Modelling and Analysis of Normative Documents

John J. Camilleri, Gerardo Schneider

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

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.

Keywords: normative documents, contract analysis, timed automata, uppaal

1. Introduction

We frequently encounter *normative documents* (or *contracts*) when subscribing to internet services and using software. These come in the forms 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 1.

Email addresses: john.j.camilleri@cse.gu.se (John J. Camilleri), gerardo@cse.gu.se (Gerardo Schneider)

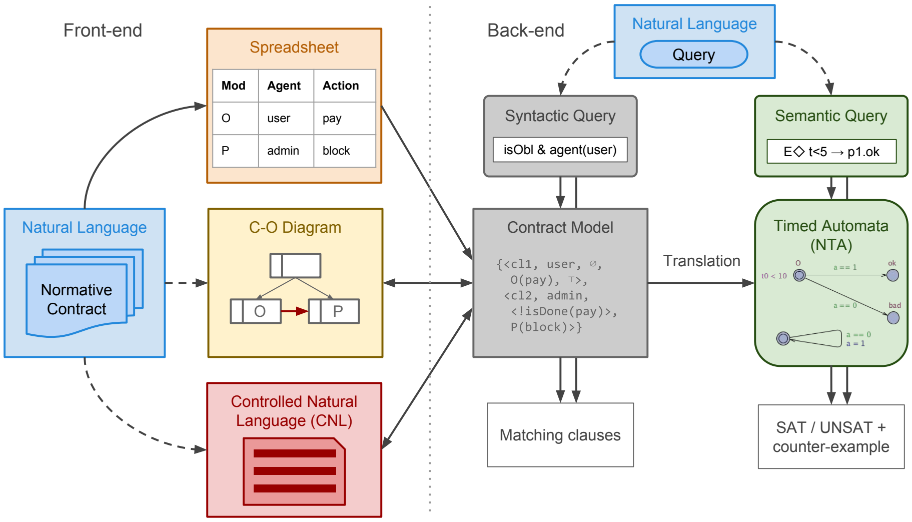


Figure 1: Overview of our contract processing framework, separating the front-end concerns of model-building from the back-end tasks related to analysis. Dashed arrows represent manual interaction, while solid ones represent automatic steps.

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 [1], 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 [3] 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 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 layed out as follows. In Section 2 we first present an extended definition of the *C-O Diagram* formalism, introducing an updated syntax and a novel trace semantics. Section 3 then describes our own translation function from the extended *C-O Diagram* formalism into Uppaal timed automata, which is more modular and fixes a number of issues with respect to the previous translation given in [4]. 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 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 5. Finally, we conclude with a comparison of some related work in Section 6 and a final discussion in Section 7.

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

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

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

2. *C-O Diagram* Formalism

C-O Diagrams were introduced by Martínez et al. [1] 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 2).

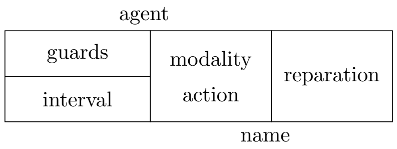


Figure 2: The various components of a single *C-O Diagram* box.

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).

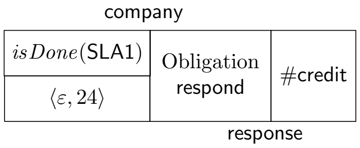
Each box also has an *agent* indicating the performer of the action, and a unique *name* for referencing purposes. Figure 3 shows a completed example of such a box. Boxes can be expanded by using three kinds of refinement: *conjunction*, *choice*, and *sequence*, which allow complex clauses to be built 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 4.

2.1. Formal Syntax

Figure 5 gives the formal grammar of our modelling language. It has been extended from its original definition given in [4]. These extensions are explained at the end of this section.

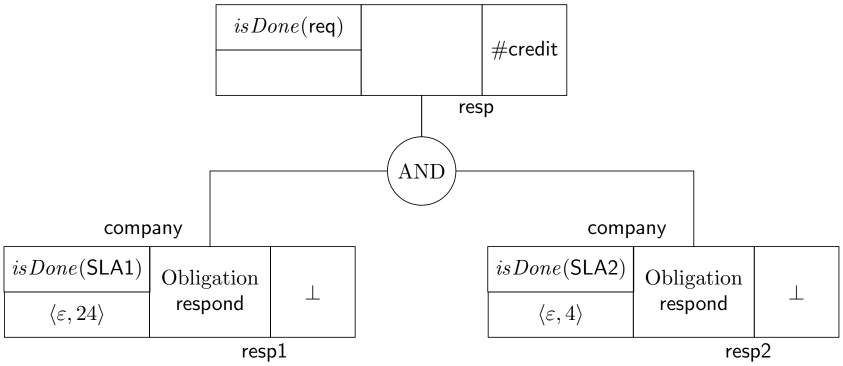
A **contract** specification K is a *forest* of top-level clause trees, each of which is tagged as either *Main* (instantiated when the contract is executed) or *Aux* (instantiated only when referenced). A **clause** C is primarily a modal statement, expressing the **obligation** $O(\cdot)$, **permission** $P(\cdot)$, 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,

[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.



