CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# From Domain-Specific Language to Timed Automata

## Automatic Translation and Verification of Contract Specifications

*Master's thesis in Algorithms, Logic and Languages*

Runa Gulliksson

Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden, 2016

# From Domain-Specific Language to Timed Automata
# Automatic Translation and Verification of Contract Specifications

Runa Gulliksson



Examiner: Patrik Jansson

Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, May 2016

**Abstract**

Analysis of contracts is becoming an increasingly important subject due to the amount of agreements on the web. In this thesis a compositional formal language, Simplified Contract Language, SCL, is used to represent contracts. A translation between SCL and Timed Automata is designed and implemented, in order to verify contracts using temporal logic. UPPAAL is used as the timed automata verifying tool.

The translation is shown to preserve the behavioral semantics of the SCL. The translation is tested thoroughly, using QuickCheck, against an implementation of the semantics in terms of trace acceptance. A case study of a university course, modeled as a contract, is done. It shows that it is possible to use the SCL and the translation for analyzing a real world contract with different traces. The case study also shows that when randomly generating events the state space can get large enough to slow down the verification speed significantly.

**Keywords**

Timed Automata, UPPAAL, Contract analysis, Simplified Contract Language, QuickCheck

## Acknowledgements

First I would like to thank John J.Camilleri for his support and collaboration during this work. His supervision and that of Gerardo Schneider has been of great value for this work. I would also like to thank Patrik Jansson, my examiner, Robert Kemi for his interest in opposing on this thesis and Sebastian Olsson for his support.

# Contents

# 1. Introduction

With the increased use of Internet and web services we are asked to agree to more and more contracts, and a lot of the time we do this without really knowing what we are agreeing to [1]. This is a problem since the contracts are legally binding. Usually there is a large body of text and a juridic jargon making it a tedious task to understand all conditions of one contract let alone several contracts. This problem is gaining more interest and some projects exist aimed at helping users understand what they are agreeing to [2][3]. Some rely on people reading and analyzing and others on more automatic approaches, looking for keywords, by means of machine learning etc.

This thesis is part of a larger research project conducted by the Formal Methods & Language Technology research groups at Chalmers University and University of Gothenburg, especially the work by John J. Camilleri [4]. Camilleri is working on the analysis of legal documents in the form of normative contracts i.e. contracts written in natural language. An overview of the system being researched can be seen in Figure 1.1. The contracts are first modelled and then translated into a formal language, Simplified Contract Language (SCL). SCL is a formal language developed by John J. Camilleri. It is used to represent a contract written in natural language in a formal and unambiguous way. The formal representations in SCL are then translated into timed automata [5].

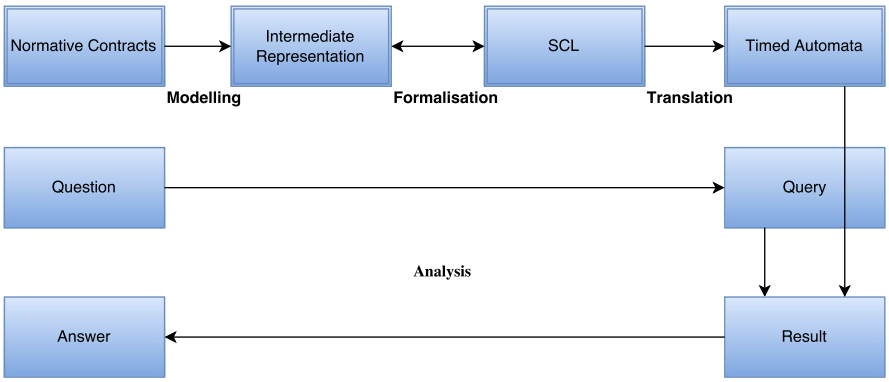



Figure 1.1: Overview of the analysis system

Timed automata is a theory for modeling finite automata with clocks. It has finite sets of nodes and edges, with clock constraints enabling specification of real-time systems. Several verification tools exist based on timed automata including UPPAAL,

KRONOS [6] and CMC [7]. In this project UPPAAL is used for verification and simulation. UPPAAL can test things like safety, liveness and reachability properties [8]. Once the contracts are modelled in timed automata it is possible to write queries to verify conditions of the contracts.

This thesis contributes to one part of the analysis system, the translation between SCL and timed automata, the translation step in Figure 1.1.

## 1.1 Related work

Some previous work have been done on modeling contracts by using other formal languages constructed for contracts. One of these formal languages is Contract Logic, CL [9]. CL models obligations, permissions and prohibitions of actions combined by propositional dynamic logic. CL does not have time or timing constraints in its language. Another language for contract modelling is Contract-Oriented (C-O) Diagrams [10]. C-O Diagrams are similar to SCL, as to the type of conditions they can model but they use a different non-compositional structure. C-O Diagrams have been translated into timed automata in order to run verifications with UPPAAL. SCL has a different structure, which should make it easier to model in timed automata.

Translating models into timed automata and using UPPAAL for verification and analysis have been done for other languages than SCL and with other systems to examine. For example, a research project translating Functional Block Diagrams in order to verify IEC 61131-3 based safety applications, was described in [11]. Translations of Timed Petri Nets- into timed automata have also been made for different purposes [12][13]. The language of Timed Petri Nets is often used to model distributed systems. Neither Functional Block Diagrams nor Timed Petri Nets are used here because their structure deviates too much from that of contracts. SCL is designed especially for contract representation, making the modeling of normative contracts less complicated, and more likely to be correct.

## 1.2 Objective

The objective is to construct an automatic translation between SCL and timed automata (UPPAAL input format) and to ensure that the translation is correct (i.e. that it preserves the behavioral semantics and conditions from the SCL code).

## 1.3 Scope

SCL has been implemented in Haskell as an embedded domain-specific language. The contracts represented in SCL will be translated to UPPAAL input format, which enables verification with UPPAAL, from within the code.

Testing the timed automata design will be done using the QuickCheck testing tool [14]. The testing will be executed against an implementation of the operational seman-

tics, in terms of trace acceptance. A case study will be performed in order to further evaluate the result with a real world problem. In the case study, a contract modeling the conditions present in a university course will be used. To test if the contract's properties hold, queries stating conditions from the course will be verified with UPPAAL.

UPPAAL will be used as the timed automata verifying tool. A proof that the translation preserves the behavior will not be attempted, due to lack of time. Neither will optimizing the verification speed in UPPAAL (except for a minimization of the contract used in the case study).

# 2. Theory

This section starts with a presentation of the SCL. Following this is a description of timed automata and UPPAAL, which SCL is to be translated into. After this the testing tool, QuickCheck, is described.

## 2.1 Simplified Contract Language (SCL)

SCL is a compositional formal language for contracts, based on the atomic deontic operators for obligation $O$, permission $P$ and prohibition $F$. It is developed by John J. Camilleri with some input from this project.

### 2.1.1 Syntax

A contract, *Contract*, is defined as a list of clauses, $C$. Clauses are recursively defined so each clause in a contract may be a tree of clauses.

**Definition 2.1.** The SCL syntax is defined by:

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

An action ($a$), is used to represent an event that may take place; $\Sigma$ is the integer set of actions. $O\langle a\rangle$ means that there is an obligation for action $a$ to take place. $\mathcal{V}$ is a set of variables and $\mathcal{N}$ is a set of names. We assume these sets, $\Sigma$, $\mathcal{V}$ and $\mathcal{N}$, are disjoint and global. $\mathbb{T}_r$ represent relative temporal values while $\mathbb{T}_a$ represent an absolute time stamp. Both these values are treated as natural numbers. There is no clause for negating subcontracts. To express that something is not allowed the prohibition clause $F\langle a\rangle$ can be used.

