usable system.

Wyner [90] 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. The tool is designed as an abstract, flexible framework in which alternative definitions of the deontic concepts can be expressed and exercised. However its high level of abstraction and lack of temporal operators make it limited in its application to processing concrete contracts. In particular, the work is focused on logic design issues and avoiding deontic paradoxes, and there is no treatment of query-based analysis as in our work.

There is a considerable body of work in the representation of contracts as knowledge bases or *ontologies*. The *LegalRuleML* project [6] is one of the largest efforts in this area, providing a rule interchange format for the legal domain, enabling modelling and reasoning that lets users evaluate and compare legal arguments. The format has a temporal model which supports the evolution of legal texts over time, such that their legal reasoner will dynamically apply the version of a document that was applicable at the time of a particular event.

Peyton Jones and Eber [73] introduce a functional combinator language for working with complex financial contracts — the kind which are traded in derivative markets — which they also implement as a DSL embedded in Haskell. These kinds of contracts are somewhat different from the normative documents 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. The compositional style of $\mathcal{SCL}$ is however similar to their language. For more recent work continuing this along these lines, see [10].

II.6 Conclusion

In this paper we have presented the language $\mathcal{SCL}$, an embedded DSL in Haskell for modelling normative texts with timing constraints. The language has an operational semantics which given a contract and event trace, returns a new residual contract. We also describe a method for translating from $\mathcal{SCL}$ models into Uppaal networks of timed automata, which we have implemented fully and tested rigorously with respect to the operational semantics using the random testing library QuickCheck. We then consider a small case study, showing how a text describing the running of a university course can be modelled in $\mathcal{SCL}$, and the various ways in which this can be tested, simulated and verified.

In this work we have used simple integers as time values. These however can be easily replaceable with calendar dates or clock times without any changes to the $\mathcal{SCL}$ language itself, but simply the implementation of the $\mathbb{T}_a$ and $\mathbb{T}_r$ types. The translation to Uppaal will also need to encode these accordingly.

Limitations

$\mathcal{SCL}$ can be seen as a *combinator library*, where complex contracts are built by stacking simple well-defined constructors together. While this makes the semantics and translation easier to define, it can make modelling normative texts from natural language less straightforward. In addition, terms in $\mathcal{SCL}$ can quickly become quite large and unwieldy, making them hard to debug or modify. Thus, we see $\mathcal{SCL}$ as more of an *assembly language* for contract analysis, which other higher-level languages or representations could be compiled into for further processing.

The concepts of obligation O and permission P do not, at first, seem to be properly differentiated in $\mathcal{SCL}$. Indeed, semantically they behave in the same way. The reason for having them as different operators in the language is that distinguishing between them can be useful when querying a contract: an unfulfilled obligation is more serious than an unfulfilled permission.

Another potentially misleading behaviour of $\mathcal{SCL}$ is that prohibition F is only persistent until violated. One might expect that stealing, for example, is always prohibited. However if we

consider the following clause and trace:

$$Rep\langle O\langle \mathsf{jail} \rangle, F\langle \mathsf{steal} \rangle \rangle$$
$$[1 : \mathsf{steal}, 2 : \mathsf{jail}, 3 : \mathsf{steal}]$$

we perhaps surprisingly discover that this trace containing two thefts is in fact a valid one, because the prohibition to steal does not get re-activated after it is repaired.

Another limitation of $\mathcal{SCL}$ is that the language does not support the concept of repetition, which would be useful for modelling recurring contracts, e.g. paying rent every month. The treatment of actions as instantaneous — that is, taking zero time to complete — may also be a limiting feature.

Unfortunately, full verification on our translated NTA when no trace is given can require more computing time and resources than is reasonably possible. While we try to mitigate this as much as possible by providing testing-based alternatives and the concept of contract compression, this still remains a significant issue.

Source code

The source code for all the work described in this paper, including the operational semantics, translation to UPPAAL, and the QuickCheck-based tests, is available under a GPL license at the following link: http://remu.grammaticalframework.org/contracts/time2016/.

II.A Operational semantics for $\mathcal{SCL}$

This appendix covers the full operational semantics for $\mathcal{SCL}$, given as a set of rules describing the evaluation of contract clauses with respect to event traces. The syntax of the language can be found in Figure II.1 (page 75).

II.A.1 Preliminaries

Traces and steps

An *event trace* is converted into a sequence of steps, where a step is either:

(a) an action: $\xrightarrow{a}$
(b) an observation (variable assignment): $\xrightarrow{v=i}$
(c) a delay of 1 time unit: $\rightsquigarrow$

We use the arrow $\dot{\rightarrow}$ to indicate either an action or an observation step, and the arrow $\dashrightarrow$ to indicate a step of *any* kind.

Environment

The shared evaluation environment Γ consists of the following:

1. a map from actions to time stamps ($\Sigma \mapsto \mathbb{T}_a$), indicating the time at which each action was performed (where -1 indicates a non-performed action)
2. a map from variables to integers ($\mathcal{V} \mapsto \mathbb{Z}$), indicating the current value of each variable
3. a map from names to clauses ($\mathcal{N} \mapsto C$), allowing the lookup of clauses by name
4. a time stamp variable t_0 for representing the current time in ticks

Values are projected from environment using the syntax $\Gamma[x]$. Conditions on the state of the environment can occur as side conditions in the semantic rules. Rules may also specify explicit updates to the environment, indicated using the syntax $\Gamma[x := y]$ where y is either a literal value or another variable in the environment. The following environment updates are implicit:

(i) an action step $\xrightarrow{a}$ updates the value of variable a to the current time stamp: $\Gamma[a := t_0]$
(ii) an observation step $\xrightarrow{v=i}$ updates the variable v to the specified value: $\Gamma[v := i]$
(iii) a delay step $\rightsquigarrow$ increments the current time by one tick: $\Gamma[t_0 \mathrel{+}= 1]$

Guards

Guards are defined as predicates over the environment. A guard G either holds in a given environment ($\Gamma \vdash G$), or it does not ($\Gamma \nvdash G$).

done$(\cdot)$ indicates whether a given action has occurred:

$$\mathit{done}(a) := \Gamma[a] > -1$$

sat$(\cdot)$ queries whether a named clause is satisfied:

$$\mathit{sat}(n) := \begin{cases} \mathit{sat}(n') & \text{if } \Gamma[n] = \mathit{Named}\langle n', C\rangle \\ \Gamma[n] = \top & \text{otherwise} \end{cases}$$

earlier$(\cdot)$ and *later*$(\cdot)$ compare a given time stamp with the current time:

$$\mathit{earlier}(t) := \Gamma[t_0] < t$$
$$\mathit{later}(t) := \Gamma[t_0] > t$$

The comparison operators over values $(<, =, >)$ and boolean operators over guards $(\neg, \wedge, \vee)$ behave as expected.

Side conditions

The following predicates over clauses are used as side conditions in the semantic rules below:

$$\mathit{isTop}(C) := \begin{cases} \mathit{isTop}(C') & \text{if } C = \mathit{Named}\langle n, C'\rangle \\ C = \top & \text{otherwise} \end{cases}$$

$$\mathit{isBot}(C) := \begin{cases} \mathit{isBot}(C') & \text{if } C = \mathit{Named}\langle n, C'\rangle \\ C = \bot & \text{otherwise} \end{cases}$$

$$\mathit{notTop}(C) := \neg \mathit{isTop}(C)$$
$$\mathit{notBot}(C) := \neg \mathit{isBot}(C)$$

II.A.2 Case analysis

Contract $[C]$

All clauses in a contract are evaluated together at each step, using a shared environment. The actual order of evaluation should respect the order of the clause list $[C]$.

Top $\top$

Top represents a satisfied clause.

Bottom $\bot$

Bottom represents a violated clause.

Obligation $O\langle a \rangle$ where $a \in \Sigma$

An obligation is satisfied when its action is performed.

$$Obl \frac{}{O\langle a \rangle \xrightarrow{x} \top} a = x$$

Permission $P\langle a \rangle$ where $a \in \Sigma$

A permission is satisfied when its action is performed.[1]

$$Per \frac{}{P\langle a \rangle \xrightarrow{x} \top} a = x$$

Prohibition $F\langle a \rangle$ where $a \in \Sigma$

A prohibition is violated when its action is performed.

$$For \frac{}{F\langle a \rangle \xrightarrow{x} \bot} a = x$$

Declaration $D\langle v, Val \rangle$ where $v \in \mathcal{V}$

Assign a value to a variable, either from another variable ($Decl_{Var}$) or as a literal integer ($Decl_{Int}$). These clauses are evaluated immediately (no step is consumed).

$$Decl_{Var} \frac{D\langle v, u \rangle}{\top} u \in \Gamma, \Gamma[v := u] \qquad Decl_{Int} \frac{D\langle v, i \rangle}{\top} \Gamma[v := i]$$

[1] Although semantically identical to obligation, a separate operator exists for permission because the two should be distinguishable at the contract level. This is relevant when querying whether a contract contains any clauses which are still *pending*. An obligation clause which has not been satisfied is considered to be pending, which means that a contract containing this clause cannot be considered satisfied either. On the other hand, a permission for an action which has not been performed is not considered pending, and a contract containing such a permission could still be considered satisfied.

Clause naming $\mathit{Named}\langle n, C\rangle$ where $n \in \mathcal{N}$

Named clauses are copied to the environment at each evaluation step, so that the status of that clause can be queried by a guard (in particular, the $\mathit{sat}(n)$ predicate).

$$\mathit{Named}\ \frac{C \dashrightarrow C'}{\mathit{Named}\langle n, C\rangle \dashrightarrow \mathit{Named}\langle n, C'\rangle}\ \Gamma[n := C']$$

Conjunction $\mathit{And}\langle C, C\rangle$

Both sub-clauses must be satisfied, in any order.

$$\mathit{And}_{\mathit{Thru}}\ \frac{C_1 \dashrightarrow C_1' \qquad C_2 \dashrightarrow C_2'}{\mathit{And}\langle C_1, C_2\rangle \dashrightarrow \mathit{And}\langle C_1', C_2'\rangle}$$

$$\mathit{And}_{\mathit{Top}}\ \frac{\mathit{And}\langle C_1, C_2\rangle}{\top}\ \mathit{isTop}(C_1) \wedge \mathit{isTop}(C_2) \qquad \mathit{And}_{\mathit{Bot}}\ \frac{\mathit{And}\langle C_1, C_2\rangle}{\bot}\ \mathit{isBot}(C_1) \vee \mathit{isBot}(C_2)$$

Choice $\mathit{Or}\langle C, C\rangle$

Either sub-clause can be satisfied.

$$\mathit{Or}_{\mathit{Thru}}\ \frac{C_1 \dashrightarrow C_1' \qquad C_2 \dashrightarrow C_2'}{\mathit{Or}\langle C_1, C_2\rangle \dashrightarrow \mathit{Or}\langle C_1', C_2'\rangle}$$

$$\mathit{Or}_{\mathit{Bot}}\ \frac{\mathit{Or}\langle C_1, C_2\rangle}{\bot}\ \mathit{isBot}(C_1) \wedge \mathit{isBot}(C_2) \qquad \mathit{Or}_{\mathit{Top}}\ \frac{\mathit{Or}\langle C_1, C_2\rangle}{\top}\ \mathit{isTop}(C_1) \vee \mathit{isTop}(C_2)$$

Sequence $\mathit{Seq}\langle C, C\rangle$

Both sub-clauses must be satisfied, in order.

$$\mathit{Seq}_{\mathit{Thru}}\ \frac{C_1 \dashrightarrow C_1'}{\mathit{Seq}\langle C_1, C_2\rangle \dashrightarrow \mathit{Seq}\langle C_1', C_2\rangle}$$

$$\mathit{Seq}_{\mathit{Top}}\ \frac{\mathit{Seq}\langle C_1, C_2\rangle}{C_2}\ \mathit{isTop}(C_1) \qquad \mathit{Seq}_{\mathit{Bot}}\ \frac{\mathit{Seq}\langle C_1, C_2\rangle}{\bot}\ \mathit{isBot}(C_1)$$

Reparation $Rep\langle C, C\rangle$

If the main clause is violated, the reparation clause (C_r) must then be satisfied.