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

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



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 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.

and $\perp$ 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 $\phi(name)$. Similarly, given a finite set of 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

[2] The reason for the separation between guards and intervals is discussed on page 7.

$$
\begin{align}
K &:= \big\{ \langle C, \mathit{Type} \rangle^{+} \big\} \text{ where } \mathit{Type} \in \{\mathit{Main}, \mathit{Aux}\} \tag{1}\\
C &:= \langle \mathit{name}, \mathit{agent}, \mathit{Conditions}, O(C_2), R \rangle \tag{2}\\
&\;\big|\; \langle \mathit{name}, \mathit{agent}, \mathit{Conditions}, P(C_2) \rangle \tag{3}\\
&\;\big|\; \langle \mathit{name}, \mathit{agent}, \mathit{Conditions}, F(C_2), R \rangle \tag{4}\\
&\;\big|\; \langle \mathit{name}, \mathit{Conditions}, C_1, R \rangle \tag{5}\\
&\;\big|\; \mathit{Ref} \tag{6}\\
C_1 &:= C\ (\mathit{Seq}\ C)^{+} \;\big|\; C\ (\mathit{And}\ C)^{+} \;\big|\; C\ (\mathit{Or}\ C)^{+} \tag{7}\\
C_2 &:= \mathit{action} \;\big|\; C_3\ (\mathit{Seq}\ C_3)^{+} \;\big|\; C_3\ (\mathit{And}\ C_3)^{+} \;\big|\; C_3\ (\mathit{Or}\ C_3)^{+} \tag{8}\\
C_3 &:= \langle \mathit{name}, C_2 \rangle \tag{9}\\
R &:= \mathit{Ref} \;\big|\; \top \;\big|\; \bot \tag{10}\\
\mathit{Ref} &:= \#\mathit{name} \tag{11}\\
\mathit{Conditions} &:= \big\langle \mathit{Guard}, \mathit{Interval} \big\rangle \tag{12}\\
\mathit{Guard} &:= \{\mathit{Constraint}^{*}\} \tag{13}\\
\mathit{Constraint} &:= \phi(\mathit{name}) \;\big|\; v\ [-w] \sim z \;\big|\; c\ [-d] \sim t \tag{14}\\
&\quad\ \text{where } \phi \in \{\mathit{isDone}, \mathit{isComplete}, \mathit{isSat}, \mathit{isVio}, \mathit{isSkip}\} \notag\\
&\qquad\qquad v, w \in \mathcal{V},\ \ c, d \in \mathcal{C},\ \ \sim\, \in \{<, =, >\},\ \ z : \mathbb{Z},\ \ t : \mathbb{T} \notag\\
\mathit{Interval} &:= \langle \varepsilon \,|\, t,\ \varepsilon \,|\, t \rangle \text{ where } t : \mathbb{T} \tag{15}
\end{align}
$$

Figure 5: Extended version of the *C-O Diagram* syntax [4] for contracts, where $\mathit{name} \in \mathcal{N}$, $\mathit{agent} \in \mathcal{A}$ and $\mathit{action} \in \Sigma$. $\mathbb{T}$ represents the type of time stamps. Differences from the original include the top-level contract type K indicating *Main*/*Aux* clauses (1), the addition of cross-references (6), top/bottom as reparations (10), the distinction between guards and intervals (12), and the inclusion of predicates as constraints (14). In addition, our version of *C-O Diagrams* does not support repetition.

the relative difference of two clocks ($c - d \sim t$).[3] By convention, we assume that for every $\mathit{name} \in \mathcal{N}$ there is a clock in $\mathcal{C}$ with identifier t_{name}. 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.

2.2. Extensions

The syntax presented here adds a number of extensions to the previous definition of *C-O Diagrams* given in [4]. These extensions were mainly introduced as a result of implementing the system as a runnable tool (see Section 3.2) and to help the modeller by making common constructs easily expressible without requiring extra encoding. The extensions include:

(i) *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:

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

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 be appear multiple times in different parts of the same model.

(ii) *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:

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

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.

(iii) *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:

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

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.

(iv) *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).

2.3. Trace Semantics

Previous work defines the semantics of *C-O Diagrams* via translation to timed automata [4]. 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 3) may be proven by comparison with the semantics of Uppaal automata, also defined in terms of traces [5].

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 $\forall i, j \cdot (0 \leq i < n) \wedge (i < j < n) \Longrightarrow time(\sigma(i)) \leq time(\sigma(j))$. We assume all our traces are well-formed.

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

Definition 2. *We write $\sigma \vDash C$ to mean that* trace σ respects contract C *and $\sigma \nvDash C$ 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 $\vDash_t^c$). It is also extended to actions, where it is further parametrised by an agent (written $\vDash_t^{c,a}$). 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 6 on page 12.

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 (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.

$$\begin{aligned} Env = \mathbb{T} \to \langle \mathcal{V} \mapsto \mathbb{Z},\, \mathcal{C} \mapsto \mathbb{T},\, \mathcal{B} \mapsto \mathit{Boolean} \rangle \quad (16) \\ get_v \;:\; Env \to \mathbb{T} \to \mathcal{V} \to \mathbb{Z} \quad (17) \\ get_c \;:\; Env \to \mathbb{T} \to \mathcal{C} \to \mathbb{T} \quad (18) \\ get_b \;:\; Env \to \mathbb{T} \to \mathcal{B} \to \mathit{Boolean} \quad (19) \\ set_v \;:\; Env \to \mathbb{T} \to \mathcal{V} \to \mathbb{Z} \to Env \quad (20) \\ set_b \;:\; Env \to \mathbb{T} \to \mathcal{B} \to \mathbb{Z} \to Env \quad (21) \\ reset_c \;:\; Env \to \mathbb{T} \to \mathcal{C} \to Env \quad (22) \end{aligned}$$

The environment can be queried via the *get* functions (17–19). Integer and Boolean variables can be updated using the *set* functions (20 and 21), while clocks can be reset to 0 with $reset_c$ (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 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 *name* is enabled, clock t_{name} is reset;
(ii) when a clause *name* is satisfied (including via reparation), Sat_{name} is set to *true* and clock t_{name} is reset;
(iii) when a clause *name* is violated, Vio_{name} is set to *true*;
(iv) when the guard for clause *name* expires, $Skip_{name}$ 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 $Done_{name}$ is also set to *true* for the parent clause *name*.

