A Domain-Specific Language for Normative Texts with Timing Constraints

Runa Gulliksson, John J. Camilleri
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
Email: runa.gulliksson@gmail.com, john.j.camilleri@cse.gu.se

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

***Index Terms*—normative texts; electronic contracts; timed automata; UPPAAL; QuickCheck; embedded DSL**

I. Introduction

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

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

II. Language

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

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

A. Syntax

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

We distinguish two kinds of temporal values in $\mathcal{SCL}$: those which are relative ($\mathbb{T}_r$), as in *5 days*, and those which are absolute ($\mathbb{T}_a$), as in *31st May 2016*. Again for simplicity, here we treat both relative and absolute temporal values as natural numbers. However these could just as easily be implemented as types representing "real" time units and calendar dates without any changes to the language.

The full syntax of $\mathcal{SCL}$ is shown in Figure 1. We use the term *clause* (or *sub-clause*) to refer to anything of type C, while *contract* refers to a list of top-level clauses (type $Contract$). In brief, $\top$ is the trivially satisfied clause while $\bot$ is unsatisfiable and indicates irreparable violation. O, P and F are the basic deontic operators over actions described above. The declaration operator D allows variables to be updated using either literal values or other variables. A clause can be

2332-6468/16 $31.00 © 2016 IEEE
DOI 10.1109/TIME.2016.14

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

Fig. 1. $\mathcal{SCL}$ syntax.

Named so that its status may be queried in guard expressions. Clauses can be refined into sub-clauses by conjunction And, exclusive choice Or, and sequence Seq. The reparation operator Rep specifies an alternative clause to be applied if the clause is violated.

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

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

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

B. Example

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

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

C. Implementation

$\mathcal{SCL}$ has been implemented in Haskell [1] as an embedded domain-specific language (EDSL) [2]. This allows us to implement the language without the need to design a concrete syntax or build any compilation tools. Haskell makes a suitable host language because its algebraic data types allow for a clean and direct implementation of $\mathcal{SCL}$'s combinators, whilst its strong static type system can be leveraged to ensure that all constructed $\mathcal{SCL}$ terms are type-correct.

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

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

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

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

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

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

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

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

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

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

This produces the following clause:

```
Seq
  (Seq
    (Rep
      (D "pass_proposal" (VInt 1))
      (Named
        "8893"
        (At 10 (O (Action "publish proposal")))))
    (When
      (GSat "8893")
      (Within 5 (O (Action "submit proposal")))))
  (Seq
    (Seq
      (Rep
        (D "pass_essay" (VInt 1))
        (Named
          "5075"
          (At 22 (O (Action "publish essay")))))
      (When
        (GSat "5075")
        (Within 5 (O (Action "submit essay")))))
    (Seq
      (Rep
        (D "pass_review" (VInt 1))
        (Named
          "4783"
          (At 26 (O (Action "publish review")))))
      (When
        (GSat "4783")
        (Within 5 (O (Action "submit review"))))))
```

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

D. Operational Semantics

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

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

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

The operational semantics for $\mathcal{SCL}$ are defined as a residual function which given a contract and an event trace (and an initial environment) returns an updated contract and environment. We treat time as discrete and actions as instantaneous (they take no time to complete).

An event trace is expanded into a sequence of *steps*, where a step is either: the doing of an action a (indicated $\xrightarrow{a}$), an update of a variable ($\xrightarrow{x=1}$), or a delay of one time unit ($\rightsquigarrow$). We use the $\dot{\rightarrow}$ to mean a single step of any kind. The event trace above is thus expanded to the sequence of steps:

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

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

Apart from a trace, the evaluation of a contract also requires an *environment* Γ which maps actions to time stamps ($\Sigma \mapsto \mathbb{T}_a$) for recording when they take place, names to their inner clauses ($\mathcal{N} \mapsto C$), and variables to their integer values ($\mathcal{V} \mapsto \mathbb{Z}$). It also contains a variable t_0 which indicates the current time in *ticks* (number of delay steps consumed). An action step sets the time stamp for the corresponding action to the current time ($\Gamma[a := t_0]$) while a delay step increments the current time by one unit ($\Gamma[t_0 \mathrel{+}= 1]$). Guard predicates are evaluated within an environment as follows: $\mathsf{done}(a)$ holds if action a has occurred ($\Gamma[a] > -1$); $\mathsf{sat}(n)$ holds if name n is satisfied $\top$ ($\Gamma[n] = \top$); $\mathsf{earlier}(t)$ holds if $\Gamma[t_0] < t$ and $\mathsf{later}(t)$ holds if $\Gamma[t_0] > t$.