$$Rep_{Thru}\frac{C \dashrightarrow C'}{Rep\langle C_r, C\rangle \dashrightarrow Rep\langle C_r, C'\rangle}$$

$$Rep_{Top}\frac{Rep\langle C_r, C\rangle}{\top} isTop(C) \qquad Rep_{Bot}\frac{Rep\langle C_r, C\rangle}{C_r} isBot(C)$$

Relative delay $Wait\langle \mathbb{T}_r, C\rangle$

Wait a relative amount of time before enacting a clause.

$$Wait_{o}\frac{Wait\langle 0, C\rangle}{C}$$

$$Wait_{Del}\frac{}{Wait\langle z, C\rangle \rightsquigarrow Wait\langle z-1, C\rangle} z \geq 1$$

Absolute lower bound $After\langle \mathbb{T}_a, C\rangle$

Wait until an absolute time stamp before enacting a clause.

$$After\frac{After\langle t, C\rangle}{C}\Gamma[t_0] \geq t$$

Relative deadline $Within\langle \mathbb{T}_r, C\rangle$

A clause which must be satisfied within a relative amount of time.

$$Within_{Thru}\frac{C \dot{\rightarrow} C'}{Within\langle z, C\rangle \dot{\rightarrow} Within\langle z, C'\rangle} \qquad Within_{Del}\frac{C \rightsquigarrow C'}{Within\langle z, C\rangle \rightsquigarrow Within\langle z-1, C'\rangle} z \geq 1$$

$$Within_{Exp}\frac{Within\langle 0, C\rangle}{\bot} notTop(C)$$

$$Within_{Top}\frac{Within\langle z, C\rangle}{\top} isTop(C) \qquad Within_{Bot}\frac{Within\langle z, C\rangle}{\bot} isBot(C)$$

Absolute deadline $\mathit{Before}\langle \mathbb{T}_a, C\rangle$

A clause which must be satisfied before an absolute time stamp.

$$\mathit{Before}_{\mathit{Thru}}\frac{C \dashrightarrow C'}{\mathit{Before}\langle t, C\rangle \dashrightarrow \mathit{Before}\langle t, C'\rangle}\Gamma[t_0] < t$$

$$\mathit{Before}_{\mathit{Exp}}\frac{\mathit{Before}\langle t, C\rangle}{\bot}\Gamma[t_0] \geq t, \mathit{notTop}(C)$$

$$\mathit{Before}_{\mathit{Top}}\frac{\mathit{Before}\langle t, C\rangle}{\top}\Gamma[t_0] < t, \mathit{isTop}(C) \qquad \mathit{Before}_{\mathit{Bot}}\frac{\mathit{Before}\langle t, C\rangle}{\bot}\mathit{isBot}(C)$$

Waiting relative deadline $\mathit{In}\langle \mathbb{T}_r, C\rangle$

Similar to *Within*, a clause which must be satisfied within a relative amount of time. However, this is only checked when the deadline expires.

$$\mathit{In}_{\mathit{Thru}}\frac{C \stackrel{\cdot}{\rightarrow} C'}{\mathit{In}\langle z, C\rangle \stackrel{\cdot}{\rightarrow} \mathit{In}\langle z, C'\rangle}z \geq 1 \qquad \mathit{In}_{\mathit{Del}}\frac{C \rightsquigarrow C'}{\mathit{In}\langle z, C\rangle \rightsquigarrow \mathit{In}\langle z-1, C'\rangle}z \geq 1$$

$$\mathit{In}_{\mathit{Top}}\frac{\mathit{In}\langle 0, C\rangle}{\top}\mathit{isTop}(C) \qquad \mathit{In}_{\mathit{Bot}}\frac{\mathit{In}\langle 0, C\rangle}{\bot}\mathit{notTop}(C)$$

Waiting absolute deadline $\mathit{At}\langle \mathbb{T}_a, C\rangle$

Similar to *Before*, a clause which must be satisfied before an absolute time stamp. However, this is only checked when the deadline expires.

$$\mathit{At}_{\mathit{Thru}}\frac{C \dashrightarrow C'}{\mathit{At}\langle t, C\rangle \dashrightarrow \mathit{At}\langle t, C'\rangle}\Gamma[t_0] < t$$

$$\mathit{At}_{\mathit{Top}}\frac{\mathit{At}\langle t, C\rangle}{\top}\Gamma[t_0] \geq t, \mathit{isTop}(C) \qquad \mathit{At}_{\mathit{Bot}}\frac{\mathit{At}\langle t, C\rangle}{\bot}\Gamma[t_0] \geq t, \mathit{notTop}(C)$$

Non-expiring guard $\mathit{When}\langle G, C\rangle$

Enact a clause when the guard condition is met.

$$\mathit{When}\ \frac{\mathit{When}\langle G, C\rangle}{C}\ \Gamma \vdash G$$

Relative expiring guard $\mathit{WhenWithin}\langle \mathbb{T}_r, G, C\rangle$

A guarded clause which expires (with $\top$) within a relative amount of time.

$$\mathit{WhenWithin}_{Sat}\ \frac{\mathit{WhenWithin}\langle z, G, C\rangle}{C}\ \Gamma \vdash G$$

$$\mathit{WhenWithin}_{Exp}\ \frac{\mathit{WhenWithin}\langle 0, G, C\rangle}{\top}\ \Gamma \nvdash G$$

$$\mathit{WhenWithin}_{Del}\ \frac{}{\mathit{WhenWithin}\langle z, G, C\rangle \rightsquigarrow \mathit{WhenWithin}\langle z-1, G, C\rangle}\ \Gamma \nvdash G, z \geq 1$$

Absolute expiring guard $\mathit{WhenBefore}\langle \mathbb{T}_a, G, C\rangle$

A guarded clause which expires (with $\top$) after an absolute time stamp.

$$\mathit{WhenBefore}_{Sat}\ \frac{\mathit{WhenBefore}\langle t, G, C\rangle}{C}\ \Gamma \vdash G, \Gamma[t_0] < t$$

$$\mathit{WhenBefore}_{Exp}\ \frac{\mathit{WhenBefore}\langle t, G, C\rangle}{\top}\ \Gamma[t_0] \geq t$$

Paper III

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.

Contents

III.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 [67]. 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 two 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 III.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 III.4, as proof-of-concepts to show the feasibility of our approach. Before concluding in the last section, we discuss related work in Section III.5.

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

Figure III.1: $\mathcal{CL}$ syntax.

III.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.

III.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 [77, 78, 74]. $\mathcal{CL}$ 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 III.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 [78]. A contract in $\mathcal{CL}$ may be obtained by using the syntax grammar rules shown in Figure III.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 $\bot$ 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 III.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.* [62]). 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 [77, 76] 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.*

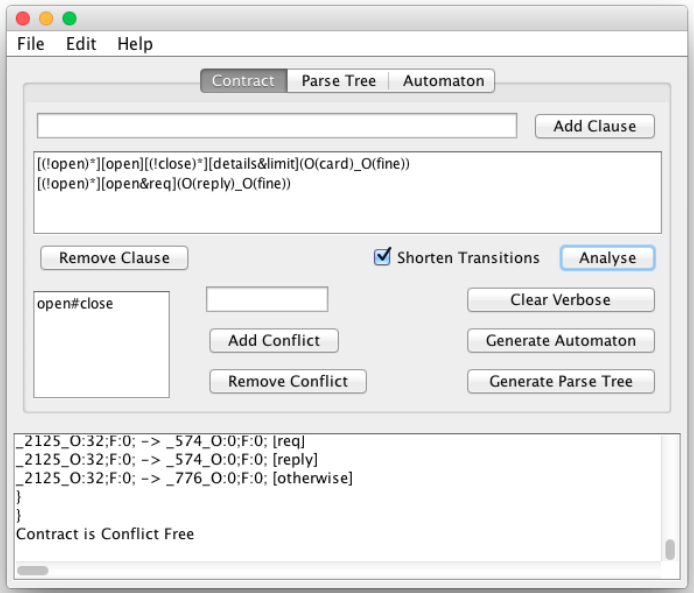
This penalty must be applied in case the ground crew does not respect the above obligations. Assuming that p represents the phrase *"paying a fine"*, one would capture all the above in $\mathcal{CL}$ as $[a]O_{O(p)}(b\&c)$.

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 III.2.3.

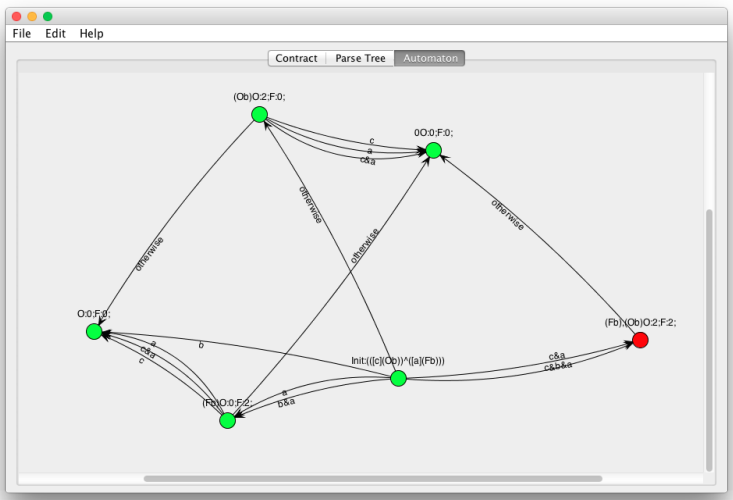
III.2.2 CLAN

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 [32]. 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/



(a) Main user interface



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

Figure III.2: Screenshots of the CLAN tool [32]

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 III.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 III.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 III.3.2.

III.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 [67, 36, 19], 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 [93].

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 III.3.2.

III.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 [67], 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 [45], 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 [79]. This is a realisation of the separation between *tectogrammatical* and *phenogrammatical* features as was first proposed by Curry [26].

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) [80]. 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 [3]. 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.

III.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.

III.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 III.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 III.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 III.3.2).

The entire system is summarised in Figure III.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 III.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 re-linearising these trees using the $\mathcal{CL}$ concrete syntax (see Section III.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

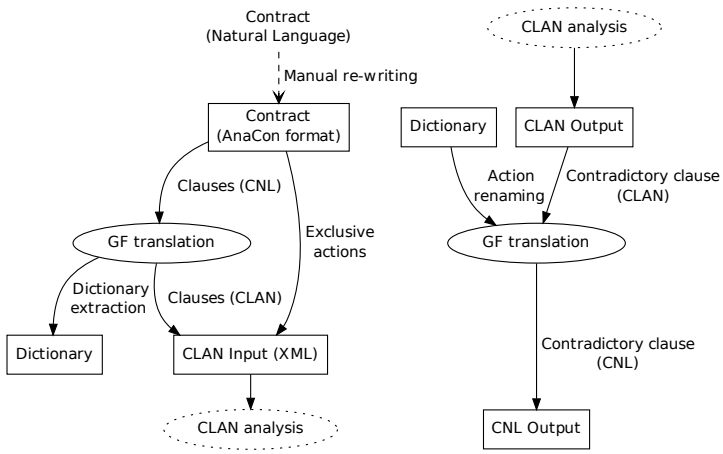


Figure III.4: AnaCon processing workflow.

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.

III.3.2 About the CNL

Wyner et al. [93] identify 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 [48, 65]. 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% [87]). 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 $\bot$. This is also the only way to introduce $\bot$ in the logic. Similarly to 0 and 1 for actions, the clauses $\top$ and $\bot$ cannot be used directly in the CNL.

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

[3] http://www.ietf.org/rfc/rfc2119.txt

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.

III.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 III.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 III.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 triples 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:

(i) *the crew is required to close the check-in desk*

(ii) *the crew shall close the check-in desk*

(iii) *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} ;
```

III.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.

III.4.1 Case Study 1: Airline check-in process

Our first case study has been taken from [31]. It consists of the description of the check-in process of an airline company, given in Figure III.5.

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 III.5: Airline contract case study. [31]

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 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:

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

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:

$\boldsymbol{\mathcal{CL}}$: $[b6](F(a1) \wedge F(a4))$

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 III.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:

$\boldsymbol{\mathcal{CL}}$: $[b4](O(b1\&a2) \wedge [b6](O(b2) \wedge (F(a1) \wedge F(a4))))$

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 III.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
      ↪ ...
```

$\mathcal{CL}$: $[a5](O(a8\&\ldots) \wedge (O(b5) \wedge [b5](\ldots)))$

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}
```

$\mathcal{CL}$: $[a6](O(a9\&\ldots) \wedge [a5](O(b6) \wedge [b6](O(b2) \wedge (F(a7) \wedge F(a4)))))$

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 III.4.3 and Section III.6.

III.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 [69]. The fragment of the contract which we will consider is reproduced in Figure III.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:

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 III.6: ISP Contract case study.

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}
```

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

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.