The *lookup* function (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 $\bot$ if none is found.

$$lookup : K \to \mathcal{N} \to C \quad (23)$$

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

$$\begin{aligned} &\tau : \{Constraint^*\} \rightarrow Name \rightarrow Interval \rightarrow \{Constraint^*\} \\ &\tau(c, n, \langle l, u\rangle) = c \cup \{t_n > l, t_n < u\} \end{aligned} \tag{24}$$

$$\begin{aligned} &\tau' : \{Constraint^*\} \rightarrow Name \rightarrow Interval \rightarrow \{Constraint^*\} \\ &\tau'(c, n, \langle \_, u\rangle) = c \cup \{t_n \geq u\} \end{aligned} \tag{25}$$

Checking constraints from both guards and intervals is done with the *check* function (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 (27).

$$\begin{aligned} check &: \{Constraint^*\} \rightarrow \mathbb{T} \rightarrow Boolean \\ check\big(\{c_1, \ldots, c_n\}, t\big) &= \begin{cases} true & \text{if } n = 0 \\ \bigwedge_{1 \leq j \leq n} eval(c_j, t) & \text{otherwise} \end{cases} \end{aligned} \tag{26}$$

$$\begin{aligned} eval &: Constraint \rightarrow \mathbb{T} \rightarrow Boolean \\ eval\big(x \sim n, t\big) &= get(\Gamma, t, x) \sim n \\ eval\big(x - y \sim n, t\big) &= get(\Gamma, t, x) - get(\Gamma, t, y) \sim n \\ eval\big(is\phi(name), t\big) &= \begin{cases} eval(isSat(name), t) \vee eval(isSkip(name), t) \\ \qquad\qquad\qquad\qquad \text{if } \phi = Complete \\ get_b(\Gamma, t, \phi_{name}) \\ \qquad\qquad\qquad \text{if } \phi \in \{Done, Sat, Vio, Skip\} \end{cases} \end{aligned} \tag{27}$$

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 (28), which is a partial function as the guard may in fact never be satisfied.

$$\begin{aligned} &lst : Guard \rightarrow \mathbb{T} \rightarrow \mathbb{T} \\ &lst(g, t) = \begin{cases} t' & \text{if } \exists t' : \mathbb{T} \cdot t' = \min\limits_{\forall u \geq t} \big[check(g, u) = true\big] \\ undefined & \text{otherwise} \end{cases} \end{aligned} \tag{28}$$

Rules. The rules defining the *respects* relation are given in Figure 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 (34 and 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 A.5.

Contract

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

Deontic operators

$$\sigma \models^{c}_{t_0} \langle n, a, \langle g, i\rangle, O(C_2), R\rangle \tag{30}$$
$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \models^{\tau(c,n,i),a}_{t} C_2 \text{ or } \sigma \models^{\tau'(c,n,i)}_{t} R\big)$$
$$\sigma \models^{c}_{t_0} \langle n, a, \langle g, i\rangle, P(C_2)\, \rangle \tag{31}$$
$$\sigma \models^{c}_{t_0} \langle n, a, \langle g, i\rangle, F(C_2), R\rangle \tag{32}$$
$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \models^{\tau(c,n,i),a}_{t} C_2 \text{ implies } \sigma \models^{c}_{t} R\big)$$

Refinement

$$\sigma \models^{c}_{t_0} \langle n, \langle g, i\rangle, C_1, R\rangle \tag{33}$$
$$\text{iff } \big(\exists t : \mathbb{T} \cdot t = \mathit{lst}(g, t_0)\big) \text{ implies } \big(\sigma \models^{\tau(c,n,i)}_{t} C_1 \text{ or } \sigma \models^{\tau'(c,n,i)}_{t} R\big)$$
$$\sigma \models^{c}_{t_0} C' \ \mathit{Seq}\ C'' \tag{34}$$
$$\text{iff } \exists j : \mathbb{N} \cdot \big(0 \le j \le \mathit{length}(\sigma) \wedge \sigma(..j) \models^{c}_{t_0} C' \wedge \sigma(j..) \models^{c}_{t_0} C''\big)$$
$$\sigma \models^{c}_{t_0} C' \ \mathit{And}\ C'' \text{ iff } \sigma \models^{c}_{t_0} C' \text{ and } \sigma \models^{c}_{t_0} C'' \tag{35}$$
$$\sigma \models^{c}_{t_0} C' \ \mathit{Or}\ C'' \text{ iff either } \sigma \models^{c}_{t_0} C' \text{ or } \sigma \models^{c}_{t_0} C'' \tag{36}$$

Actions

$$\sigma \models^{c,a}_{t_0} x \text{ iff } \exists j : \mathbb{N} \cdot \big(0 \le j < \mathit{length}(\sigma)\ \wedge \tag{37}$$
$$\langle a, x, t\rangle = \sigma(j) \wedge t_0 \le t \wedge \mathit{check}(c, t)\big)$$
$$\sigma \models^{c,a}_{t_0} C'_3 \ \mathit{Seq}\ C''_3 \tag{38}$$
$$\text{iff } \exists j : \mathbb{N} \cdot \big(0 < j < \mathit{length}(\sigma) \wedge \sigma(..j) \models^{c,a}_{t_0} C'_3 \wedge \sigma(j..) \models^{c,a}_{t_0} C''_3\big)$$
$$\sigma \models^{c,a}_{t_0} C'_3 \ \mathit{And}\ C''_3 \text{ iff } \sigma \models^{c,a}_{t_0} (C'_3 \ \mathit{Seq}\ C''_3) \ \mathit{Or}\ (C''_3 \ \mathit{Seq}\ C'_3) \tag{39}$$
$$\sigma \models^{c,a}_{t_0} C'_3 \ \mathit{Or}\ C''_3 \text{ iff either } \sigma \models^{c,a}_{t_0} C'_3 \text{ or } \sigma \models^{c,a}_{t_0} C''_3 \tag{40}$$
$$\sigma \models^{c,a}_{t_0} \langle n, C_2\rangle \text{ iff } \sigma \models^{c,a}_{t_0} C_2 \tag{41}$$