### 2.1.2 Structural Operational Semantics

Contracts are evaluated as time passes and events take place. To represent time changes and events, traces are used. A trace is a list of events ordered by time. For example, a trace stating that an order is placed at time 2 and payed for at time 5 can look like:

$$[2 : \mathsf{order},\ 5 : \mathsf{pay}]$$

Before and between these events time changes, which may evaluate contracts, but it is not written out in the trace. The semantics operate in discrete time. The time progressions and the events are called steps, or transitions. There are 3 kinds of steps; action, observation and delay steps. During an action step, either an *action* takes place

or an *observation* is made. During an observation step, a variable is updated. A delay step increases the time with 1 time unit. Clauses can also be evaluated in between transitions through simplifications. The transitions and the simplifications are used to define evaluation rules for clauses. These rules are defined in Appendix A.

The contract has a shared environment where global variables can be stored. It contains a global clock keeping track on which time step the state is currently in. There are also mappings between names and contracts, variables and values and actions and time stamps, stating at which time an action took place. A short description of how every clause operates is presented below.

- $\top$ is always satisfied.
- $\bot$ is unsatisfiable.
- $Named\langle \mathcal{N}, C \rangle$ assigns a name to a clause so it can be referenced from a guard.
- $D\langle v, V \rangle$ assigns $V$ to $v$ and is then satisfied.
- $O\langle a \rangle$ obligation, action $a$ must take place.
- $P\langle a \rangle$ permission, action $a$ may take place.
- $F\langle a \rangle$ prohibition, action $a$ must not take place.
- $And\langle C, C \rangle$ conjunction, both clauses need to be satisfied.
- $Or\langle C, C \rangle$ choice, one of the clauses need to be satisfied.
- $Seq\langle C, C \rangle$ clauses must be satisfied in sequence. The second clause is not activated until the transition satisfying the first clause is finished.
- $Rep\langle C_r, C \rangle$ reparation, if the $C$ clause is violated the $C_r$ clause can repair the result so that the *Rep* clause may still be satisfied. The reparation is not started until the transition violating the $C$ clause has ended.
- $Wait\langle \mathbb{T}_r, C \rangle$ waits for $\mathbb{T}_r$ time units before evaluating the clause.
- $After\langle \mathbb{T}_a, C \rangle$ waits until time stamp $\mathbb{T}_a$ has passed before evaluating the clause.
- $Within\langle \mathbb{T}_r, C \rangle$ the clause needs to be satisfied within $\mathbb{T}_r$ time units.
- $Before\langle \mathbb{T}_a, C \rangle$ the clause need to be satisfied before time stamp $\mathbb{T}_a$.
- $In\langle \mathbb{T}_r, C \rangle$ the clause needs to be satisfied in $\mathbb{T}_r$ time units. In always waits until all time has passed before evaluating to $\top$ or $\bot$.
- $At\langle \mathbb{T}_a, C \rangle$ the clause needs to be satisfied at time stamp $\mathbb{T}_a$. At always waits until all time has passed before evaluating to $\top$ or $\bot$.

- *When*$\langle G, C\rangle$ the guard, $G$, is checked every transition step. If it evaluates to true the clause, C, is started. As long as the guard is not true the *When* clause is waiting.
- *WhenWithin*$\langle \mathbb{T}_r, G, C\rangle$ the guard, G, is checked every transition step. If it evaluates to true the clause, C, is started. If the guard is not true within $\mathbb{T}_r$ time units *WhenWithin* evaluates to $\top$.
- *WhenBefore*$\langle \mathbb{T}_a, G, C\rangle$ the guard, $G$, is checked every transition step. If it evaluates to true the clause, $C$, is started. If the guard is not true before time stamp $\mathbb{T}_a$ *WhenBefore* evaluates to $\top$.

For example a contract stating that a bill needs to be paid before the end of the month can be constructed like below.

$$[\mathit{Named}\langle \mathsf{Bill}, \mathit{Before}\langle 31, O\langle \mathsf{pay}\rangle\rangle\rangle]$$

Guards ($G$) are evaluated as follows:

- $\mathsf{done}(a)$ is true if action $a$ has occurred.
- $\mathsf{sat}(n)$ is true if name $n$ is mapped to $\top$.
- $\mathsf{earlier}(t)$ is true iff the global time is less than t.
- $\mathsf{later}(t)$ is true iff the global time is more than t.
- *Val* comparisons check if values are less, equal to or larger ($<, =$ and $>$) than other values.
- $\neg G$ negates $G$.
- $G \wedge G$ is true if both guards are true.
- $G \vee G$ is true if either of the guards are true.

### 2.1.3 Contract evaluation

When evaluating contracts the root clause in each tree of clauses is initiated. That clause can in turn initiate other clauses, wait for a result or return a result depending on the behavior of that specific clause. At each transition step the clauses are evaluated with a modified tree and updated environment as a result.

The Bill contract from section 2.1.2 is satisfied by trace $[2 : y = 3,\ 5 : \mathsf{pay}]$ since the action pay takes place at time 5 which is before 31. The evaluation steps for this can be seen in Table 2.1. The trace also includes an observation step stating that during time 2 the variable $y$ is set to 3, but this will not effect the outcome of the contract. Table 2.2 shows an evaluation of the same contract but without the action pay in the trace. Naturally without the action pay the contract is violated at the time 31.

| **Contract** | **Trace** | **Environment** |
|---|---|---|
| $[Named\langle \mathsf{Bill}, Before\langle 31, \langle O\langle \mathsf{pay}\rangle\rangle\rangle\rangle]$ | $[2 : y = 3,\ 5 : \mathsf{pay}]$ | $t == 0, y == 0, \mathsf{pay} == -1$ |
| $[Named\langle \mathsf{Bill}, Before\langle 31, \langle O\langle \mathsf{pay}\rangle\rangle\rangle\rangle]$ | $[5 : \mathsf{pay}]$ | $t == 2, y == 3, \mathsf{pay} == -1$ |
| $[Named\langle \mathsf{Bill}, Before\langle 31, \top\rangle\rangle]$ | [] | $t == 5, y == 3, \mathsf{pay} == 5$ |
| $[Named\langle \mathsf{Bill}, \top\rangle]$ | [] | $t == 5, y == 3, \mathsf{pay} == 5,$ $\mathsf{Bill} == \top$ |

Table 2.1: Example contract evaluated by trace is satisfied

| **Contract** | **Trace** | **Environment** |
|---|---|---|
| $[Named\langle \mathsf{Bill}, Before\langle 31, \langle O\langle \mathsf{pay}\rangle\rangle\rangle\rangle]$ | $[2 : y = 3]$ | $t == 0, y == 0$ |
| $[Named\langle \mathsf{Bill}, Before\langle 31, \langle O\langle \mathsf{pay}\rangle\rangle\rangle\rangle]$ | [] | $t == 2, y == 3$ |
| $[Named\langle \mathsf{Bill}, \bot\rangle]$ | [] | $t == 31, y == 3, \mathsf{Bill} == \bot$ |

Table 2.2: Example contract evaluated by trace is violated

In addition to contracts being satisfied or violated there is a third possibility. When a clause without any time constraints is waiting for an action that never takes place, that clause will never evaluate. For example the contract $O\langle \mathsf{pay}\rangle$ states that there is an obligation to pay but since there is no deadline specified, the contract might never evaluate any further. If this contract is evaluated by a trace without the action pay, the contract will be unevaluated and thus neither $\top$ nor $\bot$. Unevaluated contracts are called non-violated and can be useful for some properties in contracts.

## 2.2 Timed Automata

Timed automata was developed by Alur and Dill in 1994 [15]. It is a simple and powerful way of modeling state-transition graphs with clock constraints. Clocks are variables which increase their value as time passes. All clocks progress at the same rate. It is often used to model and analyze real-time systems. Figure 2.1 shows a timed automaton modeling a time switch. OFF is the start location. When the transition from OFF to ON is made, the clock $x$ is reset to 0. The ON node has an invariant stating that $x$ may be no more than 3 on that node. To avoid a deadlock when the clock becomes larger than 3 an edge has to be taken from that node. The edge back to the OFF node has a guard stating that $x$ needs to be equal to 3 for transitions to be allowed. This makes sure that the progress back to OFF always take place exactly when $x$ is equal to 3. These attributes present in Figure 2.1 are the main components of timed automata. Real time systems are usually represented by more than one timed automaton.