III.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.

```
(((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 III.7: Sample CLAN output.

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 Figure III.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 III.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).

III.5 Related work

The basic ideas of this journal paper have appeared on the workshop paper [67], where the conceptual model of AnaCon was first introduced. The only commonalities between our current version of AnaCon and the one in [67] are the use of $\mathcal{CL}$ [78] and CLAN [32], besides the overall idea of the framework. In [67] it was shown that it was possible to relate the formal language for contracts $\mathcal{CL}$ and a restricted NL by using GF [79]. Our CNL, however, is based on a formal grammar inspired from NL sentences (i.e. using a subject, verb and complement) unlike that in [67] 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.* [93]).

A notable example of this is Attempto Controlled English (ACE) [37]. 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 [5] 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 [70], 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. [19], 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 [19] cannot be used for diverse contracts as in our case.

Our work is also similar to [44] 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 [47] 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 [46] 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. [81] 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 [62], and references therein.

III.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 agnostic towards 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.

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 [72].

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 III.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 III.4.1 and Section III.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 [86].

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 [54].

Another interesting future work concerns the use of passage retrieval tools like JIRS [81, 15] 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.

Paper IV

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.

Contents

IV.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 [28], 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 IV.2.4);
2. A CNL for *C-O Diagrams* implemented using the Grammatical Framework (GF), precisely mapping to the formal grammar of the diagrams (Section IV.3).
3. Tools for visualising and manipulating *C-O Diagrams* (Section IV.2):
   (i) A web-based visual editor for *C-O Diagrams*;
   (ii) A web-based CNL editor with real-time validation;
   (iii) An XML format COML used as a storage and interchange format.

We also present a small example to show our CNL in practice (Section IV.4) and an an initial evaluation of the CNL (Section IV.5). In what follows we provide some background for *C-O Diagrams* and GF.

IV.1.1 *C-O Diagrams*

Introduced by Martínez et al. [60], *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 IV.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;

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

Figure IV.1: Formal syntax of *C-O Diagrams* [28], where a is an agent, n a name, g a guard, tr a timing restriction, and x is an action.

(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 IV.1. For an example of a *C-O Diagram*, see Figure IV.5 (this example will be explained in more detail in Section IV.4).

IV.1.2 Grammatical Framework

GF [79] 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) [80]. 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

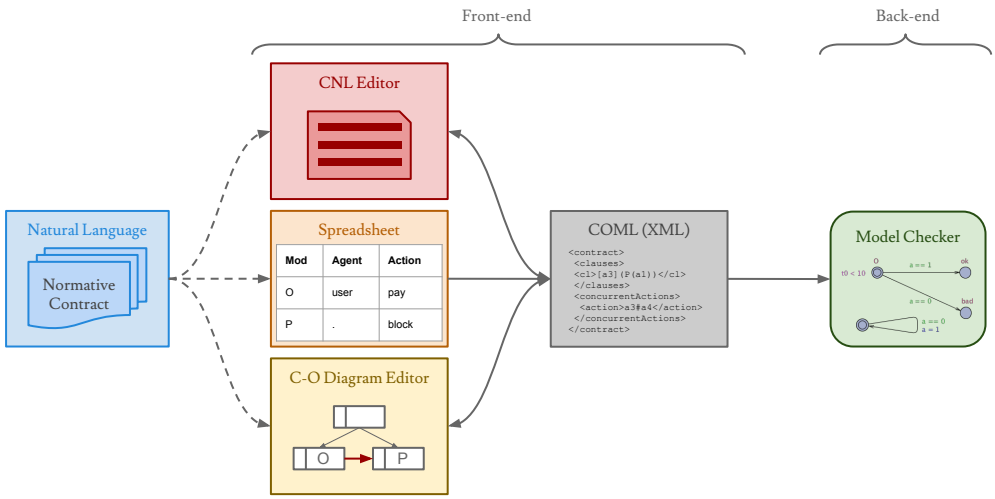


Figure IV.2: The contract processing framework. Dashed arrows represent manual interaction, solid ones automated interaction.

development and making it easier to port the system to other languages.

IV.2 Implementation

IV.2.1 Architecture

The contract processing framework presented in this work is depicted in Figure IV.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 [56].

The front-end, which is the focus of this paper, is a collection of web tools that communicate using our XML format named COML. This format closely resembles the *C-O Diagram* syntax (Figure IV.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 [30] local storage for exchanging data.

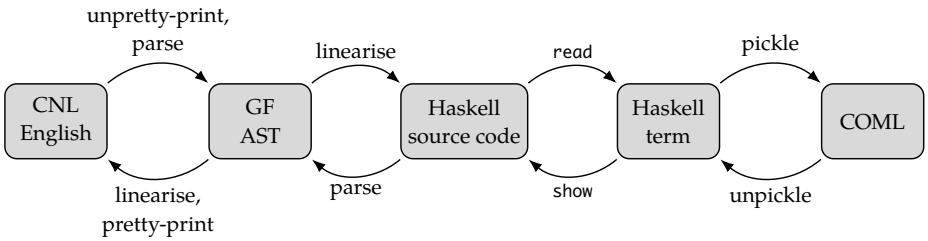


Figure IV.3: Conversion process from CNL to COML and back.

IV.2.2 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 IV.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 (Section IV.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[1], which generates both a parser and generator from a single *pickler* function. The entire process is summarised in Figure IV.3.

IV.2.3 Editing tools

The visual editor allows users to visually construct and edit *C-O Diagrams* of the type seen in Section IV.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.

[1] https://hackage.haskell.org/package/hxt

IV.2.4 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_n for each box n. When the action or sub-contract n is completed, the clock t_n is reset, allowing the user to refer to the time elapsed since the completion of a particular box.

IV.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 IV.4.

IV.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.

IV.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 IV.6.

Each contract clause n in a *C-O Diagram* has an implicit timer associated with it called t_n, 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.

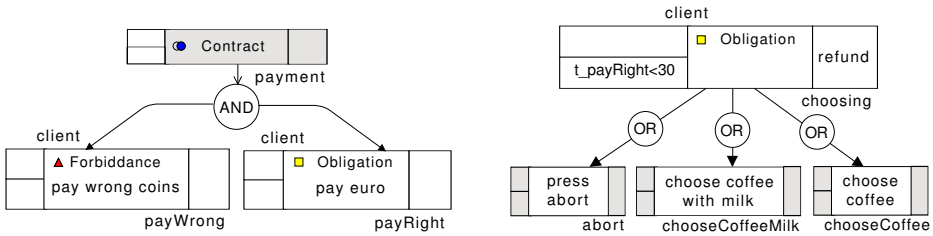
Names

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 IV.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.

IV.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 IV.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 IV.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 IV.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 IV.6.

The *C-O Diagram* for the final contract is shown in Figure IV.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.

IV.5 Evaluation

IV.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.

IV.5.2 Classification

Kuhn suggests the PENS scheme for the classification of CNLs [53]. 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.

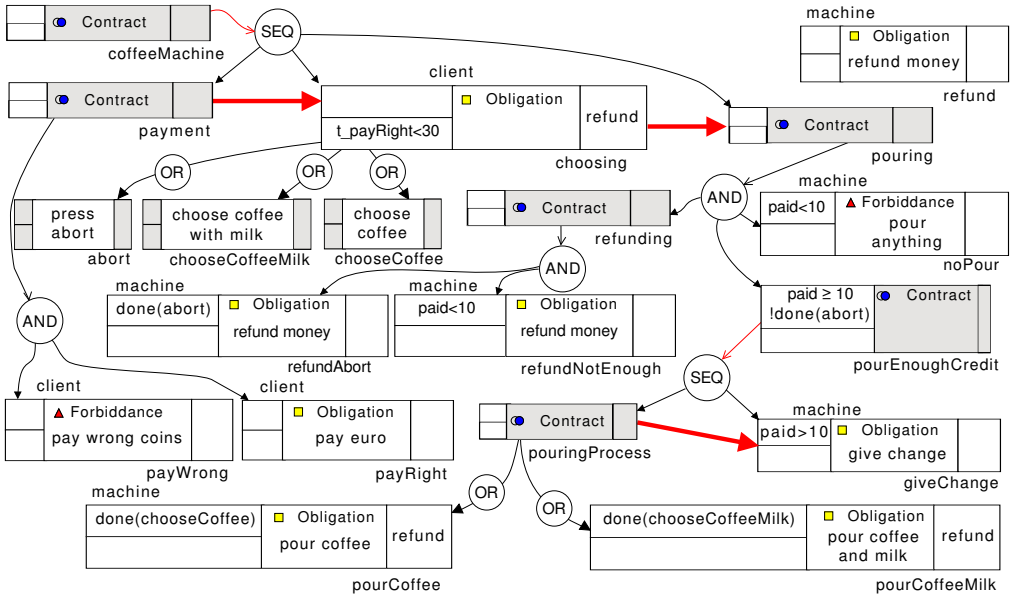


Figure IV.5: The complete *C-O Diagram* for the coffee machine example.

```
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 IV.6: The final verbalisation for the coffee machine example.

IV.6 Related work

C-O Diagrams may be seen as a generalisation of $\mathcal{CL}$ [77, 78, 76] 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 [32]. 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) [37] is a CNL for universal domain-independent use. It comes with a parser to discourse representation structures and a first-order reasoner RACE [35]. 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 [55] as excellent examples of how UI design can help towards solving the problems of writability with CNLs.

IV.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 IV.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 machine learning techniques. This will help the first part of the modelling process by generating

[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.

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 IV.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.

Paper V

Extracting formal models from normative texts

John J. Camilleri, Normunds Grūzītis and Gerardo Schneider

Abstract. *Normative texts* are documents based on the deontic notions of obligation, permission, and prohibition. Our goal is model such texts using the *C-O Diagram* formalism, making them amenable to formal analysis, in particular verifying that a text satisfies properties concerning causality of actions and timing constraints. We present an experimental, semi-automatic aid to bridge the gap between a normative text and its formal representation. Our approach uses dependency trees combined with our own rules and heuristics for extracting the relevant components. The resulting tabular data can then be converted into a *C-O Diagram*.

An extended version of this paper is available at: http://arxiv.org/abs/1706.04997

Contents

V.1 Introduction

Normative texts are concerned with what must be done, may be done, or should not be done (*deontic norms*). This class of documents includes contracts, terms of services and regulations. Our aim is to be able to query such documents, by first modelling them in the deontic-based *C-O Diagram* [28] formal language. Models in this formalism can be automatically converted into networks of timed automata [2], which are amenable to verification. There is, however, a large gap between the natural language texts as written by humans, and the formal representation used for automated analysis. The task of modelling a text is completely manual, requiring a good knowledge of both the domain and the formalism. In this paper we present a method which helps to bridge this gap, by automatically extracting a partial model using NLP techniques.

We present here our technique for processing normative texts written in natural language and building partial models from them by analysing their syntactic structure and extracting relevant information. Our method uses dependency structures obtained from a general-purpose statistical parser, namely the Stanford parser [52], which are then processed using custom rules and heuristics that we have specified based on a small development corpus in order to produce a table of predicate candidates. This can be seen as a specific information extraction task. While this method may only produce a *partial* model which requires further post-editing by the user, we aim to save the most tedious work so that the user (knowledge engineer) can focus better on formalisation details.

V.2 Extracting predicate candidates

The proposed approach is application-specific but domain-independent, assuming that normative texts tend to follow a certain specialised style of natural language, even though there are variations across and within domains. We do not impose any grammatical or lexical restrictions on the input texts, therefore we first apply the general-purpose Stanford parser acquiring a syntactic dependency tree representation for each sentence. Provided that the syntactic analysis does not contain significant errors, we then apply a number of interpretation rules and heuristics on top of the dependency structures. If the extraction is successful, one or more predicate candidates are acquired for each input sentence as shown in Table V.1. More than one candidate is extracted in case of explicit or implicit coordination of subjects, verbs, objects or main clauses. The dependency representation allows for a more straightforward predicate extraction based on syntactic relations, as compared to a phrase-structure representation.