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

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

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

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

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

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

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

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

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

$Named\langle \mathsf{Lab}, Rep\langle \ldots, Within\langle 5, \top \rangle \rangle \rangle$ by (1)
$Named\langle \mathsf{Lab}, Rep\langle \ldots, \top \rangle \rangle$ by (7)
$Named\langle \mathsf{Lab}, \top \rangle$ by (8)

where the final state of the evaluation environment is:

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

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

III. Translation to Timed Automata

To enable property-based analysis on contract models, we define a translation from $\mathcal{SCL}$ into *networks of timed automata* (NTAs). A *timed automaton* (TA) [3] is a finite automaton extended with clock variables which increase their value as time elapses, all at the same rate. The model also includes clock constraints, allowing clocks to be used in guards on transitions and in invariants on locations, in order to restrict the behaviour of the automaton. Clocks can be reset to zero during the execution of a transition. An NTA is a set of TAs which are run in parallel, sharing the same set of clocks. The definition of NTA also includes a set of channels which allow synchronisation between independent automata.

Uppaal [4] is a tool for the modelling, simulation and verification of real-time systems. The modelling language used in Uppaal extends timed automata with a number of features, amongst them the concepts of *urgent* and *committed* locations which prevent time from elapsing when any process is in such a location. It also introduces the idea of *broadcast* channels, which allow one sender to synchronise with an arbitrary number of receivers. The query language of Uppaal, which is used to define properties to be checked over a system of automata, is a subset of timed computation tree logic (TCTL) [5].

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

$$At_{Thru}\ \frac{C \dot{\rightarrow} C'}{At\langle t, C \rangle \dot{\rightarrow} At\langle t, C' \rangle}\ \Gamma[t_0] \leq t \tag{2}$$

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

$$Seq_{Thru}\ \frac{C_1 \dot{\rightarrow} C_1'}{Seq\langle C_1, C_2 \rangle \dot{\rightarrow} Seq\langle C_1', C_2 \rangle} \tag{4}$$

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

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

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

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

Fig. 2. Selection of $\mathcal{SCL}$ semantic rules. The predicate $\mathsf{isTop}(C)$ holds if C is $\top$, taking into account clause naming.

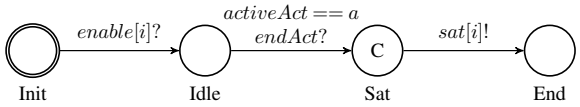

Fig. 3. Template for the obligation $O\langle \mathsf{a} \rangle$, where i is the index of the clause.

A. Modularity

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

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

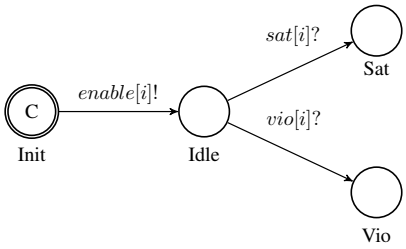

Fig. 4. A *Start* template, where i is the index of the clause being started.

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

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

The *Start* process will send a synchronisation signal on the enabling channel with index 0, which is the index that the *And* process is listening to. This in turn enables processes with indexes 1 and 2, corresponding to $O\langle \mathsf{a}\rangle$ and $O\langle \mathsf{b}\rangle$ respectively, where `a` and `b` are constants referring to the indexes for those actions.

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

B. Step generation

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

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

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

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

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

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

C. Automatic testing for correctness

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

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

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

To thoroughly test our translation we repeat this process for many different contracts and traces which are generated randomly. To do this we use QuickCheck [6], a tool for testing Haskell programs automatically. By providing a specification of a program in the form of properties which its functions should satisfy, QuickCheck then tests that these properties hold in a large number of randomly generated cases. QuickCheck