Reparation

$$\sigma \models^{c}_{t_0} \top \tag{42}$$
$$\sigma \not\models^{c}_{t_0} \bot \tag{43}$$
$$\sigma \models^{c}_{t_0} \#\mathit{name} \text{ iff } \sigma \models^{c}_{t_0} \mathit{lookup}(\mathit{name}) \tag{44}$$

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

2.4. Example

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

$$\begin{aligned} C = &\big\{\big\langle\langle \mathsf{resp}, \langle \mathit{isDone}(\mathsf{req}), \langle\varepsilon,\varepsilon\rangle\rangle, C' \mathit{And}\ C'', \#\mathsf{credit}\rangle, \mathit{Main}\big\rangle\big\} \\ &\text{where} \\ &C' \ = \langle \mathsf{resp1}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{SLA1}), \langle\varepsilon, 24\rangle\rangle, O(\mathsf{respond}), \bot\rangle \\ &C'' = \langle \mathsf{resp2}, \mathsf{company}, \langle \mathit{isDone}(\mathsf{SLA2}), \langle\varepsilon, 4\rangle\rangle, O(\mathsf{respond}), \bot\rangle \end{aligned}$$

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

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

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

$$\begin{aligned} \Gamma_{\mathcal{B}}(0..2) &= \{\mathit{Done}_{SLA1} \mapsto \mathit{false},\ \mathit{Done}_{SLA2} \mapsto \mathit{true},\ \mathit{Done}_{req} \mapsto \mathit{false}\} \\ \Gamma_{\mathcal{B}}(3) &= \{\mathit{Done}_{SLA1} \mapsto \mathit{false},\ \mathit{Done}_{SLA2} \mapsto \mathit{true},\ \mathit{Done}_{req} \mapsto \mathit{true}\} \end{aligned}$$

Note that at time stamp 3, the value of Done_{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 (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:

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

By rule (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_{req} becomes *true*. Thus we have either that the main clause is satisfied:

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

or that the clause is repaired like so:

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

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

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

and

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

We then apply the rule for obligation clauses (30) to both of these cases. The guard in the first case (line 48), namely $\mathit{isDone}(\mathsf{SLA1})$, will never be true in the environment Γ. Thus $\mathit{lst}(\mathit{isDone}(\mathsf{SLA1}), 3)$ is 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 49), $\mathit{lst}(\mathit{isDone}(\mathsf{SLA2}), 3) = 3$, meaning that we now need to either satisfy the inner action:

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

or the reparation of the clause:

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

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

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

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 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.

3. Translation to Timed Automata

3.1. Timed Automata

In order to enable property-based analysis on contract models, Díaz et al. [4] 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

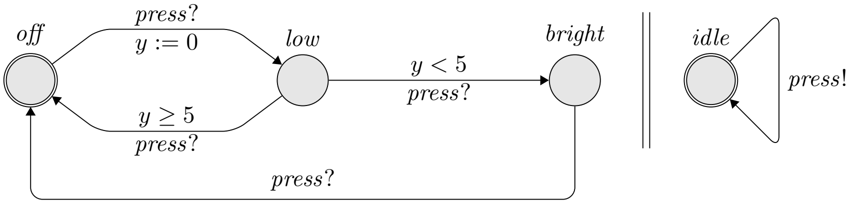


Figure 7: Timed automata modelling a lamp (left) and a user (right), where y is a clock and *press* is a channel. Double circles indicate initial locations. Taken from [6].

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 set of channels which allow automata to synchronise.

Figure 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 [7] 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 [6], 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 [4] is described in terms of abstract NTA, followed by

$$C = \langle name, agent, \langle guards, interval\rangle, O(C_2), R\rangle$$

$$\text{where } guards = g_{low} \wedge g_{upp} \wedge g_{vars} \text{ and } interval = \langle i_{low}, i_{upp}\rangle$$

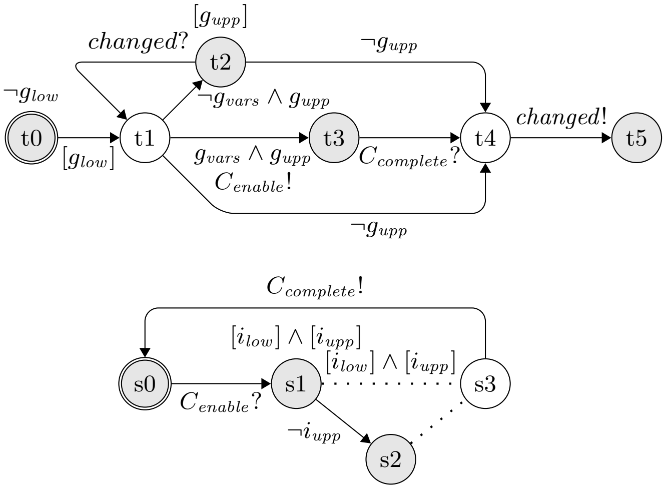


Figure 8: Translation of an obligation clause (top) into two timed automata: the *thread* (middle) and *main* automaton (bottom). The dotted lines $s1 \cdots s3$ and $s2 \cdots s3$ are replaced with the translations of the complex action C_2, and of the reparation R, respectively. Square brackets indicate 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.

explanations of how 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.

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 Appendix A.