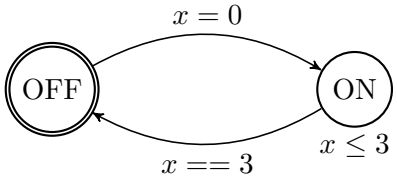



Figure 2.1: Timed Automata used to model a time switch

### 2.2.1 UPPAAL

UPPAAL uses an extended version of timed automata [16]. It contains binary synchronization between automata, through channels, and urgency demands on locations and channels. Another extension is that data variables are allowed as guards and may be reset during an action transition.

Channels can provide synchronization between two automata (also called templates) in UPPAAL. "!" is used to symbolize sending on a channel and "?" for receiving. This communication is binary and done one-to-one. However it is possible to use a broadcast channel where one sender sends to a number of receivers. Figure 2.2 shows two templates where Template 1 sends on *channelA* to Template 2. The two templates will progress on the edges simultaneously, but the transition may be blocked by the guard, $x == 5$. If that guard is true ($x$ is equal to 5) then the templates will move on to their next nodes. If $x$ is not equal to 5 the system will never progress. If *channelA* is a broadcast channel Template 1 may progress even when the guard is not true.


Figure 2.2: Templates using synchronization in UPPAAL

UPPAAL has been extended with committed locations. If a process is in a committed location no time is allowed to pass and the next transition needs to be on an edge away from a committed location. If there is no satisfied edge away from the committed location, the system will deadlock. In Figure 2.2 the initial node in Template 1 is committed, so no time can pass before the synchronization on *channelA*. There is also a second progress demand setting, urgent. Both locations and channels can be set to urgent. No time is allowed to pass in urgent locations, and urgent channels will be sent on without any time delays. If a system is in both a committed and an urgent location the committed location will be left first. Progress will still be made from the urgent location before any time can pass.

**Definitions**

The following definitions for UPPAAL models are reproduced from David et al. [17]. David et al. also define the semantics of UPPAAL.

**Definition 2.2.** A template (single automaton) $A$ in UPPAAL 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 $g$, synchronization label $s$ and assignment $a$,
3. $Type : L \rightarrow \{ordinary, urgent, committed\}$ is a typing function that marks each location as ordinary, urgent or committed, and
4. $l^0 \in L$ is the initial location.

**Definition 2.3.** An UPPAAL model M (network of automata) is a tuple $\langle \overrightarrow{A}, Vars, Clocks, Chan, Type \rangle$, where

1. $\overrightarrow{A}$ is a vector of templates $A_1, ..., A_n$,
2. $Vars$ is a set of variables,
3. $Clocks$ is a set of clocks,
4. $Chan$ is a set of synchronization channels, and
5. $Type$ is a polymorphic typing function for locations, channels and variables.

To verify models in UPPAAL, properties and expressions are used to construct queries [8]. Expressions are local or global variables or values, which can be combined together by a number of operators. The available properties are shown in Table 2.3.

| **Name** | **Property** |
|---|---|
| Possibly | $E\Diamond\ p$ |
| Invariantly | $A\Box\ p$ |
| Potentially always | $E\Box\ p$ |
| Eventually | $A\Diamond\ p$ |
| Leads to | $p \implies q$ |

Table 2.3: Properties for constructing queries in UPPAAL, where $p$ and $q$ are expressions

A query stating that the bill contract from section 2.1.2 can be satisfied can look like below. The variable *status* states whether or not the clause named Bill is satisfied.

$$E\Diamond\ \mathsf{Bill.status} == SAT$$

## 2.3 QuickCheck testing tool

QuickCheck is a testing tool that uses property based testing with arbitrary generated test cases [18]. In order to use QuickCheck a property needs to be defined. A property has one or several data types as input, and returns a Boolean value stating the test result. When running the property with QuickCheck a large number of test cases can be randomly generated for the specified data types in the input parameters. An example property, that checks that a guard evaluates to the same value after negating it twice, can be seen in Figure 2.3.

```
prop :: Guard -> Bool
prop g = eval g == eval (GNot (GNot g))
```

Figure 2.3: Property for testing double negation of a guard, where *eval* is a function evaluating a guard to a Boolean value

It is possible to add conditions on the test cases in the properties. Test cases that do not satisfy the condition are not evaluated by the property but are still regarded as passed tests. This makes it important to check the distribution of the cases, when using conditions, so that there still is a sufficient number of tests actually being evaluated. An example of this, where a test case is used only when the guard evaluates to True can be seen in Figure 2.4. When a test case fails to satisfy the property, a counterexample is presented. QuickCheck shrinks the example before presenting it [19]. This means that similar and smaller test cases are tested until the smallest counterexample is found. This is useful when debugging since it removes unnecessary information and focuses on the actual reason for the failed test.

```
prop :: Guard -> Bool
prop g = eval g == True ==> eval g == eval (GNot (GNot g))
```

Figure 2.4: Property that will only use test cases when g evaluates to True

Arbitrary is a class that can be used for generating test cases of a certain type [14]. With Arbitrary it is possible to generated more specific test cases for common types or test cases for user-defined data types. To aid in designing generators there are combinators to choose between options as well as a frequency setting to control the distribution. An example of a simplified Arbitrary instance for a guard can be seen in Figure 2.5. When using an Arbitrary type generator a shrinking function needs to be defined in order to shrink test cases. A shrinking function specifies how the type can be reduced to create smaller test cases.

```
instance Arbitrary Guard where
  arbitrary = do
    g1 :: Guard <- arbitrary
    g2 :: Guard <- arbitrary
    ts :: Time <- arbitraryTime

    frequency $ map (\(a,b)->(a, return b))
      [ (2, GEarlier ts)
      , (2, GLater ts)
      , (1, GNot g1)
      , (1, GAnd g1 g2)
      , (1, GOr g1 g2)
      ]
```

Figure 2.5: Arbitrary being used to generate simplified instances of type *Guard*

# 3. Method

The method sections consist of two parts, the translation and the testing. In the Translation section it is explained how contracts written in SCL are modelled in UPPAAL and how the behavioral semantics is preserved. This is followed by the Testing section that describes the testing method along with the case study.

## 3.1 Translation

For each of the SCL clause constructors ($C$) a template, or timed automaton, is built in UPPAAL, that models the behavior of the clause. Every template representing a clause has a channel receiver 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. In Figure 3.1 the template for the obligation clause can be seen with enable and response synchronization channels. Actions are represented by integer variables in UPPAAL.

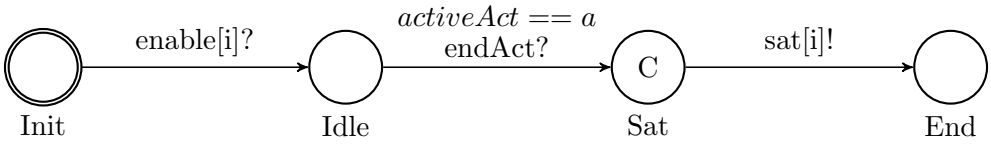



Figure 3.1: Obligation template, $O\langle \mathtt{a} \rangle$, where `enable` and `sat` are lists of channels, $i$ is the index of the clause, `activeAct` is a state variable, $a$ is an action and `endAct` is a broadcast channel

### 3.1.1 Construction of Contracts

The templates are linked together by these start and response channels to form contracts. For every tree of clauses a start template is used, see Figure 3.2. The start template enables the root of the tree and then waits for either a violated or a satisfied signal to be returned. The first location is committed, meaning that the first thing that happens (before time passes) is the enabling of the first clause.