Table V.1: Sample input text and partial output in tabular form, where **Ref.** stands for *refinement*, and **Mod.** stands for *modality*.

1. *You must not, in the use of the Service, violate any laws in your jurisdiction (including but not limited to copyright or trademark laws).*

Ref.	**Mod.**	**Subject** (S)	**Verb** (V)	**Object** (O)	**Modifiers**
	F	User	violate	law	V: in User's jurisdiction V: in the use of the Service

2. *You will not post unauthorised commercial communication (such as spam) on Facebook.*

	F	User	post	unauthorised commercial communication	O: such as spam O: on Facebook

3. *You will not upload viruses or other malicious code.*

	F	User	upload	virus	
OR	F	User	upload	other malicious code	

4. *Your login may only be used by one person - a single login shared by multiple people is not permitted.*

	P	person	use	login of User	S: one

5. *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	V: under this rental agreement
AND	O	renter	pay	other fee	V: under this rental agreement

6. *The equipment shall be delivered to renter and returned to owner at the renter's risk.*

	O	equipment	[is] delivered [to]	renter	V: at renter's risk
AND	O	equipment	[is] returned [to]	owner	V: at renter's risk

V.2.1 Expected input and intended output

The basic requirement for pre-processing the input text is that it is split by sentence and that only relevant sentences are included. In this experiment, we have manually selected 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 and thus the output of our tool.

The output is a table where each row corresponds to a *C-O Diagram* box (clause), containing fields for: **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 dec-

laration (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 modifiers indicating temporality; **Adverbials:** other adverbial phrases that modify the action; **Conditions:** phrases indicating conditions on agents, actions or objects; **Notes:** other phrases providing additional information (e.g. relative clauses), indicating the head word they attach to.

Values of the Subject, Verb and Object fields undergo certain normalisation and formatting: head words are lemmatised; Saxon genitives are converted to of-constructions if contextually possible; the preposition "to" is explicitly added to indirect objects; prepositions of prepositional objects are included in the Verb field as part of the predicate name, as well as the copula if the predicate is expressed by a participle, adjective or noun; 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. There is also the question of what is considered *correct* output. It may also be the case that certain clauses can be encoded in multiple ways, and, while all fields may be filled, the user may find it more desirable to change the encoding.

V.2.2 Rules

We make a distinction between rules and heuristics that are applied on top of the Stanford dependencies. Rules are everything that explicitly follow from the dependency relations and part-of-speech tags. For example, the head of the subject noun phrase (NP) is labelled by `nsubj`, and the head of the direct object NP by `dobj`; fields Subject and Object of the output table can be straightforwardly populated by the respective phrases (as in Table V.1).

We also count as lexicalised rules cases when the decision can be obviously made by considering both the dependency label and the head word. For example, modal verbs and other auxiliaries of the main verb are labelled as `aux` but words like "may" and "must" clearly indicate the respective modality (P and O). Auxiliaries can be combined with other modifiers, for example, the modifier "not" (`neg`) which indicates prohibition. 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 (as shown by the fourth example in Table V.1).

V.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 the language of normative texts.

First of all, auxiliaries are compared and classified against extended lists of keywords. For example, the modal verb "can" most likely indicates permission while "shall" and "will" indicate obligation. In addition to auxiliaries, we consider the predicate itself (expressed by a verb, adjective or noun). For example, words like "responsible" and "require" most likely express obligation.

For prepositional phrases (PP) which are direct dependants of Verb, we first check if they reliably indicate a temporal modifier and thus should be put in the Time field. The list of such prepositions include "after", "before", "during" etc. If the preposition is ambiguous, the head of the NP is checked if it bears a meaning of time. There is a relatively open list of such keywords, including "day", "week", "month" etc. Due to PP-attachment errors that syntactic parsers often make, if a PP is attached to Object, and it has the above mentioned indicators of a temporal meaning, the phrase is put in the Verb-dependent Time field.

Similarly, we check the markers (`mark`) of adverbial clauses if they indicate time ("while", "when" etc.) or a condition (e.g. "if"), as well as values of simple adverbial modifiers, looking for "always", "immediately", "before" etc. Adverbial modifiers are also checked against a list of irrelevant adverbs used for emphasis (e.g. "very") or as gluing words (e.g. "however", "also").

Subject and Object are checked for attributes: if it is modified by a number, the modifier is treated as a condition and is separated in the respective field.

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 Verb is treated as a prepositional object and thus is included in the Object field.

Additionally, 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 do such a simple but effective anaphora resolution (see Table V.1).

Table V.2: Evaluation results based on a small set of test sentences (10 per document).

Document	Rules only			Rules & heuristics		
	Precision	*Recall*	F_1	*Precision*	*Recall*	F_1
Ph.D.	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

V.3 Experiments

In order to test the potential and feasibility of the proposed approach, we have selected four normative texts from three different domains:

1. Ph.D. regulations from Chalmers University;
2. Rental agreement from RSO, Inc.;
3. Terms of service for GitHub; and
4. Terms of service for Facebook.

In the development stage, we considered first 10 sentences of each document, based on which the rules and heuristics were defined. For the evaluation, we used the next 10 sentences of each document.

We use a simple precision-recall metric over the following fields: Subject, Verb, Object and Modality. The other fields of our table structure are not included in the evaluation criteria as they are intrinsically too unstructured and will always require some post-editing in order to be formalised. The local scores for precision and recall are often identical, because a sentence in the original text would correspond to one row (clause) in the table. This is not the case when unnecessary refinements are added by the tool or, conversely, when co-ordinations in the text are not correctly added as refinements.

The evaluation was performed twice: first 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 V.2, including the harmonic mean scores (F_1) between precision and recall. The first observation from the results is that the F_1 score varies quite a lot between documents; from 0.49 to 0.86. This is mainly due to the variations in language style present in the documents. Overall the application of heuristics together with the rules does improve the scores obtained.

On the one hand, many of the sentence patterns which we handle in the heuristics appear only in the development set and not in the test set. On the other hand, there are few cases which

occur relatively frequently among the test examples but are not covered by the development set. For instance, the introductory part of a sentence, the syntactic main clause, is sometimes pointless for our formalism, and it should be ignored, taking instead the sub-clause as the semantic main clause, e.g. *"User understands that […]"*.

The small corpus size is of course an issue, and we cannot make any strong statements about the coverage of the development and test sets. Analysing the modal verb "shall" is particularly difficult to get right. It may either be an indication of an obligation when concerning an action, or it may be used as a prescriptive construct as in "shall be" which is more indicative of a declaration. The task of extracting the correct fields from each sentence can be seen as paraphrasing the given sentence into one of the known patterns, which can be handled by rules. The required paraphrasing, however, is often non-trivial.

V.4 Related work

Our work can be seen as similar to that of Wyner and Peters [94], who present a system for identifying and extracting rules from legal texts using the Stanford parser and other NLP tools within the GATE system. Their approach is somewhat more general, producing as output an annotated version of the original text. Ours is a more specific application of such techniques, in that we have a well-defined output format which guided the design of our extraction tool, which includes in particular the ability to define clauses using refinement.

Mercatali et al. [63] tackle the automatic translation of textual representations of laws to a formal model, in their case UML. This underlying formalism is of course different, and the main interest is in the hierarchical structure of the documents rather than the norms themselves. Their method does not use dependency or phrase-structure trees but shallow syntactic chunks.

Cheng et al. [22] also describe a system for extracting structured information for texts in a specific legal domain. Their method combines surface-level methods like tagging and named entity recognition (NER) with semantic analysis rules which were hand-crafted for their domain and output data format.

V.5 Conclusion

Our main goal is to perform formal analyses of normative texts through model checking. In this paper we have briefly described how we can help to bridge the gap between natural language texts and their formal representations. Though the results reported here are indicative at best

(due to the small test corpus), the application of our technique to the case studies we have considered has definitely helped increase the efficiency of their encoding into *C-O Diagrams*. Future plans include extending the heuristics, comparing the use of other parsers, and applying our technique to larger case studies.

Paper VI

Contract Verifier: A web-based tool for analysing normative documents in English

John J. Camilleri, Mohammad Reza Haghshenas and Gerardo Schneider

Abstract. Our goal is to use formal methods to analyse normative documents written in English, such as privacy policies and regulations. This requires the combination of a number of different elements, including information extraction from natural language, formal languages for model representation, and an interface for property specification and verification. A number of components for performing these tasks have separately been developed: a natural language extraction tool, a suitable formalism for representing such documents, an interface for building models in this formalism, and methods for answering queries asked of a given model. In this work, each of these concerns is brought together in a web-based tool, providing a single interface for analysing normative texts in English. Through the use of a running example, we describe each component and demonstrate the workflow established by our tool.

Contents

VI.1 Introduction

Normative texts, or *contracts*, are documents which describe the permissions, obligations, and prohibitions of different parties over a set of actions. They also include descriptions of the penalties which must be paid when the main norms of the document are violated. We frequently encounter such texts in the form of privacy policies, software licenses, workflow descriptions, terms of use, regulations, and SLAs (service-level agreements). Despite being written for human consumption and thus expressed in natural language, these kinds of documents are typically long and difficult to follow, making them hard to analyse manually.

> *What commitments am I agreeing to make?*
> *Can my information be shared with third parties?*
> *How late can I make my payment without facing fines?*

These are the kinds of questions about a contract that we may want answered, both as users and as authors. Using text-based search to find such answers can be tedious and unreliable, for example when clauses cross-reference each other, and when the document contains exceptions and timing constraints. Our goal is to bring formal methods to this kind of natural language analysis, packaged as a tool which is usable by non-experts and which requires as little understanding of the underlying technologies as possible. We do this by first modelling these documents using a suitable formalism, which then makes them amenable to verification using standard techniques. This includes answering queries based on a syntactic traversal of the model, as well as using model-checking to verify temporal properties, by converting the model to a timed automata representation.

The main components required for this have been individually described in previous papers [17, 20, 21]. This paper presents a new web-based tool for the analysis of normative texts in English, bringing each of these components together into a single interface. This integration requires the development of suitable transformations between formats, besides maintaining a *dictionary* of key elements at each level of abstraction in order to guarantee the translation from/to the natural language text and the different formal representations. Our novel contributions also include a set of query templates in natural language and a method for processing counter-example traces, allowing users to interact with the system completely in English.

The rest of the paper continues as follows. Section VI.2 introduces the structure of the tool and gives an overview of the workflow it establishes. In Section VI.3 we then present the running example which will be used throughout the paper to demonstrate the use of our tool. Section VI.4 describes extraction and building a model through post-editing of tabular data, including ver-

balisation of an existing model using controlled natural language. Section VI.5 describes the analysis which can be performed on the model, by using a set of query templates which can be customised based on the current model. The software architecture is then summarised in Section VI.6. Section VI.7 takes a look at some related work in this area, and we conclude with a summary of the benefits and limitations of our approach in Section VI.8.

VI.2 The *Contract Verifier* tool

The main contribution of this work is the *Contract Verifier*, a web-based tool for modelling and analysing normative texts in English. Figure VI.1 shows an overview of the workflow which our tool covers, summarised in the following steps:

1. Users start with an English text which they wish to build a model out of.

2. The text is submitted to an **extraction** phase which attempts to automatically extract the clauses from the text, each of which concerning at least an agent, action and modality.

3. The results of the extraction are shown to the user in a tabular format where each cell is editable. At this point, the user must check the results of the extraction and **post-edit** the clauses which are not completely correct.

4. Once the user is happy with the model in tabular format, this is **converted** into an actual model, which is internally represented in an XML format.

5. From here, the model can be verbalised in a controlled natural language and viewed in a compact formal notation.

6. The user then performs **analysis** by selecting the queries which should be run against the model. Queries are presented as English sentence templates, which may include slots for specifying relevant arguments.

7. The **queries** are then submitted to the server which computes the **results** and displays them to the user beside each respective query.