Figure 8 shows a generic obligation clause together with the UPPAAL automata produced from its translation. Informally, this box is interpreted as follows: when *guards* become true, *agent* is obliged to do action C_2 within the

$$C = \langle name, \langle guards, \langle i_{low}, i_{upp}\rangle\rangle, C'\ Seq\ C'', R\rangle$$

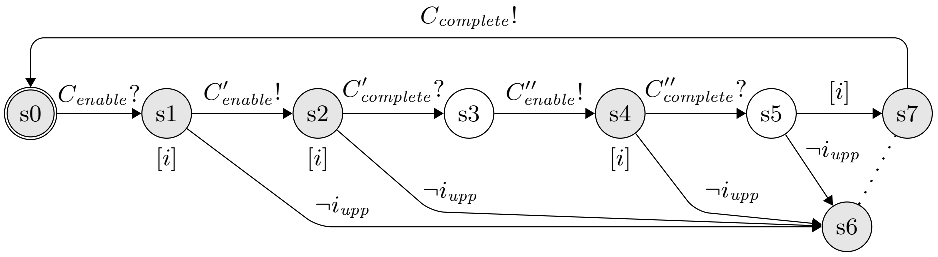


Figure 9: Main automaton from the translation of a *Seq* refinement (thread automaton not shown here). Inner clauses C' and C'' are translated separately (not shown here) and controlled via their respective *enable* and *complete* channels. The automaton for reparation R is inserted between locations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

time frame described by *interval*. If *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 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 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 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 37 in Appendix A.5 (page 42).

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 5.

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 2.3. We approach this by relating our trace semantics for *C-O Diagrams* with that of Uppaal.

David et al. [5] 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 Appendix A.2.

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 *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 10, we state the following theorem relating our trace semantics for *C-O Diagrams* with Uppaal model traces:

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

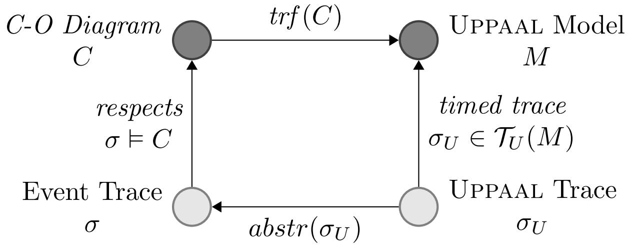


Figure 10: Relations between *C-O Diagram* and UPPAAL model and trace representations.

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

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

Proof Sketch. The proof is performed by structural induction over the *C-O Diagram* syntax (see Figure 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. [5], 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 A. □

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 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 5.

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

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 $\top$
$\mathit{agentOf}(a)$	agent a is responsible for clause
$\mathit{hasAction}(x)$	action x appears in the body of clause

Table 2: Predicates used in syntactic analysis.

for a clause. Table 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 $\mathcal{Q}$ (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.

$$\mathcal{Q} : (C \to \mathit{Boolean}) \to K \to \mathcal{P}(C) \tag{53}$$
$$\mathcal{Q}\big(\psi, \big\{\langle C^1, T^1\rangle, \ldots, \langle C^n, T^n\rangle\big\}\big) = \big\{C^i \bigm| 1 \leq i \leq n,\ \psi \text{ holds w.r.t. } C^i\big\}$$

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 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 [6]. 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 5, rule 14) are also implemented as functions in the 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. $\mathit{isComplete}(\mathsf{clause})$ or $\mathit{isDone}(\mathsf{agent.action})$. 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. $\mathit{Clocks}[\mathsf{clause}]$ or $\mathit{Clocks}[\mathsf{agent.action}]$. 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.

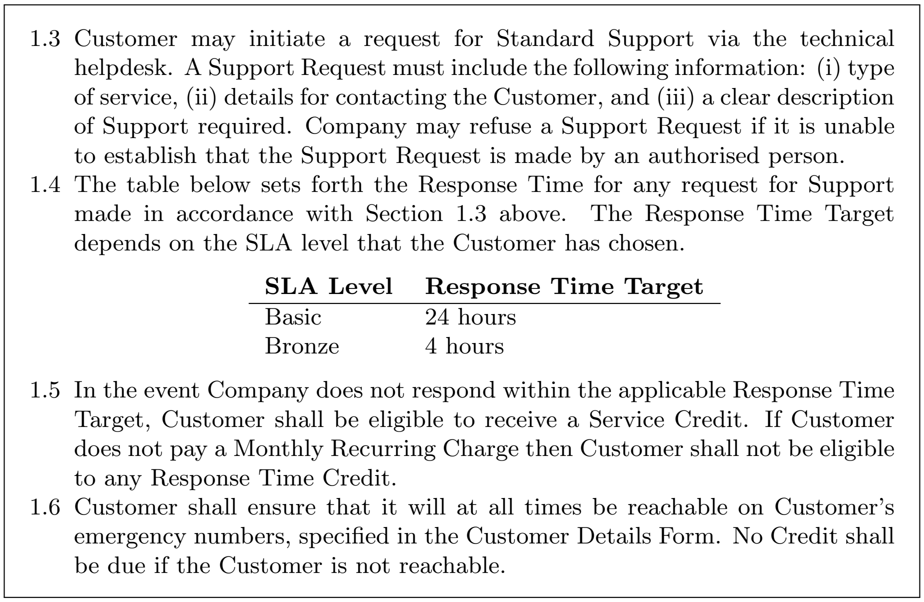
5. Case Study

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

[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.

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

full agreement, which we have abridged into 4 clauses (see Figure 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 B.

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.