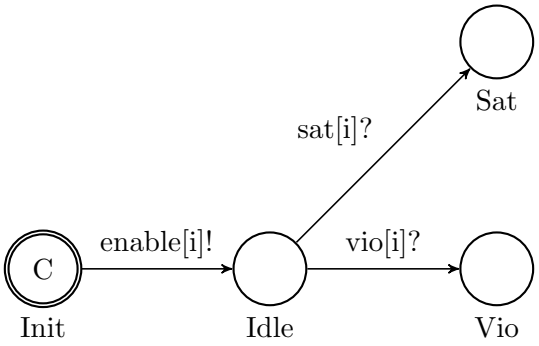
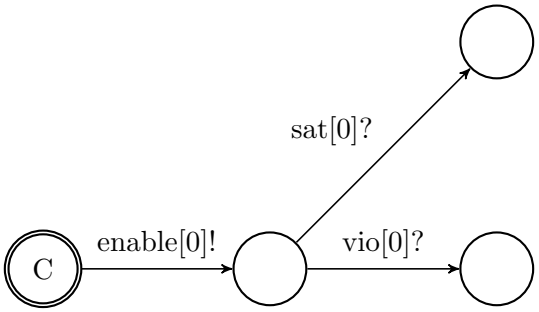



Figure 3.2: A *Start* template, where `enable`, `vio` and `sat` are lists of channels and $i$ is the index of the clause being started

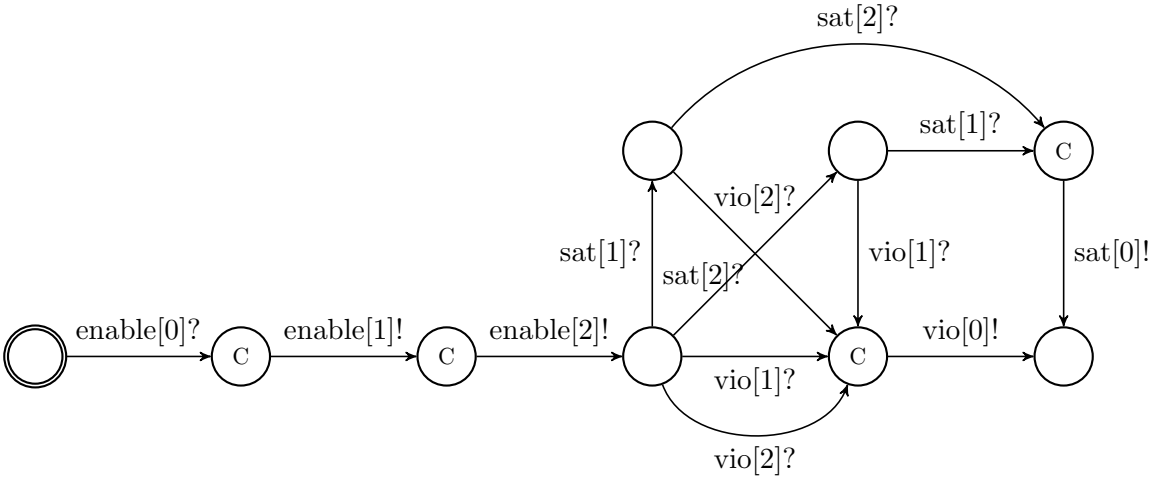
Contracts are constructed by creating instances of templates in the system declarations in UPPAAL. They are linked together by specifying indexes for the enabling/finishing channels as parameters. Figure 3.3 shows the declarations for the contract $[And\langle O\langle \mathsf{a}\rangle, O\langle \mathsf{b}\rangle\rangle]$. A *Start* template will send on the enabling channel on index 0, which is the index that the *And* template is listening to. The *And* template in turn enables index 1 and 2, ($O\langle \mathsf{a}\rangle$ and $O\langle \mathsf{b}\rangle$). The template instances generated by the declarations in Figure 3.3 can be seen in Figure 3.4. The templates for $When$, *WhenWithin* and *WhenBefore* are built for every occurrence of them in the contract. This is due to the fact that they use a guard ($G$), that can not be used a parameter.

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

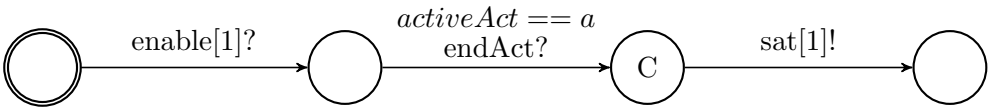
Figure 3.3: Declarations for the contract $[And\langle O\langle \mathsf{a}\rangle, O\langle \mathsf{b}\rangle\rangle]$



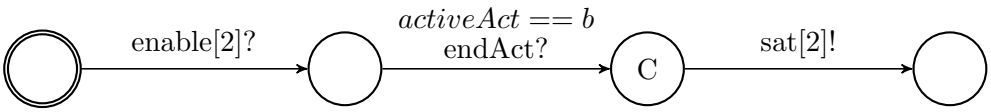

(a) *Start* template that sends, and then receives a response, on index 0




(b) *And* template, that is enabled on index 0 and then starts templates on index 1 and 2




(c) *O* template, that is enabled on index 1




(d) *O* template, that is enabled on index 2

Figure 3.4: Network of templates used to represent the contract $[And\langle O\langle \mathtt{a} \rangle, O\langle \mathtt{b} \rangle\rangle]$, where `enable`, `sat` and `vio` are lists of channels, `activeAct` is a state variable, $a$ and $b$ are actions and `endAct` is a broadcast channel. (b) is an implementation of parallel *And*

Templates for clauses lower in a tree are assigned a higher priority compared to those higher up in the tree. This means child templates will always have a higher priority than their parents. In contract $[And\langle O\langle \mathtt{a} \rangle, O\langle \mathtt{b} \rangle\rangle]$ the $O$ templates will have a higher priority than the *And* template. The templates are prioritized so that templates higher up are able to receive responses from those further down, when the templates expire at the same time.

### 3.1.2 Global Time

Since SCL operates in discrete time and UPPAAL in real time, discrete time is simulated in UPPAAL. A global clock `t0` is used to keep track of the system time. When `t` is used it refers to a local clock (a clock for a specific template). There is also an integer variable, `ticks`, which is incremented every time step to always reflect the discrete time. If the contract should be evaluated by a specific trace, that trace is translated into a template. If there is no trace, two templates (a *Ticker* and a *Doer*) are built to simulate the passing of time and events taking place.

In a *Trace* template every transition is predetermined. The next transition is either a time step, an action, an observation or a simplification. A time step takes place when time turns from one time unit to another and corresponds to the global clock moving to a new natural number in UPPAAL. Invariants are used to make sure the time synchronization is sent at the right time. An example trace is shown in Figure 3.5. The example trace is $[1 : a,\ 2 : v = j]$ (action $a$ take place at time 1 and the variable $v$ is updated to $j$ at time 2).