8. If the answer to a query is not as expected, this could point to either:

   (a) a problem with the contract model, in which case the user may go back and edit the model and repeat the steps as above; or

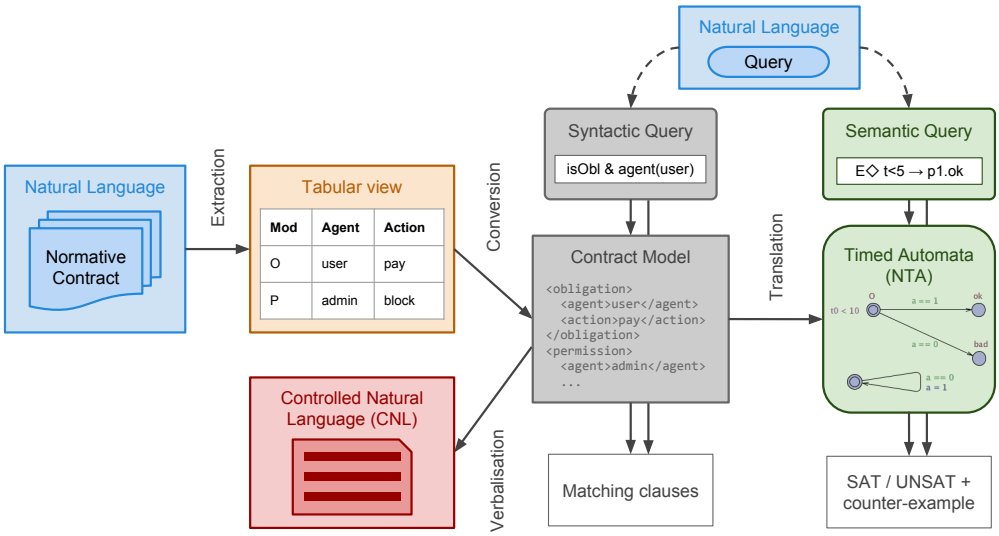


Figure VI.1: Overview of the contract analysis workflow established by the *Contract Verifier*.

(b) a problem with the original normative document, where the user may then modify the text and perform the analysis again, or simply leave the original formulation as it is (e.g., the analysis might detect a lack of a deadline associated with a given obligation, but this might be considered desirable by the user).

A demonstration of the *Contract Verifier* is openly available on the web.[1] Upon visiting this URL, users will be asked to log in. New users may create an account so that their work can be saved under their profile. Alternatively, one may log in with a guest account, using username `guest@demo` and password `contract`.

VI.3 Running example

To show how our tool is used in practice, we pick a running example of a normative text describing the rules of a university course (see Figure VI.2). This example is based on courses held at our own department, covering the requirements for passing the course and the deadlines for the submission and grading of assignments. It has been chosen as an example because it is concise yet contains a variety of temporal constraints and dependencies between clauses. The text itself has been written by ourselves.

[1] http://remu.grammaticalframework.org/contracts/verifier/

Students need to register for the course before the registration deadline, one week after the course has started.
To pass the course, a student must pass both the assignment and the exam.
The deadline for the first assignment submission is on day 10.
Graders should correct an assignment within one week of it being submitted.
If the submission is not accepted, the student will have until the final deadline on day 25 to re-submit it.
The exam will be held on day 60.
Registered students must sign up for the exam latest 2 weeks before it is held.

Figure VI.2: Example of a normative text describing the rules involved in the running of a university course. The course is assumed to start on day 0. This text is used as the running example throughout this paper.

Before publishing these rules as an official course description, the authors (teachers/administrators) may wish to ensure that all the requirements for passing the course are enforced and that the rules are consistent. Once published, end users of these rules (students) may wish to query the parts which are relevant to them or work out any flexibilities in their deadlines.

VI.4 Building a contract model

VI.4.1 Extraction from English

The first step towards building a formal model of our natural language contract is to process the text in order to see what clause information can be extracted automatically. This is done using the ConPar tool [17], which can extract partial contract models from English normative texts. ConPar is a natural language processing (NLP) tool, which uses the Stanford Parser [52] to obtain dependency trees for each sentence in the input text. These trees are then processed by ConPar, which uses the dependency representation to attempt to extract information related to:

(i) subject (agent)
(ii) verb and object (action)
(iii) modality (obligation, permission, prohibition)
(iv) temporal and non-temporal conditions.

In addition, ConPar will also try to identify refinement clauses, where a single input sentence may translate to multiple sub-clauses joined together using a connective such as conjunction, choice or sequence.

The output of the ConPar tool is a tabular representation of the extracted data, which is produced in a tab-separated value (TSV) format. This is a simple format which can be easily

Sentence	No	Modality	Subject	Verb	Object	Time	Conditions
Students need to register for the course before the registration deadline , one week after the course has started .	1	D	student	need	to register		
To pass the course , a student must pass both the assignment and the exam .	2	AND[2.1,2.2]					
	2.1	O	student	pass	exam		
	2.2	O	student	pass	assignment		
The deadline for the first assignment submission is on day 10 .	3	D	deadline	be on	day		DAY: 10
Graders should correct an assignment within one week of it being submitted .	4	{should}	grader	correct	assignment	within one week of it being submitted	
If the submission is not accepted , the student will have until the final deadline on day 25 to re-submit it .	5	O	student	have until	final deadline	on day 25	HAVE: if the submission is not accepted

Figure VI.3: Screenshot showing the output from passing our normative text through the extraction tool (image has been cropped to improve readability). Each row indicates a clause in the model. Note how the second English sentence has been refined into multiple sub-clauses. The O in the modalities column stands for *obligation*, while D stands for *declaration*.

loaded in any spreadsheet or table-editing software. In this representation, each row corresponds to a clause while the columns indicate the various components, as listed above. The result of passing our example through this tool is shown in Figure VI.3.

VI.4.2 Post-editing

While quite a lot of clause information has been correctly extracted by the tool, there are still some errors which need to be corrected manually by the user. The use of a tabular format for displaying the output of the extraction phase facilitates this post-editing. All cells in the table are editable, and rows can be added/deleted as needed.

For example, in the case of clause 4 (Figure VI.3), the value in the *Modality* field (`{should}`) should be replaced with the obligation specifier `0`. Additionally, the value of *Time* field (`within one week of it being submitted`) can be encoded with the phrase `within 7`, which is the accepted syntax for expressing this kind of time restraint.

VI.4.3 Conversion to contract model

After post-editing the extracted clause information, this can be converted into a formal contract model. The formalism used for representing contracts is based on the deontic modalities of **obligation**, **permission** and **prohibition** of agents over actions. It includes constructs for refining clauses by conjunction, choice and sequence, and includes the possibility to specify reparations for when a clause is violated. Clauses can be constrained by temporal restrictions or guarded based on the status of other clauses. This formalism, which is based on *C-O Diagrams* [28], is described fully in [21].

This conversion step is implemented as a straight-forward script which takes the TSV representation as input and produces a contract model file. We refer to the format of this file as COML, which is an XML-based format for storing contract models in our formalism. Once the conversion is complete, the COML file is stored on the server and the user can view it using three different representations simultaneously:

(i) post-edited input text,
(ii) controlled natural language (CNL), and
(iii) a compact formal notation (*C-O Diagram* Shorthand or CODSH).

These are shown in Figure VI.4 and explained in the sections below.

VI.4.4 Verbalisation using CNL

Given a contract model in our formalism, we have developed a method for linearising it as a phrase in a *controlled natural language* (CNL) [93]. A CNL is a reduced version of a natural language (NL) which has limited syntax and vocabulary, making it in fact a formal language and thus expressible using a grammar. CNLs are often used as interfaces for formal languages which are human-friendly, yet still unambiguous and well-defined.

The CNL designed for this contract formalism is described in [20]. We use the Grammatical Framework (GF) [79] for defining the grammar for our CNL and converting to and from our internal formal representation. This also includes the possibility of building a contract model directly using the CNL (rather than using the extraction step), however we do not cover this input method in the present work. The CNL representation may resemble the original NL text

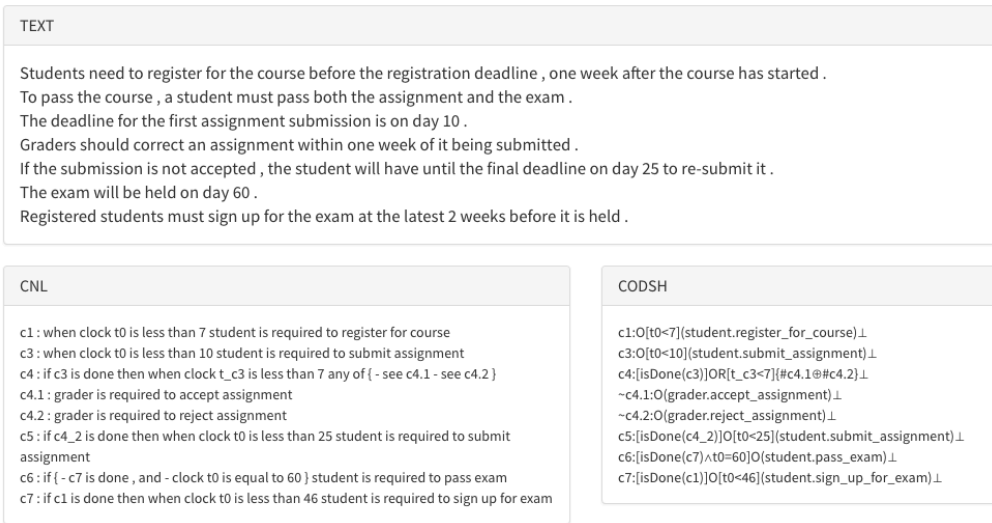


Figure VI.4: Screenshot showing the different output representations of the contract model (image has been cropped to improve readability). Top: the post-edited input text; bottom-left: controlled natural language; bottom-right: compact formal notation.

in some ways, however it is characterised by less variation in the expressions used and by certain structural features such as labels before each clause.

The generation of the CNL representation requires that subjects, verbs and objects are present in the lexicon. The lexicon used is a large-scale English dictionary containing over 64,000 entries, but if the contract contains terms which are not present in this lexicon then the generation to CNL will fail. This failure however will not affect the usability of the rest of the tool.

VI.4.5 Compact formal notation

In addition to the post-edited text and the CNL, we also display a view of the model in a formal syntax. We refer to this notation as *C-O Diagram* Shorthand (CODSH). It is designed mostly for developers who understand the formal structure of the contract model but would like a more condensed representation than the COML format. This can be helpful when debugging. In particular, this notation reveals the names which are automatically assigned to each clause in the conversion phase.

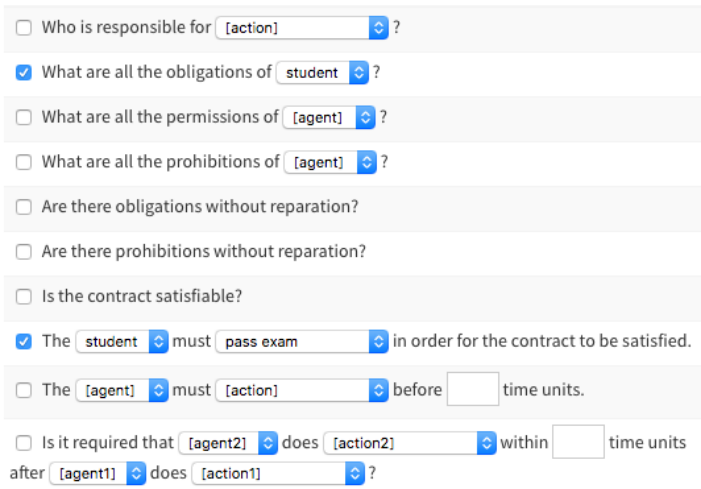


Figure VI.5: Available query templates which users may execute against the contract model, with drop-downs for specifying agents and actions extracted from the model. Queries 1–6 are *syntactic*, while queries 7–10 are *semantic*.

VI.5 Analysis

Once the model has been built, we can perform analysis on the contract by running queries against it. The user is presented with a list of query templates, as shown in Figure VI.5. Each query may have slots for parameters which the user should provide; these are either names of clauses, agents or actions. The possible completions for these slots are extracted automatically from the contract model.

Internally, queries provided by our tool are computed in one of two ways, either through syntactic filtering or via conversion to timed automata and using the UPPAAL verification tool [56]. Both of these techniques are described below.