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 ([3], [8]) for facilitating the modelling process. The construction of the contract

$$
\begin{aligned}
&C = \Big\{\\
&\quad \Big\langle \langle \mathsf{request}, \epsilon, \#\mathsf{req\_type}\ \mathit{Seq}\ \#\mathsf{req\_info}\ \mathit{Seq}\ \#\mathsf{resp}, \top\rangle, \mathit{Main}\Big\rangle,\\
&\quad \Big\langle \langle \mathsf{req\_type}, \mathsf{customer}, \epsilon, P(\mathsf{standard\ support})\rangle, \mathit{Aux}\Big\rangle,\\
&\quad \Big\langle \langle \mathsf{req\_info}, \mathsf{customer}, \epsilon, O(C_2), \top\rangle, \mathit{Aux}\Big\rangle,\\
&\quad \Big\langle \langle \mathsf{cust\_auth}, \mathsf{customer}, \epsilon, O(\mathsf{prove\ authorisation}), \top\rangle, \mathit{Main}\Big\rangle,\\
&\quad \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,\\
&\quad \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,\\
&\quad \Big\langle \langle \mathsf{resp}, \epsilon, \#\mathsf{resp1}\ \mathit{And}\ \#\mathsf{resp2}, \top\rangle, \mathit{Aux}\Big\rangle,\\
&\quad \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,\\
&\quad \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,\\
&\quad \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,\\
&\quad \Big\langle \langle \mathsf{reach}, \mathsf{customer}, \epsilon, O(\mathsf{be\ reachable}), \top\rangle, \mathit{Main}\Big\rangle\\
&\Big\}\\
&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\ desc.}\rangle
\end{aligned}
$$

Figure 12: Contract model for the normative text shown in Figure 11.

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 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 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.

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:

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

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 $\mathsf{chooseSLA}$, 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.

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 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:

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

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 $\mathsf{req\_info}$ 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

[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.

model. If we change the reparation for the clause to $\perp$, 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:

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

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

$$\begin{aligned}&\forall \Box\, \mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{resp1}) \land \mathit{isDone}(\mathsf{reach})\\ &\quad \land \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] > 24\\ &\quad \implies \mathit{isDone}(\mathsf{credit})\end{aligned} \tag{56}$$

$$\begin{aligned}&\forall \Box\, \mathit{isComplete}(\mathsf{request}) \land \mathit{isDone}(\mathsf{resp1})\\ &\quad \land \mathit{Clocks}[\mathsf{resp1}] - \mathit{Clocks}[\mathsf{company.respond}] < 24\\ &\quad \implies \neg \mathit{isDone}(\mathsf{credit})\end{aligned} \tag{57}$$

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 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 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 7.

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

Table 3: Resources recquired for verifying the properties in Section 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.

6. Related Work

C-O Diagrams were introduced by Martínez et al. in [1], and further refined in [4]. 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 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 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 [9] where a trace semantics is defined for the contract language $\mathcal{CL}$ [10]. 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 [9], but must search through the entire trace looking for events which satisfy the given conditions.

Llana et al. [11] 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. [3], 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 [12] is a similar framework for the analysis of contracts, based on the contract logic $\mathcal{CL}$ [10], which allows for the detection of contradictory clauses in normative texts using the CLAN tool [13]. 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 [14] 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 [15] 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 [16] 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 particularly, 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 [17] 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 [18], 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)* [19] 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.

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 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 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 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 [8], as well as for building contract models graphically and using controlled natural language [3].

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

The authors wish to thank the Swedish Research Council for financial support under grant number 2012-5746. We are also very grateful to Gabriele Paganelli and Filippo Del Tedesco for their contributions to earlier versions of this work.

References

[1] E. Martínez, E. Cambronero, G. Díaz, G. Schneider, A Model for Visual Specification of e-Contracts, in: 5th IEEE International Conference on Services Computing (SCC 2010), IEEE Computer Society, 2010, pp. 1–8.

[2] R. Alur, D. L. Dill, A Theory of Timed Automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[3] J. J. Camilleri, G. Paganelli, G. Schneider, A CNL for Contract-Oriented Diagrams, in: 4th International Workshop on Controlled Natural Language (CNL 2014), Vol. 8625 of LNCS, Springer, 2014, pp. 135–146.

[4] G. Díaz, M. E. Cambronero, E. Martínez, G. Schneider, Specification and Verification of Normative Texts using C-O Diagrams, IEEE Transactions on Software Engineering 40 (8) (2014) 795–817.

[5] A. David, M. O. Möller, W. Yi, Verification of UML Statechart with Real-time Extensions, Tech. rep., Department of Information Technology, Uppsala University, Sweden (2003).

[6] G. Behrmann, A. David, K. G. Larsen, A Tutorial on UPPAAL 4.0, Tech. rep., Department of Computer Science, Aalborg University (2006).

[7] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a Nutshell, Software Tools for Technology Transfer 1 (1-2) (1997) 134–152.

[8] J. J. Camilleri, N. Grūzītis, G. Schneider, Extracting Formal Models from Normative Texts, in: 21st International Conference on Applications of Natural Language to Information Systems (NLDB 2016), Springer, 2016, pp. 403–408.

[9] S. Fenech, G. J. Pace, G. Schneider, Automatic Conflict Detection on Contracts, in: 6th International Colloquium on Theoretical Aspects of Computing (ICTAC 2009), Vol. 5684 of LNCS, Springer, 2009, pp. 200–214.

[10] C. Prisacariu, G. Schneider, CL: An Action-based Logic for Reasoning about Contracts, in: 16th Workshop on Logic, Language, Information and Computation (WOLLIC 2009), Vol. 5514 of LNCS, Springer, 2009, pp. 335–349.

[11] L. Llana, M. E. Cambronero, G. Díaz, The Simulation Relation for Formal E-Contracts, in: 42nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2016), Vol. 9587 of LNCS, Springer, 2016, pp. 490–502.

[12] K. Angelov, J. J. Camilleri, G. Schneider, A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language, Language and Algebraic Programming 82 (5-7) (2013) 216–240.

[13] S. Fenech, G. J. Pace, G. Schneider, CLAN: A Tool for Contract Analysis and Conflict Discovery, in: 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009), Vol. 5799 of LNCS, Springer, 2009, pp. 90–96.

[14] G. J. Pace, F. Schapachnik, Contracts for Interacting Two-Party Systems, in: 6th Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2012), Vol. 94 of EPTCS, 2012, pp. 21–30.

[15] O. Marjanovic, Z. Milosevic, Towards Formal Modeling of e-Contracts, in: 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2001), IEEE Computer Society, Washington, DC, USA, 2001, pp. 59–68.