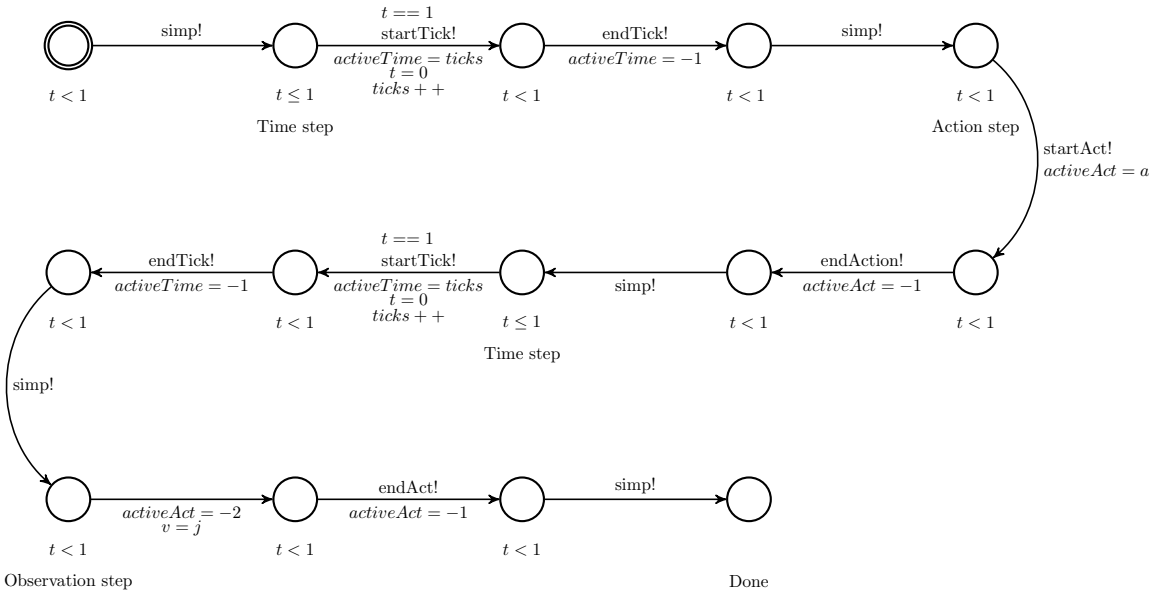



Figure 3.5: Trace template showing the example trace $[1 : a,\ 2 : v = j]$, where $a$ is an action, $v$ and `ticks` are variables, $j$ is an integer or a variable, `activeTime` and `activeAct` are state variables, $t$ is a local clock and `simp`, `startTick`, `endTick`, `startAct` and `endAct` are broadcast channels

The other way of simulating steps is to have a *Ticker* template and a *Doer* template, see Figure 3.6 and Figure 3.7. The *Ticker* template handles the simulation of time. There is a time limit on the *Ticker* which can be set as an input option. The time limit makes it possible to limit the duration of the simulation or verification. The *Doer* may generate any action in the contract or change the value of any variable. When a variable is set it can either be incremented or decremented. This may occur any number of times during a time unit. Transitions can be generated at any time as long as both the *Ticker* and the *Doer* are in their initial states. Invariants are used in both templates to make sure the time synchronization is always at the right time.

When running a contract with the *Ticker* and *Doer* set up it is possible to set options for the generation of transition steps. The parameters that can be set are the value that the variables are incremented/decremented with, limits for how high or low the variables can be set, the time limit for the simulation mentioned above, and whether or not an action can be generated more than once.

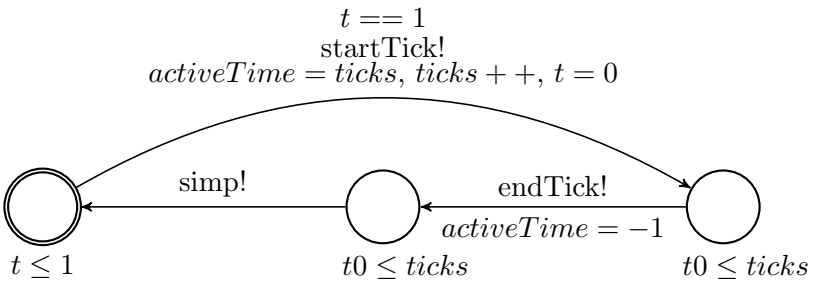



Figure 3.6: *Ticker* template, where `ticks` and `activeTime` are variables, $t$ is a local clock and `startTick` and `endTick` are broadcast channels

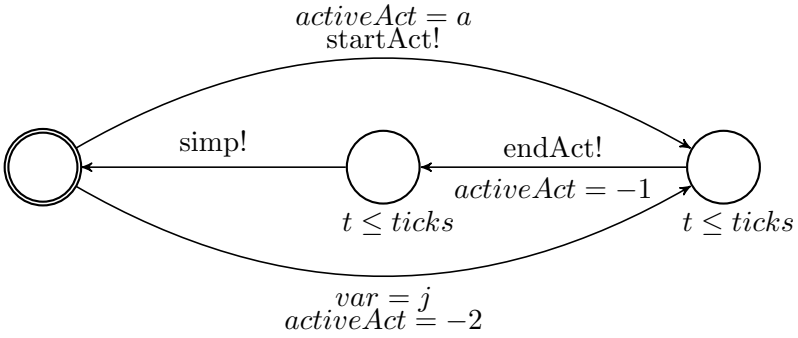



Figure 3.7: *Doer* template, where $a$ is an action, $j$ is an integer or a variable, `activeAct`, `ticks` and *var* are variables, t is a local clock and `simp`, `startAct` and `endAct` are broadcast channels

### 3.1.3 Transition States

Since templates in UPPAAL communicate through channels a time or an action synchronization happens at an instance and if a template is not listening then it will miss the signal. This is quite limiting and makes it hard to represent the conditions in the SCL. To get more flexibility the `ticks` variable, an active action and an active time variable (`activeAct` and `activeTime`) is used. The active variables are used to represent active transition states. The active action variable is -1 if there is no action step taking place. During an action step it has the value of the index for the action taking place. For observation steps `activeAct` is set to -2 since variables do not have unique index values, like actions do. The active time variable is also set to -1 when inactive and to the current time when active. Figure 3.5, Figure 3.7 and Figure 3.6 show the active variables being set to different values.

Each transition step has a couple of channels, one start and one end channel. These are broadcast channels, used to signal the start and the end of a transition step. They are listened for on edges in the templates to progress between locations at the right time. Since broadcast channels are used they can be listened for by any number of receivers.

In between every time or action transition a synchronization is sent on a simplification channel (`simp`). This is used for evaluations and progress that needs to be made between active states.

### 3.1.4 Example Translations

Some of the templates are illustrated in this section to show how the different channels and states are used to get the desired behavior.

#### Rep

The *Rep* template, seen in Figure 3.8, starts by activating its first clause, $j$. If $j$ is violated a new clause is started that may still satisfy the *Rep* template. If the first clause, $j$, violates during an active state, the clause enters an idle state until the active state is over. To progress when there is no active state the `simp` channel synchronization is listened for. When there is no active state the second clause, $r$, is activated.

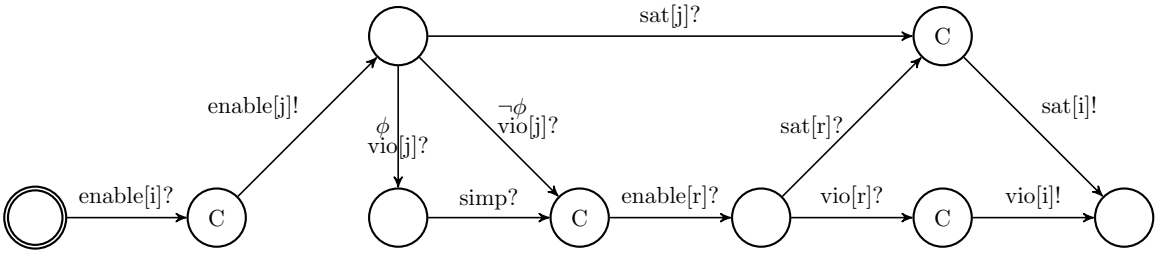



Figure 3.8: Rep template, $Rep\langle r, j\rangle$, where $\phi$ is true if there is an active state, $j$ and $r$ are indexes, `simp` is a broadcast channel and `enable`, `sat` and `vio` are lists of channels

#### At

The *At* clause enables a sub clause, $j$, and waits for a response. When `t` time units have passed the clause returns the response from clause $j$ or violates since there is no more time to wait for a response. Figure 3.9 shows how this is done in UPPAAL (the nodes in the figure have been numbered to make them easier to refer to). Node 4 is an idle state waiting for a response from the $j$ template. If a response is given before `t` time units have passed the template moves on to node 6 or 8, depending on the response. These are waiting nodes where the template is paused until `t` time units have passed.