VI.5.1 Syntactic analysis

Our tool currently provides six different syntactic queries (items 1–6 in Figure VI.5). These queries are *syntactic* in the sense that each can be solved by traversing the contract model and filtering out the corresponding clauses which match the query. As an example, consider the following query:

> *What are the obligations of* [agent]*?*

This query is internally encoded as the following conjunction of predicates:

$$isObl \wedge agentOf([\mathsf{agent}])$$

where [agent] is replaced with a concrete agent name chosen by the user. The solution to this query is computed via a Haskell function which takes the contract model as a Haskell term together with the query as a set of predicate functions, and returns a list of matching clauses. The final output is given as a natural language sentence listing the actions corresponding to the matching clauses. For the example, asking for all the obligations for *student* will give the output below:

> *The following are obligations of student:*
>
> - *register for course*
> - *submit assignment*
> - *sign up for exam*
> - *pass exam*

The way the result is phrased will vary based on the query, as well as the number of items in the result: up to two results are inlined, while three or more are given as bullets. This is to make the response more natural for the user.

VI.5.2 Semantic analysis

Our tool also includes four semantic queries (items 7–10 in Figure VI.5). We use the term *semantic* to refer to those queries which cannot be answered simply by looking at the structure of the model. Consider the following example:

> *The* [agent] *must* [action] *before time* [number].

Determining this must take into consideration the operational behaviour of a contract model, including when actions are performed, how new clauses are enabled and others expire. Processing semantic queries is achieved through using model checking techniques, by first converting a contract model into a network of timed automata [2] and then using the UPPAAL tool to verify temporal properties against the translated model. This idea was introduced for *C-O Diagrams* in [28]; the details of our own translation can be found in [21]. By using verification, we are able to quantify over all possible sequences of events with respect to the contract.

This approach requires that the query itself is encoded as a property in a temporal logic which the model checker can process. In the case of UPPAAL, the property specification language is a subset of TCTL [11]. The example query above is encoded as the following UPPAAL property:

$$\forall \Box\, \mathit{allComplete}() \implies \Big(\mathit{isDone}\big([\mathsf{agent}]\_[\mathsf{action}]\big) \wedge t_0 - \mathit{Clocks}\big[[\mathsf{agent}]\_[\mathsf{action}]\big] < [\mathsf{number}]\Big)$$

Here, *allComplete* and *isDone* are helper functions included in the translated UPPAAL system which allow the status of clauses and actions to be queried from within the timed properties. t_0 is a never-reset clock representing global time, while the *Clocks* array contains a clock for each action in the system, which is reset when that action is performed. The expression $t_0 - \mathit{Clocks}\big[a\big]$ thus gives the absolute time at which action a was completed.

The property is then verified using UPPAAL, which will return a result of **satisfied** or **not satisfied**. In cases where a symbolic trace is produced as part of the verification, this is parsed by our tool in order to provide a meaningful abstraction of it. In this processing step, we pick out the actions performed in the trace along with their time stamps and present these as part of the result to the user. For example, when running the query below on our contract example:

> *The* student *must* register for course *before time* 5.

we get the following result in the tool:

> *NOT Satisfied*
> *The property is violated by the following action sequence:*
> *- student register for course at time 6*
> *- student submit assignment at time 6*
> *- …*

where the remainder of the trace contains the other obliged actions at time 6 or later. This is in fact as we expect; the contract states that students have up to 7 days to register for the course, and thus it is not true that they must have registered before day 5 in order to satisfy the entire contract. If we change the time value in the query from 5 to 7, then the result returned is *Satisfied*.

VI.6 Software architecture

The *Contract Verifier* tool is implemented as a PHP web application, using a MySQL database for storing user accounts and the query templates available in the system. Contract models in COML

Table VI.1: API of services provided in the web server, showing the relevant URL path and input/output formats for each service. The various formats are: TSV (tab separated values), COML (Contract-Oriented XML), CODSH (*C-O Diagram* Shorthand), CNL (Controlled Natural Language), and UPPAAL-compatible XML.

Path	Description	Request	Response
`/nl/tsv`	Clause extraction (ConPar)	English text	TSV
`/tsv/coml`	Convert TSV to COML	TSV	COML
`/coml/codsh`	Show contract in shorthand	COML	CODSH
`/coml/cnl`	Verbalise contract using CNL	COML	CNL
`/coml/syntactic`	Execute syntactic query	Query + COML	Clause names
`/coml/uppaal`	Translate contract to NTA	COML	UPPAAL XML

and UPPAAL-XML format are saved as files on the server. No server-side framework is used. The client-side interface is based on the AdminLTE Control Panel Template[2], and the tabular editing interface makes use of the editableTable jQuery library[3].

The ConPar extraction tool is written in Java, primarily because it uses the Stanford parser which is also implemented in Java. The core of our system is written in Haskell, using algebraic data types to define the structure of a contract model. The conversion from TSV and translation to NTA are thus also written as Haskell functions to and from this data type. The linearisation of a contract model to CNL uses the GF runtime, which is a standalone application. Similarly, executing semantic queries requires running UPPAAL as an external process.

Because of the variety of languages and programs used in our tool chain, we provide a convenient layer over these components in the form of a small server application which provides these separate functionalities as individual web services. This modular approach allows the web application providing the user interface to consume each component via a web API, removing limitations on implementation language and hosting requirements, and allowing a clean separation of concerns between front-end and back-end.

Table VI.1 shows a summary of the API covering all the web services provided by the server. This API is fully documented and publicly accessible online.[4] The server itself is also implemented in Haskell.

[2] https://almsaeedstudio.com/
[3] http://mindmup.github.io/editable-table/
[4] http://remu.grammaticalframework.org:5446/

VI.7 Related work

AnaCon [4] is a similar framework for the analysis of contracts, based on the contract logic $\mathcal{CL}$ [77, 76], which allows for the detection of contradictory clauses in normative texts using the CLAN tool [32]. By comparison, the underlying logical formalism we use, based on *C-O Diagrams*, is more expressive than $\mathcal{CL}$ as it includes temporal constraints, cross-referencing of clauses and more. Besides this, our translation into Uppaal allows for checking more general properties, not only normative conflicts. In addition, their interface for specifying contracts is purely CNL-based and there is no extraction tool from English as in our case.

Information extraction from natural language is of course a field within itself, and even in the domain of contractual documents in English there are many other works with similar goals. The extraction task in our work can be seen as similar to that of Wyner & Peters [94], who present a system for identifying and extracting rules from legal texts using the Stanford parser and other NLP tools within the GATE system. Their approach is somewhat more general, producing as output an annotated version of the original text, whereas we are targeting a specific, well-defined output format. Other similar works include that of Cheng et al. [22], who combine surface-level methods like tagging and named entity recognition (NER) with hand-crafted semantic analysis rules, and Mercatali et al. [63] who extract the hierarchical structure of the documents into a UML-based format using shallow syntactic chunks.

One crucial aspect in any work targeting formal analysis of natural language documents is the confidence in the extraction from a source document to the target formal language. In our tool, the result of the extraction is usually incomplete and some amount of manual post-editing is always generally required. Azzopardi et al. [8] handle this incompleteness using a deontic-based logic including *unknowns*, representing the fact that some parts have not been fully parsed. Furthermore, the same authors present in [7] a tool to support legal document drafting in the form of a plugin for Microsoft Word. Though the final objective of their work diverges from ours, we are both concerned with the translation of natural language documents into a formal language by using an intermediate CNL. The main difference is that they target a more abstract formal language (very much like $\mathcal{CL}$), and as a consequence their CNL is also different. Our formalism allows for richer representations not present in their language (e.g. real time constraints and cross-references). Additionally, they do not target complex analysis of contracts (they only provide a normative conflict resolution algorithm), and we do not provide assistance in the contract drafting process.

There is considerable work in modelling normative documents using representations other than logic-like formalisms such as our own. LegalRuleML [6] is a rule interchange format for the

legal domain, allowing the contents of legal texts to be structured in a machine-readable format. The format aims to enable modelling and reasoning, allowing users to evaluate and compare legal arguments using tools customised for this format. A similar project with a broader scope is MetaLex [14], an open XML interchange format for legal and legislative resources. Its goal is to enable public administrations to link legal information between various levels of authority and different countries and languages, improving transparency and accessibility of legal content. Semantics of Business Vocabulary and Business Rules (SBVR) [68] uses a CNL to provide a fixed vocabulary and syntactic rules for expressing the terminology, facts, and rules of business documents of various kinds. This allows the structure and operational controls of a business to have natural and accessible descriptions, while still being representable in predicate logic and convertible to machine-executable form.

We note that none of the works mentioned above present a single tool for end-to-end document analysis, starting from a natural language text and finally allowing for rich syntactic and semantic queries, as in the case of our tool.

VI.8 Conclusion

In this paper we have presented *Contract Verifier*, a web-based tool for analysing normative documents written in English. The tool brings together a number of different components packaged together as a user-friendly application. We demonstrate a typical workflow through the system, starting with an English text, extracting a contract model from it, and executing different kinds of queries against it. Each of the components used by our tool is implemented as a standalone module, with a web-based API exposing each module as a web service. Individual modules can be easily replaced and new ones can be added, such as introducing a new back-end for runtime verification. Similarly, new interfaces can be built around the existing modules without having to make changes to the underlying modules.

An important feature of the *Contract Verifier* tool is the level of automation it provides: everything except the post-editing of the extracted model (and of course choosing the queries to be performed) is automatic. For example, the names of clauses, agents and actions are automatically extracted from the contract so the user can select them using drop-down menus when making queries. Also, each clause is given a unique identifier as well as a *clock* that is reset when that clause is activated. Though these are mainly intended for internal use when performing semantic queries, users may even use them explicitly in the post-editing phase to encode relative timing constraints.

We see this tool as a successful implementation of a user interface for bringing together var-

ious separate components and providing a clear workflow for analysing normative texts in English. That being said, it is as such a proof-of-concept tool and has not undergone any extensive usability testing or application in real-world scenarios. We conclude here with a critical look at the shortcomings and limitations of the current work.

Evaluation

Our goal is to make the task of analysing a normative document easier and more reliable than if one were to do it completely manually. Measuring whether we achieve this, and to what extent, requires proper assessment. Some evaluation has already been carried out for the natural language extraction part of the workflow (the ConPar tool), measuring the accuracy of tool for extracting a correct model from a normative document. By calculating precision and recall, F_1 scores of 0.49 to 0.86 were obtained for the test set of four documents [17].

However, we currently do not have a thorough evaluation of the complete *Contract Verifier* workflow as presented here. This would take the form of an empirical study comparing document analysis using our tool with a purely manual approach, measuring the amount of post-editing required to build a correct model, the time required to formulate a query and obtain a result, and the overall ease of use of the tool in a qualitative sense. We consider a study of this kind important future work.

Limitations

Extraction. The extraction phase relies on dependency trees and thus takes a syntactic-level approach to parsing. While a fair deal of information can be extracted in this way from simpler sentences, a deeper understanding of a phrase often involves using related or opposite concepts which cannot be determined without more elaborate processing on the semantic level. In addition, we assume that each input sentence translates into one or more clauses, and have no support for detecting when a phrase should actually modify an existing clause instead.

Modelling. Our tool uses a tabular interface to help make the task of modelling user-friendly, but some understanding of the underlying formal language and its semantics is necessary in order to work efficiently with it. For example, our formalism is essentially *action-based*, where clauses prescribe what an agent should or should not *do*. However, empirically we have found that normative documents often describe what should or should not *be*, i.e. referring to state-of-affairs. While these can often be paraphrased to fit into our formalism, this is a non-trivial task

which currently must be done completely by the user. The formalism itself has its own limitations, for example we are unable to encode percentages over quantities or any kind of arithmetic in actions, which are common features in some types of contracts (e.g., in SLAs).

Verification. When it comes to running queries, those which are syntactic can quickly be answered by an algorithm which is linear in the size of the model. For semantic queries however, the conversion to timed automata means that the state-space explosion problem typical of model checking is a potential problem. Certain optimisations made during the translation process could improve this somewhat. For instance, our generated NTA contain many parallel synchronising automata as a result of the modularity of the translation, and some *ad hoc* heuristics could likely be used to reduce the number of automata or the need for certain synchronisations, thus improving the performance. That said, this is ultimately a theoretical problem which we cannot avoid altogether.