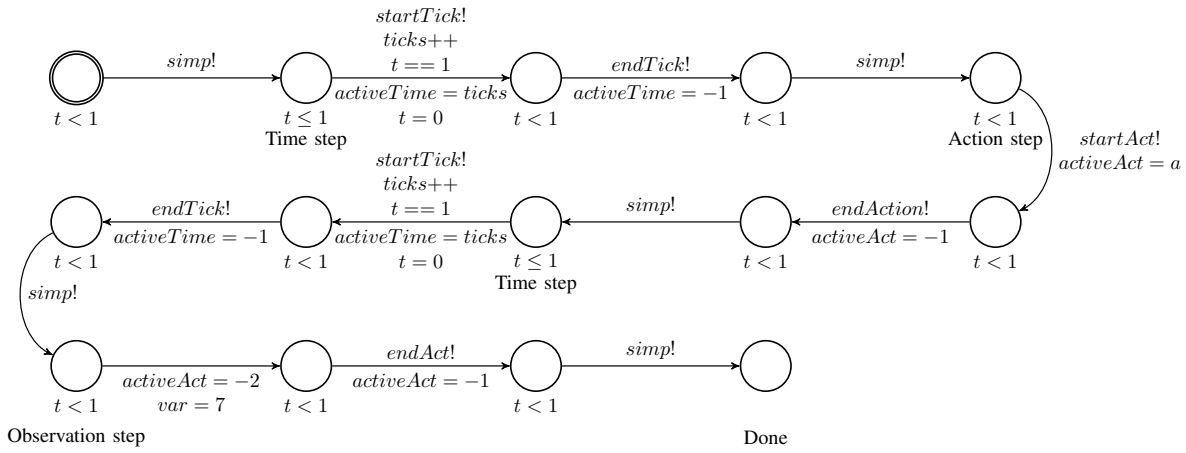

Fig. 5. Template for the trace $[1 : a,\ 2 : v = 7]$ (action a taking place at time 1 and variable v being updated to 7 at time 2).

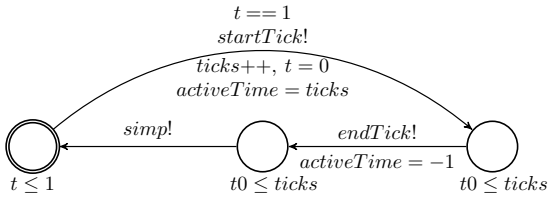

Fig. 6. *Ticker* template, which triggers delay steps in the system.

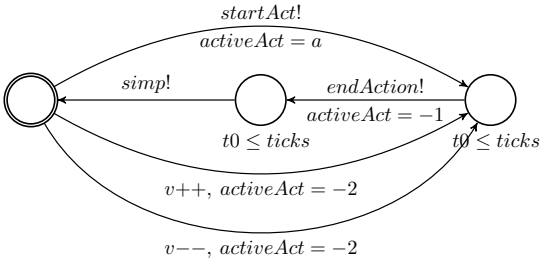

Fig. 7. *Doer* template, containing transitions for action a and variable v. Further actions can be added to the same template by replicating the top transition and replacing the relevant action index in place of a.

provides combinators for defining properties, building test data generators, shrinking counter-examples, and observing the distribution of test data.

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

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

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

IV. CASE STUDY

To demonstrate the use of $\mathcal{SCL}$, we apply it to a small case study concerning the rules governing the running of a university course. A textual description of these rules can be found in Figure 8. The corresponding $\mathcal{SCL}$ model for this case study is shown in Figure 9. It is a contract with 6 top-level clauses, where each has been named for convenience. This model was built manually by the authors. While automating this modelling process is of great interest to us, it is beyond the scope of the present work. For more about this, see [7]. The rest of this section describes the various ways in which this contract model can be tested and verified.

A. Trace-based unit testing

The first thing we can do with our model is to come up with various concrete event traces and evaluate the contract against them. This can be done either by (i) using the pure implementation of the $\mathcal{SCL}$ semantics (Section II-D), or (ii) by

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

Fig. 9. $\mathcal{SCL}$ contract for the course case study.

translating the model and trace into UPPAAL (Section III). If all we are concerned with is contract satisfaction, then it should make no difference which method is used, as both representations behave equivalently (as argued in Section III-C).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B. Trace-based random testing

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

These constraints are implemented as Haskell functions along with the rest of our framework. Consider the following:

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

This example is a conjunction of three sub-constraints, which state that: (i) the trace contains the actions regCourse, submit1 and accept1 in that specific order; (ii) action submit1 occurs between time stamps 2 and 11; and (iii) the trace should not contain action resubmit1 at all.

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

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

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

which is reduced to the minimal trace:

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

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

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

C. Verification with temporal properties

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

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

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

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

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

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

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

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

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

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

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

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

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

V. RELATED WORK

The $\mathcal{SCL}$ language is inspired by *C-O Diagrams*, introduced by Martínez et al. [8]. The idea of translation into NTA

for analysis also comes from this work [9]. Our language is compositional, where each operator serves a single purpose, and thus is structurally quite different from the *C-O Diagram* language. However both languages cover very much the same concepts, and conversion from a *C-O Diagram* into an $\mathcal{SCL}$ term would be easy to do automatically (with the exception of clauses involving repetition).

Our ultimate goal is to produce a usable high-level system for end-to-end analysis of normative contracts. The present work continues on that of Camilleri [7], which describes the other components and considerations such a system requires. These include not only the user interface and natural language processing aspects of modelling, but also a user query language which can abstract away from the different analysis methods discussed here.

The AnaCon [10] framework for contract analysis, based on the contract logic $\mathcal{CL}$ [11], has a similar goal but more limited scope than the current work. In particular, their underlying logical formalism contains no direct temporal notions other than sequencing, and the only kind of analysis possible is the detection of normative conflicts using the CLAN tool [12].