If there is no response from template $j$ and time is about to run out node 3 is entered. Here we may wait for a response until the invariant, $t0 \leq ticks$, is about to become invalid. If no response is given the clause is violated. An invariant is used here so that the template may wait for responses from templates with a higher priority.

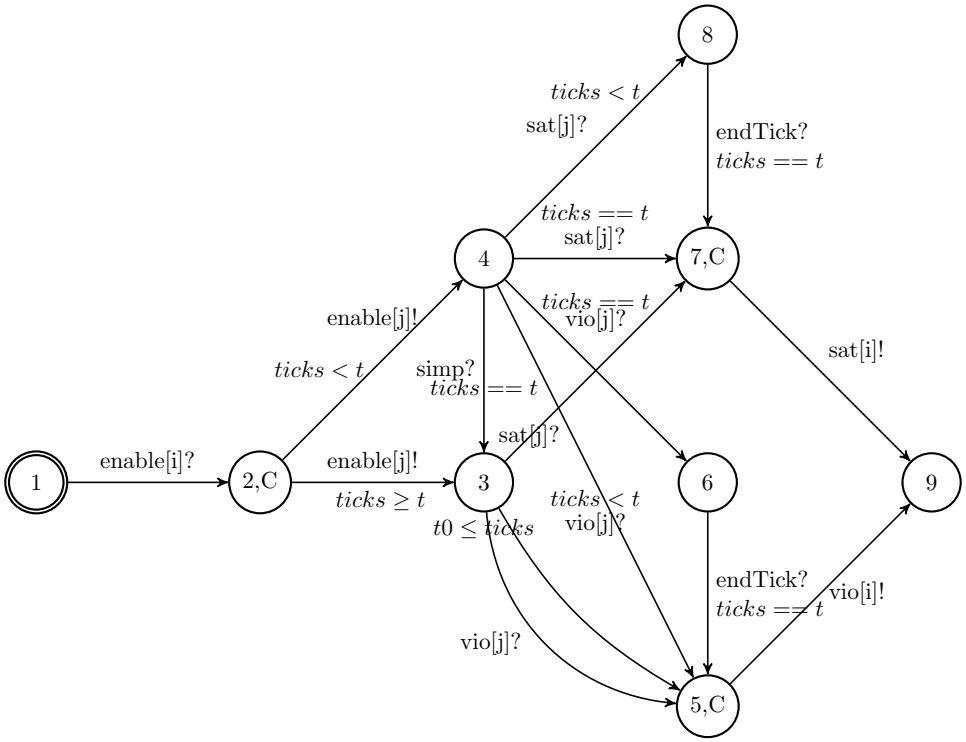



Figure 3.9: At template, $At\langle t, j\rangle$, where $t$ is an absolute time value, $j$ is an index, $t0$ is the global clock, `ticks` is a variable, `simp` and `endTick` are broadcast channels and `vio`, `sat` and `enable` are lists of channels. (The numbers in the nodes are there to make referencing easier)

### When

*When* checks at the start and after every step if the guard, $G$, is satisfied. If the guard is satisfied it starts the next clause on the next transition step. The When template can be seen in Figure 3.10.

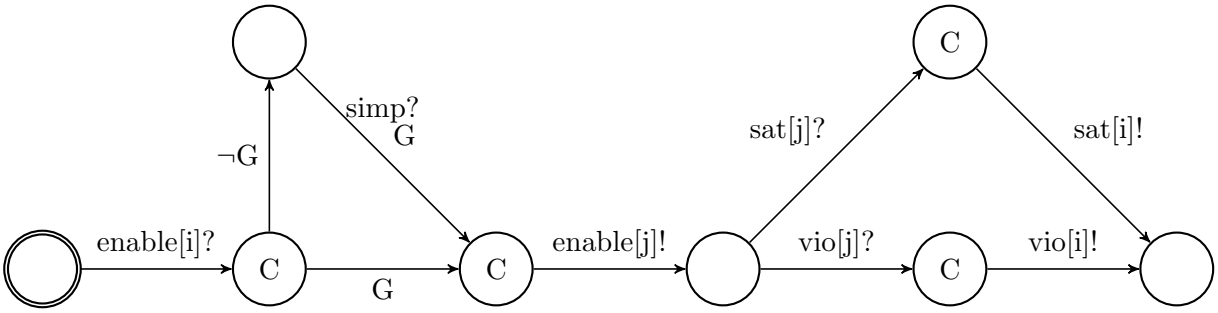



Figure 3.10: When template, $When\langle G, j\rangle$, where $G$ is a guard, $j$ is an index, `simp` is a broadcast channel and `vio`, `sat` and `enable` are lists of channels

## 3.2 Testing

To test the translation two different methods are used. The first is automated property testing, using QuickCheck. The other is a case study based on the real world conditions present in a university course.

### 3.2.1 Automated Property-based Testing

QuickCheck was used to run automated property tests. To use QuickCheck test cases needed to be generated. A contract is generated arbitrarily by recursively choosing clauses until all leaf nodes consist of base cases. If a clause in the contract includes a time, like *After* or *Within*, this time is assigned by choosing a random value between 0 and 20. If a clause with a guard is chosen, the guard is generated arbitrarily by the same method as with contracts. For a contract to be valid there are a couple of conditions that must hold. All names should be unique and when a guard has a condition including a name, that name needs to be defined in the contract. In order to evaluate a contract a trace is also generated. The generated trace consists only of steps with actions and variables present in the contract. These steps are randomly placed between 0 and 40 time units. A test case consists of both a contract and a trace.

There are shrinking functions for both traces and contracts. To minimize traces transitions are removed or moved to take place at an earlier time. For contracts, clauses are removed and time limits lowered. This reduces the counterexamples and makes it easier to understand what conditions lead to a failed test.

The generated test cases are translated from SCL to timed automata and run in UPPAAL. The run also includes a query stating that all start clauses should be satisfied (unevaluated clauses are regarded as unsatisfied). The run results in a Boolean value that is true if the query is satisfied and false otherwise. This result is compared with the result from running the same test case with an implementation of the SCL operational semantics. The property used can be seen in Figure 3.11 and the results from running the test is presented in Section 4.1.

```
prop_TraceSemantics :: Contract -> Trace -> QuickCheck.Property
prop_TraceSemantics contract trace =
    let
        sclResult = evalWithSCLSemantics contract trace
        uppaalResult = checkWithUPPAAL contract trace
    if sclResult == uppaalResult
    then True
    else False
```

Figure 3.11: Property for QuickCheck testing

### 3.2.2 Case Study

To test the translation on a real world problem a contract modelled around the conditions present in a university course was constructed. It models things like deadlines for registrations and assignments, grading of assignments, grading of exams and conditions for passing the course.

#### Course contract

All clauses in the course contract are listed below.

- The course starts at day 0.
- Students need to register for the course before the registration deadline, 1 week after the course have started.
- Students need to sign up for the exam before exam registration deadline, at day 45.
- The first deadline for assignment 1 is at day 10. If the assignment is not accepted the student have until final deadline at day 25 to improve the solution and submit it again.
- The first deadline for assignment 2 is at day 30. If the assignment is not accepted the student have until final deadline at day 48 to improve the solution and submit it again.
- Assistants have 7 days from the deadline to correct an assignment
- The exam is at day 60.
- The examiner has three weeks to correct the exams.
- To pass the course the student needs to pass all of the assignments and get a passing grade on the exam. The grade needs to be registered before day 90.