Scalability

Extraction. We are limited here by the speed of the ConPar tool, which itself uses the Stanford dependency parser. While parse time is related to the length of the input sentence, in our tests based on the test data from [17] we have found that parsing and extracting a single sentence takes on average roughly half a second.[5]

Post-editing. The tabular interface for editing the extracted clauses has no concrete limits in terms of the number of clauses it can handle. What it does lack is support for managing a document's internal hierarchy (sections and sub-sections), which is a common feature of normative texts.

Analysis. As discussed above, the running of semantic queries is the biggest barrier to the scalability of the system due to the state-space explosion problem. For our small example here, a query requiring a search of the entire search space requires a few milliseconds to complete, but this performance may degrade drastically as the model size increases.

Queries. Our current implementation only offers a limited number of syntactic and semantic queries. New query templates can easily be added without any theoretical constraints. However, the user interface may need to be updated to help users navigate and possibly search through a long list of queries.

[5] Tests carried out on a dual-core MacBook Air from 2013.

In summary, the *Contract Verifier* tool in its current state can handle documents of essentially any size in what concerns the extraction of a formal representation from a natural language text, its post-editing, and the execution of syntactic queries. This however could be improved by adding an extra layer of document hierarchy management to the tool, allowing the task to be segmented into smaller sub-parts. In what concerns semantic queries, the tool can realistically only handle smaller individual contracts containing tens of clauses. This is essentially due to our choice to translate contract models into Uppaal timed automata. An alternative here could be to use SAT solvers, or some other verification technology, to process the kind of semantic queries we perform. In any case, we believe *Contract Verifier* is an important step towards a rich analysis of normative texts, even if the analysis were to be restricted to just syntactic queries.

List of acronyms

ACE	Attempto Controlled English [37]
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
$\mathcal{CL}$	Contract Logic [78]
CLAN	$\mathcal{CL}$ Analyser [32]
CNL	Controlled Natural Language
CODSH	*C-O Diagram* Shorthand [18]
COML	Contract-Oriented XML [20, 18]
CSL	Contract Specification Language [50]
CTD	Contrary-to-Duty reparation
CTP	Contrary-to-Prohibition reparation
DRS	Discourse Representation Structure
DSL	Domain-Specific Language
EDSL	Embedded Domain-Specific Language
GATE	General Architecture for Text Engineering [25]
GF	Grammatical Framework [79]
GPL	GNU General Public License
JIRS	Java Information Retrieval System [81]
LKIF	Legal Knowledge Interchange Format [40]
LTL	Linear Temporal Logic
NER	Named Entity Recognition
NL	Natural Language
NLP	Natural Language Processing

NP	Noun Phrase
NTA	Networks of Timed Automata [13]
OCL	Object Constraint Language
PDL	Propositional Dynamic Logic
PENS	Precision, Expressiveness, Naturalness, Simplicity (CNL classification scheme) [53]
PP	Prepositional Phrase
RACE	ACE Reasoner [35]
RGL	GF Resource Grammar Library [80]
SAT	Satisfied/Satisfiability
SBVR	Semantics of Business Vocabulary and Business Rules [68]
$\mathcal{SCL}$	Simplified Contract Language [42]
SLA	Service-Level Agreement
TA	Timed Automata [2]
TCTL	Timed Computation Tree Logic
TSV	Tab-Separated Values
UI	User Interface
VP	Verb Phrase
XML	Extensible Markup Language

Bibliography

All URLs appearing in this thesis were current at the time of publication.

[1] Abubkr A. Abdelsadiq. "A Toolkit for Model Checking of Electronic Contracts". Ph.D. thesis. United Kingdom: School of Computing Science, Newcastle University, 2013.

[2] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[3] Krasimir Angelov. "Incremental Parsing with Parallel Multiple Context-Free Grammars". In: *Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*. ACL, 2009, pp. 69–76. DOI: 10.3115/1609067.1609074.

[4] Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. "A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language". In: *Logic and Algebraic Programming* 82.5-7 (2013), pp. 216–240. DOI: 10.1016/j.jlap.2013.03.002.

[5] Krasimir Angelov and Aarne Ranta. "Implementing Controlled Languages in GF". In: *International Workshop on Controlled Natural Language (CNL 2010)*. Vol. 5972. Lecture Notes in Computer Science. Springer, 2010, pp. 82–101. DOI: 10.1007/978-3-642-14418-9_6.

[6] 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)*. ACM, 2013, pp. 3–12. DOI: 10.1145/2514601.2514603.

[7] Shaun Azzopardi, Albert Gatt, and Gordon Pace. "Integrating Natural Language and Formal Analysis for Legal Documents". In: *Conference on Language Technologies and Digital Humanities*. Academic Publishing Division of the Faculty of Arts, 2016, pp. 32–35. ISBN: 978-961-237-862-2.

[8] Shaun Azzopardi, Albert Gatt, and Gordon J. Pace. "Reasoning About Partial Contracts". In: *Conference on Legal Knowledge and Information Systems (JURIX 2016)*. IOS Press, 2016, pp. 23–32. DOI: 10.3233/978-1-61499-726-9-23.

[9] Shaun Azzopardi, Gordon J. Pace, Fernando Schapachnik, and Gerardo Schneider. "Contract automata - An operational view of contracts between interactive parties". In: *Artificial Intelligence and Law* 24.3 (2016), pp. 203–243. DOI: 10.1007/s10506-016-9185-2.

[10] Patrick Bahr, Jost Berthold, and Martin Elsman. "Certified Symbolic Management of Financial Multi-party Contracts". In: *International Conference on Functional Programming (ICFP 2015)*. ACM, 2015, pp. 315–327. DOI: 10.1145/2784731.2784747.

[11] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on UPPAAL 4.0*. Denmark: Department of Computer Science, Aalborg University, 2006. URL: http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[12] 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". In: *Artificial Intelligence and Law* 20.3 (2012), pp. 215–319. DOI: 10.1007/s10506-012-9131-x.

[13] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets*. Vol. 3098. Lecture Notes in Computer Science. Springer, 2004, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.

[14] Alexander Boer, Radboud Winkels, and Fabio Vitali. "MetaLex XML and the Legal Knowledge Interchange Format". In: *Computable Models of the Law*. Vol. 4884. Lecture Notes in Computer Science. Springer, 2008, pp. 21–41. DOI: 10.1007/978-3-540-85569-9_2.

[15] Davide Buscaldi, Paolo Rosso, José Manuel Gómez-Soriano, and Emilio Sanchis. "Answering Questions with an n-gram based Passage Retrieval Engine". In: *Intelligent Information Systems* 34.2 (2010), pp. 113–134. DOI: 10.1007/s10844-009-0082-y.

[16] John J. Camilleri. "Analysing Normative Contracts — On the Semantic Gap between Natural and Formal Languages". Licentiate thesis. Sweden: Chalmers University of Technology and University of Gothenburg, 2015.

[17] John J. Camilleri, Normunds Grūzītis, and Gerardo Schneider. "Extracting Formal Models from Normative Texts". In: *International Conference on Applications of Natural Language to Information Systems (NLDB 2016)*. Vol. 9612. Lecture Notes in Computer Science. Springer, 2016, pp. 403–408. DOI: 10.1007/978-3-319-41754-7_40.

[18] John J. Camilleri, Mohammad Reza Haghshenas, and Gerardo Schneider. *A Web-Based Tool for Analysing Normative Documents in English*. 2017. arXiv: 1707.03997 [cs.CL].

[19] John J. Camilleri, Gordon J. Pace, and Michael Rosner. "Controlled Natural Language in a Game for Legal Assistance". In: *Controlled Natural Language*. Vol. 7175. Lecture Notes in Computer Science. Springer, 2012, pp. 137–153. DOI: 10.1007/978-3-642-31175-8_8.

[20] John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. "A CNL for Contract-Oriented Diagrams". In: *International Workshop on Controlled Natural Language (CNL 2014)*. Vol. 8625. Lecture Notes in Computer Science. Springer, 2014, pp. 135–146. DOI: 10.1007/978-3-319-10223-8_13.

[21] John J. Camilleri and Gerardo Schneider. "Modelling and Analysis of Normative Documents". In: *Logical and Algebraic Methods in Programming* 91 (2017), pp. 33–59. DOI: 10.1016/j.jlamp.2017.05.002.

[22] Tin Tin Cheng, Jeffrey Leonard Cua, Mark Davies Tan, Kenneth Gerard Yao, and Rachel Edita Roxas. "Information extraction from legal documents". In: *International Symposium on Natural Language Processing (SNLP 2009)*. IEEE, 2009, pp. 157–162. DOI: 10.1109/snlp.2009.5340925.

[23] Children's Commissioner for England. *Growing Up Digital: A report of the Growing Up Digital Taskforce*. 2017. URL: https://www.childrenscommissioner.gov.uk/wp-content/uploads/2017/06/Growing-Up-Digital-Taskforce-Report-January-2017_0.pdf.

[24] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *International Conference on Functional Programming (ICFP 2000)*. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[25] Hamish Cunningham. "GATE, a General Architecture for Text Engineering". In: *Computers and the Humanities* 36.2 (2002), pp. 223–254. DOI: 10.1023/A:1014348124664.

[26] Haskell B. Curry. "Some logical aspects of grammatical structure". In: *Structure of Language and its Mathematical Aspects: Proceedings of Symposia in Applied Mathematics*. Vol. 12. American Mathematical Society, 1961, pp. 56–68.

[27] Alexandre David, M. Oliver Möller, and Wang Yi. "Formal Verification of UML Statecharts with Real-Time Extensions". In: *International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*. Vol. 2306. Lecture Notes in Computer Science. Springer, 2002, pp. 218–232. DOI: 10.1007/3-540-45923-5_15.

[28] Gregorio Díaz, María-Emilia Cambronero, Enrique Martínez, and Gerardo Schneider. "Specification and Verification of Normative Texts using C-O Diagrams". In: *Transactions on Software Engineering* 40.8 (2014), pp. 795–817. DOI: 10.1109/TSE.2013.54.

[29] Robert van Doesburg and Tom M. van Engers. "Perspectives on the Formal Representation of the Interpretation of Norms". In: *Legal Knowledge and Information Systems (JURIX 2016)*. Vol. 294. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 183–186. DOI: 10.3233/978-1-61499-726-9-183.

[30] 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/.

[31] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. "Automatic Conflict Detection on Contracts". In: *International Conference on Theoretical Aspects of Computing (ICTAC 2009)*. Vol. 5684. Lecture Notes in Computer Science. Springer, 2009, pp. 200–214. DOI: 10.1007/978-3-642-03466-4.

[32] 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)*. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 90–96. DOI: 10.1007/978-3-642-04761-9_8.

[33] Mark D. Flood and Oliver R. Goodenough. "Contract as Automaton: The Computational Representation of Financial Agreements". In: *Office of Financial Research* Working Paper No. 15-04 (2015). DOI: 10.2139/ssrn.2648460.

[34] 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.

[35] Norbert E. Fuchs. "First-Order Reasoning for Attempto Controlled English". In: *International Workshop on Controlled Natural Language (CNL 2010)*. Vol. 7175. Lecture Notes in Computer Science. Springer, 2010, pp. 73–94. DOI: 10.1007/978-3-642-31175-8_5.

[36] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. "Attempto Controlled English for Knowledge Representation". In: *Reasoning Web*. Vol. 5224. Lecture Notes in Computer Science. Springer, 2008, pp. 104–124. DOI: 10.1007/978-3-540-85658-0_3.

[37] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. *Attempto Controlled English (ACE) Language Manual, Version 3.0*. Tech. rep. IFI-2011.0008. Department of Informatics, University of Zurich, 1999.

[38] Debasish Ghosh. "DSL for the Uninitiated". In: *Communications of the ACM* 54.7 (2011), pp. 44–50. DOI: 10.1145/1965724.1965740.

[39] Thomas F. Gordon. "An Overview of the Carneades Argumentation Support System". In: *Dialectics, Dialogue and Argumentation - An Examination of Douglas Walton's Theories of Reasoning*. College Publications, 2010, pp. 145–156. ISBN: 978-1-84890-005-9.