[16] A. Z. Wyner, Violations and Fulfillments in the Formal Representation of Contracts, Ph.D. thesis, Department of Computer Science, King's College London (2008).

[17] T. Athan, H. Boley, G. Governatori, M. Palmirani, A. Paschke, A. Wyner, OASIS LegalRuleML, in: 14th International Conference on Artificial Intelligence and Law (ICAIL 2013), 2013, pp. 3–12.

[18] A. Boer, R. Winkels, F. Vitali, MetaLex XML and the Legal Knowledge Interchange Format, in: Computable Models of the Law, LNAI, Springer, 2008, pp. 21–41.

[19] Object Management Group (OMG), Semantics of Business Vocabulary and Business Rules (SBVR), Tech. Rep. formal/2015-05-07 (2015).
URL http://www.omg.org/spec/SBVR/1.3/PDF

Appendix A. Translation to NTA: Proof of Correctness

Appendix 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 2.3. We do this by structural induction over the syntax in Figure 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.

Appendix A.2. UPPAAL Trace Semantics

David et al. [5] 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*

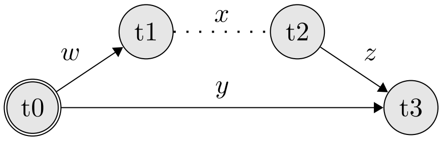
1. *all locations in the initial configuration are the initial locations for their respective processes,*
2. *all clocks in the initial configuration evaluate to 0,*
3. *if the sequence is finite, then at the last configuration no further steps are enabled (system is maximally extended/deadlocked),*
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.*

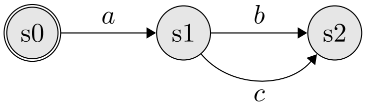
Appendix A.3. Notes and Notation

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:

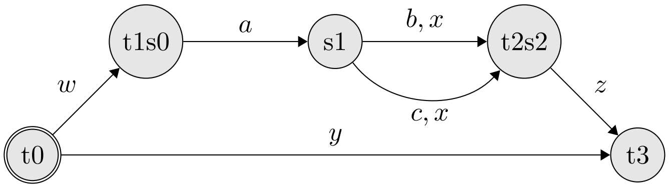
1. Initial locations are drawn with a double border.
2. Committed locations are shown in white.
3. A guard is split up into:
   (a) lower-bound time constraints g_{low} (i.e. using $>$)
   (b) upper-bound time constraints g_{upp} (i.e. using $<$)
   (c) non-temporal (variable) constraints g_{vars}
4. 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*(*name*), *vio*(*name*), *done*(*name*), *sat*(*name*), and *skip*(*name*) are abbreviated to r, v, d, s, sk respectively, where *name* is the name of the 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:





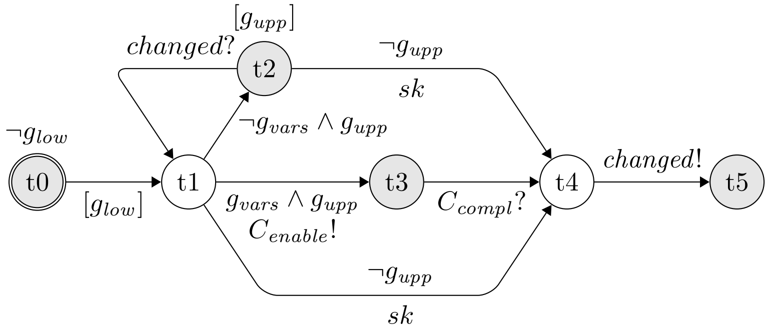
As the t automaton contains a dotted line $t1 \cdots t2$, the entire s automaton could be inserted between these two locations, resulting in the following:



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).

Appendix A.4. Thread Automaton

All top level clauses (cases 30–36, Figure 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.

Appendix A.5. Case Analysis

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

Case 29: Contract

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

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 *name* from its initial state to one in which no further steps are possible, and in which $isComplete(name)$ is *true*.

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

Case 30: Obligation

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

Event traces. We consider the following cases:

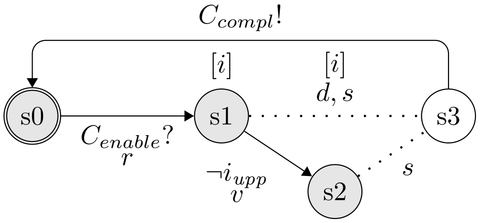
(a) Guards g are never *true* ($\nexists t$): the obligation is not enacted and thus trivially respected.

(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 Appendix A.4) and one main automaton as follows, where $s1 \cdots s3$ 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.

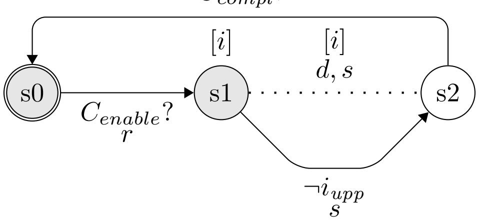
Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then either C_2 is respected within the interval i, or R is respected after i has expired.

Case 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 Appendix A.4) and one main automaton as follows, where $s1 \cdots s2$ 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 32: Prohibition

$$\begin{aligned} &\sigma \vDash^{c}_{t_0} \langle n, a, \langle g, i \rangle, F(C_2), R \rangle \\ &\qquad \text{iff } \big(\exists t : \mathbb{T} \cdot t = lst(g, t_0)\big) \text{ implies } \big(\sigma \vDash^{\tau(c,n,i),a}_{t} C_2 \text{ implies } \sigma \vDash^{c}_{t} R\big) \end{aligned}$$

Event traces. We consider the following cases:

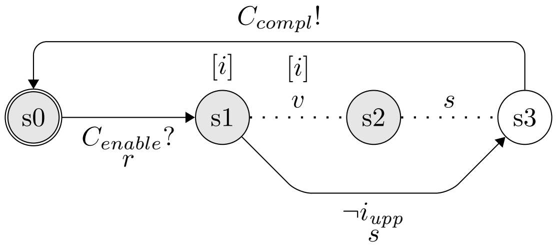
(a) Guards g are never *true* ($\nexists t$): the prohibition is not enacted and thus trivially respected.