These conditions are represented in the SCL contract, in Figure 3.12.

$$
\begin{aligned}
&\big[\mathit{Named}\langle \mathsf{InCourse}, \mathit{Before}\langle 8, P\langle \mathsf{regCourse}\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{RegisteredExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Before}\langle 45, P\langle \mathsf{regExam}\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{Ass1}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle \mathit{At}\langle 11, O\langle \mathsf{submit1}\rangle\rangle, \mathit{Rep}\langle\\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 26, O\langle \mathsf{resubmit1}\rangle\rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept1}\rangle\rangle\rangle,\\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept1}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{Ass2}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle \mathit{At}\langle 31, O\langle \mathsf{submit2}\rangle\rangle, \mathit{Rep}\langle\\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 49, O\langle \mathsf{resubmit2}\rangle\rangle, \mathit{Within}\langle 7, O\langle \mathsf{accept2}\rangle\rangle\rangle,\\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept2}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{RegisteredExam}), \mathit{After}\langle 60, \mathit{Seq}\langle\\
&\qquad \mathit{Within}\langle 1, P\langle \mathsf{takeExam}\rangle\rangle,\\
&\qquad \mathit{Within}\langle 21, O\langle \mathsf{passExam}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassCourse}, \mathit{Before}\langle 90, \mathit{When}\langle \mathsf{sat}(\mathsf{PassExam}) \wedge \mathsf{sat}(\mathsf{Ass1}) \wedge \mathsf{sat}(\mathsf{Ass2}), \top\rangle\rangle\rangle\big]
\end{aligned}
$$

Figure 3.12: Course contract written in the SCL

A minimal version of this contract was also used. In it all times in between deadlines were minimized. It includes just one assignment and has the assumption that the student takes the exam if registered for it. When assuming that the exam is taken, the $takeExam$ action is removed. The minimized contract can be seen in Figure 3.13. This minimal contract was used to reduce the model size in order to make verification faster. This is discussed further in Section 4.2.2.

$$
\begin{aligned}
&\big[\mathit{Named}\langle \mathsf{InCourse}, \mathit{Before}\langle 1, P\langle \mathsf{regCourse}\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{RegisteredExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Before}\langle 3, P\langle \mathsf{regExam}\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{Ass1}, \mathit{When}\langle \mathsf{sat}(\mathsf{InCourse}), \mathit{Seq}\langle \mathit{At}\langle 2, O\langle \mathsf{submit1}\rangle\rangle, \mathit{Rep}\langle\\
&\qquad \mathit{Seq}\langle \mathit{Before}\langle 4, O\langle \mathsf{resubmit1}\rangle\rangle, \mathit{Within}\langle 1, O\langle \mathsf{accept1}\rangle\rangle\rangle,\\
&\qquad \mathit{Within}\langle 7, O\langle \mathsf{accept2}\rangle\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassExam}, \mathit{When}\langle \mathsf{sat}(\mathsf{RegisteredExam}), \mathit{After}\langle 4, \mathit{Within}\langle 1, O\langle \mathsf{passExam}\rangle\rangle\rangle\rangle\rangle,\\
&\ \mathit{Named}\langle \mathsf{PassCourse}, \mathit{Before}\langle 6, \mathit{When}\langle \mathsf{sat}(\mathsf{PassExam}) \wedge \mathsf{sat}(\mathsf{Ass1}) \wedge \mathsf{sat}(\mathsf{Ass2}), \top\rangle\rangle\rangle\big]
\end{aligned}
$$

Figure 3.13: Minimized course contract written in the SCL

### Unit testing

To check that the contract's properties hold after the translation to timed automata, two different kinds of unit tests are used. The first kind uses a trace, a query and a Boolean variable stating whether or not the trace should satisfy the query. To execute a test the trace and the regular course contract are translated and run in UPPAAL with the query. The result from verifying the query is then be compared with the expected

outcome to see if the test was successful. An example of a test case for the first kind of tests can be seen in Table 3.1.

| | |
|---|---|
| **Trace** | [ 7 : regCource, 10 : submit1, 18:acceptedAss1, 30 : submit2, 44 : regExam, 48 : reSubmit2, 55 : acceptedAss2, 60 : passExam, takeExam ] |
| **Query** | E◊ PassCourse = SAT |
| **Expected result** | False |
| **Comment** | Approving the exam the same day as, but before, the exam is taken leads to not passing the exam |

Table 3.1: Example of a test case using the *Trace* set up

The second kind of test is based on queries that should always hold. These queries are run in UPPAAL using the *Doer* and *Ticker* set up. By doing this it is possible to check that the query holds no matter what order or when the action transitions take place. An example can be seen in Table 3.2. For this type of testing the minimized contract was used. Since all possible step combinations are generated when verifying with this set up the state space can become quite big. That is why minimizing the time span and the number of actions in the contract will limit the possible states and speed up the verification process.

| | |
|---|---|
| **Query** | $\neg$ (InCourse = SAT) $\Longrightarrow$ $\neg$ (RegisteredExam = SAT) |
| **Comment** | If InCourse is not satisfied it shouldn't be possible to be registered for the exam |

Table 3.2: Example of a test case, without a trace, using the *Doer* and *Ticker* set up

Listed below are the behaviors that were tested by the test cases. All these are present in different test cases that can be seen in Appendix B.

- The effect of named contracts on other clauses. For example that the value of sat(InCourse) leads to the expected behavior for the rest of the clauses.
- Actions taking place after deadlines, too early, or not at all.
- Actions taking place at the right time for satisfying clauses.
- Actions taking place at the right time but in the wrong order.

# 4. Results and Discussion

In the beginning of this section the results are presented and discussed. This is followed by a conclusion, a more general discussion and some ideas for future work.

## 4.1 Automated Property-based Testing

The QuickCheck property, comparing the results from evaluating test cases with the SCL semantics to translating and verifying the contracts in UPPAAL, passed one hundred thousand tests. It took about 13 hours to complete this test run on the hardware specified in Table 4.2. Figure 4.1 shows the output and distribution. Negative test cases are the ones where the contract was not satisfied and in the positive cases all subcontracts where satisfied by the trace in the test case.

```
+++ OK, passed 100000 tests:
73% Negative
26% Positive
```

Figure 4.1: Test result from the automated property testing

When generating contracts and traces for the test cases there are quite a lot of different combinations. It may take a lot of tries to generate a test that tests a certain property and it is not possible to test the entire test space since the number of combinations are infinite. However, when using this test method during development, a lot of special cases resulting in a failed test were discovered. Most of the issues found were discovered within the first forty thousand tests. Before the final design was regarded as properly tested by this method, it executed one hundred thousand tests without any failed test cases.

## 4.2 Case Study

### 4.2.1 Trace Tests

All unit test cases based on traces and expected outcome were satisfied. A short section from the end of the output of the test run can be seen in Figure 4.2. The whole test result output can be seen in appendix B.

```
--- 30 ---
Accepting exam same day as but before exam is taken -> not
    passing exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, passExam.60, takeExam
    .60]
Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 31 ---
Accepting exam same day as but after exam is taken -> passed
    exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .60]
Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is satisfied.

  Test: Sat

--- 32 ---
Late grading exam -> not passing exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .81]
Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

Passed: 32 / 32
```

Figure 4.2: End of the output from case study testing with the *Trace* set up

### 4.2.2 Doer and Clock tests

All queries used for testing the course contract with the *Doer* and *Ticker* set up were satisfied. The end of the output from running these tests can be seen in Figure 4.3, the full output is shown in in appendix B.

```
Verifying formula 18: done[passExam] == -1 --> PassExam.status
    != SAT
 -- Formula is satisfied.

Verifying formula 19: done[passExam] <= 3 --> PassExam.status
   != SAT
 -- Formula is satisfied.

Verifying formula 20: E<> done[passExam] > 4 && PassExam.
   status != SAT
 -- Formula is satisfied.

Verifying formula 21: E<> done[passExam] == 4 && PassExam.
   status == SAT
 -- Formula is satisfied.

Passed: 21 / 21
```

Figure 4.3: End of the output from case study testing without a trace, using the *Doer* and *Ticker* set up

#### Minimized Course Contract

Minimizing the times in the course contract will not affect the results of the test. The Doer may generate an action at any time which means that the outcome when verifying a query will not change when changing the window for an action being listened for from several time units to 1. Using only 1 assignment does not limit the properties that can be tested. Since both assignments have the same conditions the same kind of property will hold for both or neither of the assignments. Assuming that the exam is taken somewhat limits the properties that can be tested.