[40] Thomas F. Gordon. *The Legal Knowledge Interchange Format (LKIF)*. ESTRELLA Project Deliverable 4.1. 2008. URL: http://www.estrellaproject.org/doc/Estrella-D4.1.pdf.

[41] Daniel Gorín, Sergio Mera, and Fernando Schapachnik. "A Software Tool for Legal Drafting". In: *Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2011)*. Vol. 68. Electronic Proceedings in Theoretical Computer Science. 2011, pp. 71–86. DOI: 10.4204/EPTCS.68.7.

[42] Runa Gulliksson and John J. Camilleri. "A Domain-Specific Language for Normative Texts with Timing Constraints". In: *International Symposium on Temporal Representation and Reasoning (TIME 2016)*. IEEE, 2016, pp. 60–69. DOI: 10.1109/TIME.2016.14.

[43] Ben Hachey and Claire Grover. "Automatic Legal Text Summarisation: Experiments with Summary Structuring". In: *International Conference on Artificial Intelligence and Law (ICAIL 2005)*. ACM, 2005, pp. 75–84. DOI: 10.1145/1165485.1165498.

[44] 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)*. Vol. 2306. Lecture Notes in Computer Science. Springer, 2002, pp. 233–248. DOI: 10.1007/3-540-45923-5_16.

[45] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *ACM* 40.1 (1993), pp. 143–184. DOI: 10.1145/138027.138060.

[46] Stefan Höfler. "Legislative drafting guidelines: How different are they from controlled language rules for technical writing?" In: *International Workshop on Controlled Natural Language (CNL 2012)*. Vol. 7427. Lecture Notes in Computer Science. Springer, 2012, pp. 138–151. DOI: 10.1007/978-3-642-32612-7_10.

[47] Stefan Höfler and Alexandra Bünzli. "Designing a controlled natural language for the representation of legal norms". In: *International Workshop on Controlled Natural Language (CNL 2010)*. 2010. URL: http://www.zora.uzh.ch/35842/.

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

[49] Tom Hvitved. "Contract Formalisation and Modular Implementation of Domain-Specific Languages". Ph.D. thesis. Denmark: Faculty of Science, University of Copenhagen, 2012.

[50] Tom Hvitved, Felix Klaedtke, and Eugen Zălinescu. "A trace-based model for multiparty contracts". In: *Logic and Algebraic Programming* 81.2 (2012), pp. 72–98. DOI: 10.1016/j.jlap.2011.04.010.

[51] Andrew J. I. Jones and Marek Sergot. "On the Characterisation of Law and Computer Systems: The Normative Systems Perspective". In: *Deontic Logic in Computer Science: Normative System Specification*. John Wiley & Sons, 1993. Chap. 12, pp. 275–307.

[52] Dan Klein and Christopher D. Manning. "Accurate Unlexicalized Parsing". In: *Meeting of the Association for Computational Linguistics (ACL 2003)*. ACL, 2003, pp. 423–430. DOI: 10.3115/1075096.1075150.

[53] Tobias Kuhn. "A Survey and Classification of Controlled Natural Languages". In: *Computational Linguistics* 40.1 (2014), pp. 121–170. DOI: 10.1162/COLI_a_00168.

[54] Tobias Kuhn. "An Evaluation Framework for Controlled Natural Languages". In: *Workshop on Controlled Natural Language (CNL 2009)*. Vol. 5972. Lecture Notes in Computer Science. Springer, 2010, pp. 1–20. DOI: 10.1007/978-3-642-14418-9_1.

[55] Tobias Kuhn. "Controlled English for Knowledge Representation". Ph.D. thesis. Switzerland: Faculty of Economics, Business Administration and Information Technology, University of Zurich, 2010.

[56] Kim G. Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a Nutshell". In: *Software Tools for Technology Transfer* 1.1 (1997), pp. 134–152. DOI: 10.1007/s100090050010.

[57] Luis Llana, María-Emilia Cambronero, and Gregorio Díaz. "The Simulation Relation for Formal E-Contracts". In: *International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2016)*. Vol. 9587. Lecture Notes in Computer Science. Springer, 2016, pp. 490–502. DOI: 10.1007/978-3-662-49192-8_40.

[58] Olivera Marjanovic and Zoran Milosevic. "Towards Formal Modeling of e-Contracts". In: *International Conference on Enterprise Distributed Object Computing (EDOC 2001)*. IEEE, 2001, pp. 59–68. DOI: 10.1109/EDOC.2001.950423.

[59] Simon Marlow. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[60] Enrique Martínez, María-Emilia Cambronero, Gregorio Díaz, and Gerardo Schneider. "A Model for Visual Specification of e-Contracts". In: *International Conference on Services Computing (SCC 2010)*. IEEE, 2010, pp. 1–8. DOI: 10.1109/SCC.2010.32.

[61] John O. McGinnis and Russell G. Pearce. "The Great Disruption: How Machine Intelligence Will Transform the Role of Lawyers in the Delivery of Legal Services". In: *Fordham Law Review* 82.6 (2014), pp. 3041–3066. URL: http://ir.lawnet.fordham.edu/flr/vol82/iss6/16/.

[62] Paul McNamara. "Deontic Logic". In: *Logic and the Modalities in the Twentieth Century*. Vol. 7. Handbook of the History of Logic. North Holland, 2006. Chap. 3, pp. 197–289.

[63] 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)*. Vol. 134. Frontiers in Artificial Intelligence and Applications. IOS Press, 2005, pp. 71–80. ISBN: 978-1-60750-152-7.

[64] J.-J. Ch. Meyer, F.P.M. Dignum, and R.J. Wieringa. *The Paradoxes of Deontic Logic Revisited: A Computer Science Perspective*. Tech. rep. UU-CS-1994-38. Utrecht, the Netherlands: Department of Computer Science, Utrecht University, 1994. URL: http://doc.utwente.nl/76218/.

[65] Roger Mitton. "A partial dictionary of English in computer-usable form". In: *Literary & Linguistic Computing* 1.4 (1986), pp. 214–215.

[66] 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)*. ACM, 2007, pp. 225–230. DOI: 10.1145/1276318.1276362.

[67] 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)*. Vol. 68. Electronic Proceedings in Theoretical Computer Science. 2011, pp. 55–69. DOI: 10.4204/EPTCS.68.6.

[68] Object Management Group (OMG). *Semantics of Business Vocabulary and Business Rules (SBVR)*. Document number: formal/2015-05-07. 2015. URL: http://www.omg.org/spec/SBVR/1.3/PDF.

[69] Gordon J. Pace, Cristian Prisacariu, and Gerardo Schneider. "Model Checking Contracts — A Case Study". In: *Automated Technology for Verification and Analysis (ATVA 2007)*. Vol. 4762. Lecture Notes in Computer Science. Springer, 2007, pp. 82–97. DOI: 10.1007/978-3-540-75596-8_8.

[70] Gordon J. Pace and Michael Rosner. "A Controlled Language for the Specification of Contracts". In: *Workshop on Controlled Natural Language (CNL 2009)*. Vol. 5972. Lecture Notes in Computer Science. Springer, 2010, pp. 226–245. DOI: 10.1007/978-3-642-14418-9_14.

[71] 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)*. Vol. 94. Electronic Proceedings in Theoretical Computer Science. 2012, pp. 21–30. DOI: 10.4204/EPTCS.94.3.

[72] Gordon J. Pace and Gerardo Schneider. "Challenges in the Specification of Full Contracts". In: *Integrated Formal Methods (iFM 2009)*. Vol. 5423. Lecture Notes in Computer Science. Springer, 2009, pp. 292–306. DOI: 10.1007/978-3-642-00255-7_20.

[73] Simon Peyton Jones and Jean-Marc Eber. "How to write a financial contract". In: *The Fun of Programming*. Palgrave Macmillan, 2003, pp. 105–129.

[74] Cristian Prisacariu. "A Dynamic Deontic Logic over Synchronous Actions". Ph.D. thesis. Norway: Department of Informatics, University of Oslo, 2010.

[75] Cristian Prisacariu. *Logics for Terms of Services and their Usefulness for Automation*. Slide presentation. Free Society Conference and Nordic Summit (FSCONS 2013). 2013. URL: https://frab.fscons.org/en/fscons13/public/events/27.

[76] Cristian Prisacariu and Gerardo Schneider. "A Dynamic Deontic Logic for Complex Contracts". In: *Logic and Algebraic Programming* 81.4 (2012), pp. 458–490. DOI: 10.1016/j.jlap.2012.03.003.

[77] 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)*. Vol. 4468. Lecture Notes in Computer Science. Springer, 2007, pp. 174–189. DOI: 10.1007/978-3-540-72952-5_11.

[78] Cristian Prisacariu and Gerardo Schneider. "CL: An Action-based Logic for Reasoning about Contracts". In: *Workshop on Logic, Language, Information and Computation (WOLLIC 2009)*. Vol. 5514. Lecture Notes in Computer Science. Springer, 2009, pp. 335–349. DOI: 10.1007/978-3-642-02261-6_27.

[79] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. Studies in Computational Linguistics. CSLI, 2011. ISBN: 1-57586-626-9.

[80] Aarne Ranta. "The GF Resource Grammar Library". In: *Linguistic Issues in Language Technology* 2.2 (2009). ISSN: 1945-3604.

[81] Paolo Rosso, Santiago Correa, and Davide Buscaldi. "Passage Retrieval in Legal Texts". In: *Logic and Algebraic Programming* 80.3–5 (2011), pp. 139–153. DOI: 10.1016/j.jlap.2011.02.001.

[82] RuleML. *Rule Markup Language Initiative*. 2015. URL: http://wiki.ruleml.org/.

[83] 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)*. ACM, 2001, pp. 78–87. DOI: 10.1145/383535.383544.

[84] Harry Surden. "Computable Contracts". In: *UC Davis Law Review* 46.629 (2012). URL: https://ssrn.com/abstract=2216866.

[85] Nick Szabo. *A Formal Language for Analyzing Contracts*. Preliminary Draft. 2002. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/contractlanguage.html.

[86] Romuald Thion and Daniel Le Métayer. "FLAVOR: A Formal Language for a Posteriori Verification of Legal Rules". In: *International Symposium on Policies for Distributed Systems and Networks (POLICY 2011)*. IEEE, 2011, pp. 1–8. DOI: 10.1109/POLICY.2011.26.

[87] 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)*. Vol. 13. ACL, 2000, pp. 63–70. DOI: 10.3115/1117794.1117802.

[88] Jan van Eijck and Fengkui Ju. "Modelling Legal Relations". In: *Logic and the Foundations of Game and Decision Theory (LOFT 2016)*. 2016. URL: https://ir.cwi.nl/pub/25326.

[89] Adam Z. Wyner. "From the Language of Legislation to Executable Logic Programs". In: *Logic in the Theory and Practice of Lawmaking*. Vol. 2. Legisprudence Library. Springer, 2015, pp. 409–434. DOI: 10.1007/978-3-319-19575-9_15.

[90] Adam Z. Wyner. "Violations and Fulfillments in the Formal Representation of Contracts". Ph.D. thesis. United Kingdom: Department of Computer Science, King's College London, 2008.

[91] Adam Z. Wyner, Johan Bos, Valerio Basile, and Paulo Quaresma. "An Empirical Approach to the Semantic Representation of Laws". In: *Legal Knowledge and Information Systems (JURIX 2012)*. Vol. 250. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 177–180. DOI: 10.3233/978-1-61499-167-0-177.

[92] Adam Z. Wyner, Tom M. van Engers, and Kiavash Bahreini. "From Policy-Making Statements to First-Order Logic". In: *Electronic Government and the Information Systems Perspective (EGOVIS 2010)*. Vol. 6267. Lecture Notes in Computer Science. Springer, 2010, pp. 47–61. DOI: 10.1007/978-3-642-15172-9_5.

[93] 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)*. Vol. 5972. Lecture Notes in Computer Science. Springer, 2010, pp. 281–289. DOI: 10.1007/978-3-642-14418-9_17.

[94] Adam Wyner and Wim Peters. "On rule extraction from regulations". In: *Conference on Legal Knowledge and Information Systems (JURIX 2011)*. Vol. 235. Frontiers in Artificial Intelligence and Applications. IOS Press, 2011, pp. 113–122. DOI: 10.3233/978-1-60750-981-3-113.