Pace and Schapachnik [13] introduce the *Contract Automata* formalism for modelling interacting two-party systems. Their approach is similarly based on deontic norms, but with a strong focus on synchronous actions where a permission for one party is satisfied together with a corresponding obligation on the other party. Their formalism is limited to strictly two parties, and does not have any support for timing notions, which are key to our work.

Marjanovic and Milosevic [14] also defend a deontic approach for modelling of contracts. They pay special attention to temporal aspects, distinguishing between three different kinds of time: absolute, relative and repetitive. They also introduce visualisation concepts such as *role windows* and *time maps* and describe how they could be used as decision support tools during contract negotiation. Their ideas however do not seem to have been implemented as any usable system.

Wyner [15] 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 a considerable body of work in the representation of contracts as knowledge bases or *ontologies*. The *LegalRuleML* project [16] is one of the largest efforts in this area, providing a rule interchange format for the legal domain, enabling modelling and reasoning that lets users evaluate and compare legal arguments. The format has a temporal model which supports the evolution of legal texts over time, such that their legal reasoner will dynamically apply the version of a document that was applicable at the time of a particular event.

Peyton Jones et al. [17] introduce a functional combinator language for working with complex financial contracts—the kind which are traded in derivative markets—which they also implement as a DSL embedded in Haskell. These kinds of contracts are somewhat different from the normative documents we are concerned with in that they do not feature the deontic modalities, and are thought of as having an inherent financial value which varies over time. The compositional style of $\mathcal{SCL}$ is however similar to their language. For more recent work continuing this along these lines, see [18].

VI. Conclusion

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

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

Limitations

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

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

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

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

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

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

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

Source code

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

ACKNOWLEDGEMENT

The authors wish to thank the Swedish Research Council for financial support under grant number 2012-5746.

REFERENCES

[1] S. Marlow, *Haskell 2010 Language Report*, 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf

[2] D. Ghosh, "DSL for the Uninitiated," *Communications of the ACM*, vol. 54, no. 7, pp. 44–50, July 2011.

[3] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[4] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *STTT*, vol. 1, no. 1-2, pp. 134–152, 1997.

[5] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on UPPAAL 4.0," Department of Computer Science, Aalborg University, Tech. Rep., 2006.

[6] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *ICFP 2000*. ACM, 2000, pp. 268–279.

[7] J. J. Camilleri, "Analysing Normative Contracts — On the Semantic Gap between Natural and Formal Languages," Licentiate thesis, Chalmers University of Technology and University of Gothenburg, Sweden, 2015.

[8] E. Martínez, E. Cambronero, G. Díaz, and G. Schneider, "A Model for Visual Specification of e-Contracts," in *SCC 2010*. IEEE Computer Society, 2010, pp. 1–8.

[9] G. Díaz, M. E. Cambronero, E. Martínez, and G. Schneider, "Specification and Verification of Normative Texts using C-O Diagrams," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 795–817, 2014.

[10] K. Angelov, J. J. Camilleri, and G. Schneider, "A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language," *JLAP*, vol. 82, no. 5-7, pp. 216–240, 2013.

[11] C. Prisacariu and G. Schneider, "CL: An Action-based Logic for Reasoning about Contracts," in *WOLLIC 2009*, ser. LNCS, vol. 5514. Springer, 2009, pp. 335–349.

[12] S. Fenech, G. J. Pace, and G. Schneider, "CLAN: A Tool for Contract Analysis and Conflict Discovery," in *ATVA 2009*, ser. LNCS, vol. 5799. Springer, 2009, pp. 90–96.

[13] G. J. Pace and F. Schapachnik, "Contracts for Interacting Two-Party Systems," in *FLACOS 2012*, ser. EPTCS, vol. 94, 2012, pp. 21–30.

[14] O. Marjanovic and Z. Milosevic, "Towards Formal Modeling of e-Contracts," in *EDOC 2001*. IEEE Computer Society, 2001, pp. 59–68.

[15] A. Z. Wyner, "Violations and fulfillments in the formal representation of contracts," Ph.D. dissertation, Department of Computer Science, King's College London, 2008.

[16] T. Athan, H. Boley, G. Governatori, M. Palmirani, A. Paschke, and A. Wyner, "OASIS LegalRuleML," in *ICAIL 2013*. ACM, 2013.

[17] S. Peyton Jones and J.-M. Eber, "How to write a financial contract," in *The Fun of Programming*, G. and de Moor, Ed. Palgrave Macmillan, 2003, pp. 105–129.

[18] P. Bahr, J. Berthold, and M. Elsman, "Certified symbolic management of financial multi-party contracts," in *ICFP 2015*. ACM, 2015, pp. 315–327.