(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 Appendix A.4) and one main automaton as follows, where $s1 \cdots s2$ 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 33: Refinement

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

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 35–36 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 Appendix A.4) and one main automaton as described in cases 35–36 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.

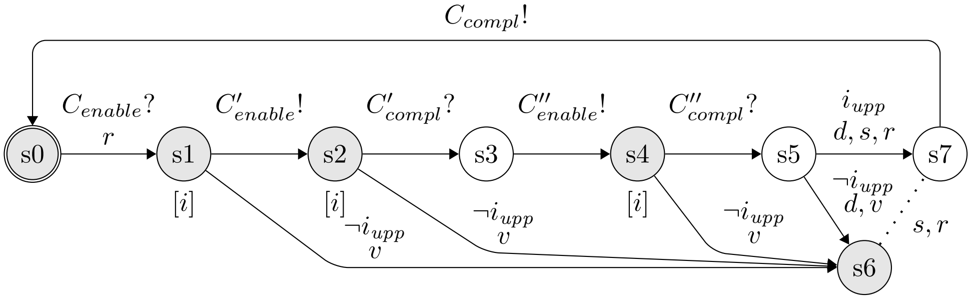
Argument. The case distinctions above map directly to each other, such that both sets of traces require that if the clause is enacted, then either C_2 is respected within the interval i, or R is respected after i has expired.

Case 34: Sequence

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

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.

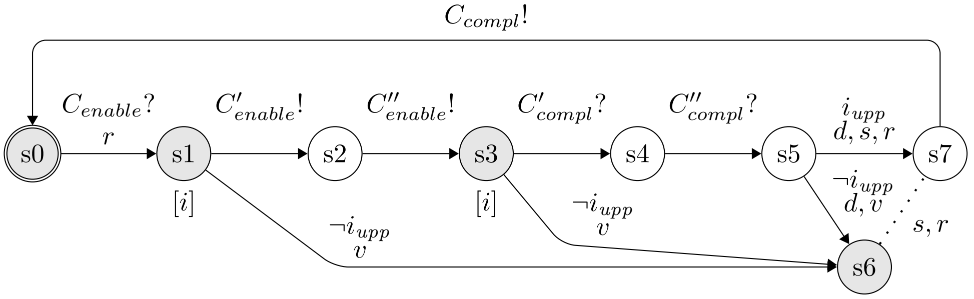
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in order, within the interval i, or that the reparation is respected.

Case 35: Conjunction

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

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.

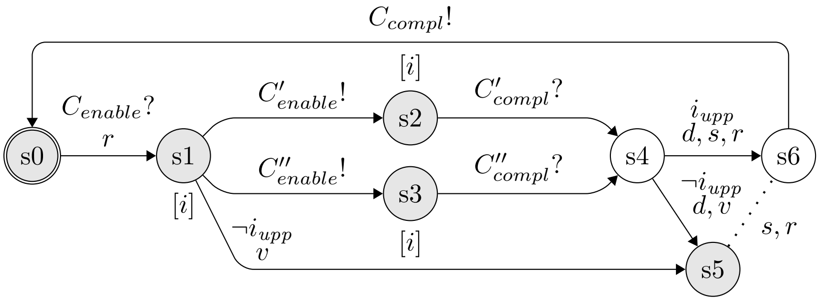
Argument. Both sets of traces require that either both the clauses in the refinement are respected, in any order, within the interval i, or that the reparation is respected.

Case 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.

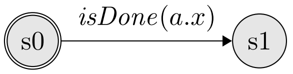
Argument. Both sets of traces require that either only one of the clauses in the refinement is respected within interval i, or that the reparation is respected.

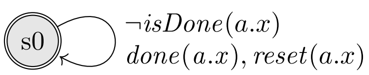
Case 37: Simple action

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

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*.

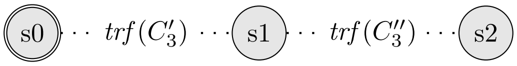
Argument. Both sets of traces require that the action is performed within a certain frame.

Case 38: Action Sequence

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

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 39: Action Conjunction

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

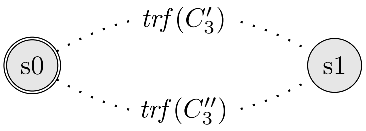
Argument. In both the trace semantics and the translation to Uppaal, the *And* refinement is defined in terms of *Seq* and *Or*, and thus needs no special treatment here.

Case 40: Action Choice

$$\sigma \models^{c,a}_{t_0} C'_3 \ \mathit{Or}\ C''_3 \text{ iff either } \sigma \models^{c,a}_{t_0} C'_3 \text{ or } \sigma \models^{c,a}_{t_0} C''_3$$

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 41: Action Naming

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

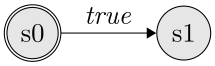
Argument. This case is simply handled recursively by considering the inner C_2 element.

Case 42: Top

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

Event traces. Any event trace respects *top*.

Translation. The following automaton fragment.



Uppaal *traces.* This automaton is trivially satisfied by any trace.

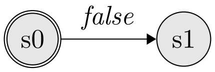
Argument. Both sets of traces are maximally inclusive.

Case 43: Bottom

$$\sigma \not\models^{c}_{t_0} \bot$$

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 44: Reference

$$\sigma \models^{c}_{t_0} \#name \text{ iff } \sigma \models^{c}_{t_0} lookup(name)$$

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

Appendix B. Case Study

"Support and Service Level Schedule" for the 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)