Making these minimizations was necessary in order to limit the state space to speed up the verification in order to enable running the tests on the available computer. In Table 4.1 the memory and CPU usage during a query validation using the regular full course contract and the minimized course contract can be seen. When comparing the data in the table it is clear that using a minimized contract limits the memory usage needed for validating queries. The CPU TIME is lower in the minimized case, this is because the verification finishes shortly after, so no later state could be documented. Over the course of time, when using the full course contract, memory usage grows

continually and CPU usage decreases until the process either slows down the whole system or is terminated.

| Contract | Current CPU % | Current MEM % | CPU TIME min | Result |
|---|---|---|---|---|
| Full | 34,6 | 80,7 | 2:35 | No result |
| Minimized | 99,8 | 16,1 | 1:03 | Result |

Table 4.1: Memory and CPU usage data from validating queries with different course contracts

Even only reintroducing the action of taking the exam to the minimized contract affects the number of states enough so that no results can be produced, with the available computer. The hardware specifications of the computer used for testing can be seen in Table 4.2.

| **CPU** | |
|---|---|
| Version | Intel® Core™ i5-3317U CPU @ 1.70GHz |
| Architecture | x86_64 |
| Threads per core | 2 |
| Cores per socket | 2 |
| Sockets | 1 |
| BogoMIPS | 3392.40 |
| L1d cache | 32 K |
| L1i cache | 32 K |
| L2 cache | 256 K |
| L3 cache | 3072 K |
| **Memory** | |
| Size | 6 GiB |
| Description | SODIMM DDR3 Synchronous 1333 MHz (0,8 ns) |

Table 4.2: CPU and memory specifications of the computer used for testing

## 4.3 Conclusion

The report presents a compositional design of timed automata that follows the behavioral semantics of the SCL. The design has been tested by a large number of automatically generated test cases and examined through a case study. This fulfills the objective of the thesis. The translation has successfully been used to model a real Contract. A drawback with the timed automata design is that it requires a lot of computer memory for

validations when randomly generating actions, which limits its usefulness. A summary of the contributions is listed below.

- Automatic translation of the SCL to UPPAAL timed automata
- The translation is shown to preserve the behavioral operational semantics via testing
- Application to a realistic contract

### 4.3.1 Limitations

SCL is very useful when it comes to modeling contracts with timing constraints. There are lots of variations on how to express them and different combinations of them are easily evaluated by UPPAAL. However most real world contracts include a limited amount of time constraints which means they would not really take advantage of the expressiveness in SCL.

Another limitation to the usefulness is that validating contracts with the *Doer* and *Ticker* set up is limited to contracts with few variables, as can be seen in the case study. Only using traces when validating a contract restricts the test runs to the users imagination and also requires time and effort to assemble. An alternative could be to use QuickCheck and generate arbitrary traces when testing a query. This method is not as thorough as using the *Doer* generator since testing cannot cover all possibilities, only a number of arbitrarily generated ones.

## 4.4 Future work

To develop the work in this thesis further a formal proof can be constructed that proves that the behavioral semantics is preserved through the translation. This would require considering the translation of each constructor and arguing that its corresponding automaton describes a set of UPPAAL traces which correspond to a sequence of SCL steps.

Another direction for development would be to extend the SCL language with for example real time or repetitive behavior. Using real time in SCL would remove the need to simulate discrete time in UPPAAL, which might limit the number of states and speed up the query verifications. Repetitive behavior is something that appears in contracts, for example when something should be done once a month. Being able to model this would further extend the usefulness of the system. Any extension of SCL would require extending or modifying the translation as well.

Comparing this work with previous work on analysis of contracts is another possible extension. For example comparing the verification efficiency in UPPAAL when using CO-diagrams [10]. Another possibility would be to implement a translation from SCL to Timed Petri-Nets [13] or a SAT-solver [20] and compare the results. Timed Petri-Nets are similar to timed automata but they might allow for a neater translation of

the SCL. SAT-solvers verify SAT problems, which have entirely different structure from timed automata. Therefore this would require a vastly different translation. For example queries would need to be encoded as part of the SAT problem. The advantage with this approach is that a lot of work has been done on developing fast SAT-solvers [20], which might be possible to leverage and get a much faster verification system.

# References


[1] Chavalarias D. The unlikely encounter between von Foerster and Snowden: When second-order cybernetics sheds light on societal impacts of Big Data. Big Data & Society. 2016;3(1).

[2] APPS S. Terms of Service; Didn't Read; 2016. Accessed 2016-04-30. `https://tosdr.org/`.

[3] License CCANSI. Privacy Icons; 2016. Accessed 2016-04-30. `https://disconnect.me/icons`.

[4] Camilleri J J. Analysing normative contracts: On the semantic gap between natural and formal languages [Licentiate thesis]. Chalmers University of Technology and University of Gothenburg. Gothenburg, Sweden; 2015.

[5] Bengtsson J, Yi W. Timed Automata: Semantics, Algorithms and Tools. Uppsala: Uppsala Universitet; 2004. Available from: `http://www.win.tue.nl/~pcuijper/docs/QEES/TA/timed-automata-intro.pdf`.

[6] Bozga M, Daws C, Maler O, Olivero A, Tripakis S, Yovine S. Kronos: A Model-Checking Tool for Real-Time Systems. Instituto de Computacion, Universidad de la Republica; 1998.

[7] Laroussinie F, Larsen K G. Formal Description Techniques and Protocol Specification, Testing and Verification. In: CMC: A Tool for Compositional Model-Checking of Real-Time Systems. vol. 6 of IFIP — The International Federation for Information Processing. Springer US; 1998. p. 439–456.

[8] UP4ALL AB. GUI REFERENCE; 2012. Accessed 2016-03-30. `http://www.uppaal.com/index.php?sida=216&rubrik=101`.

[9] Prisacariu C, Schneider G. CL: An Action-based Logic for Reasoning about Contracts. In: WOLLIC 2009. vol. 5514 of LNCS. Springer; 2009. p. 335–349.

[10] Camilleri J J, Paganelli G, Schneider G. A CNL for Contract-Oriented Diagrams. In: CNL 2014. vol. 8625 of LNCS. Springer; 2014. p. 135–146.

[11] Soliman D, Thramboulidis K, Frey G. Transformation of Function Block Diagrams to UPPAAL timed automata for the verification of safety applications. Annual Reviews in Control. 2012;36(2):p.338–345.

[12] Gong S. A Translation Method from Time Petri Nets to Timed Automata. In: International Conference on Convergence Information Technology. ACM; 2012. p. 20–24.

[13] Cicirelli F, Furfaro A, Nigro L. Model checking time-dependent system specifications using Time Stream Petri Nets and Uppaal. Applied Mathematics and Computation. 2012;218(16):p.8160–8186.

[14] Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. ACM; 2000. p. 268 – 279.

[15] Alur R, Dill D. A theory of timed automata. Theoretical Computer Science. 1994;126:p.183–236.

[16] Bengtsson J, Christensen P, Jensen P, Larsen G Kim, Larsson F, Pettersson P, et al. UPPAAL: a tool suite for validation and verification of real-time systems. Chalmers University of Technology and University of Gothenburg; 1996.

[17] David A M, Möller O, Yi W. Verification of UML statechart with real-time extensions. Department of Information Technology, Uppsala University; 2003.

[18] Hughes J. QuickCheck: An Automatic Testing Tool for Haskell;. Available from: `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`.

[19] Hughes J. Specification based testing with QuickCheck. In: Formal Methods in Computer-Aided Design (FMCAD). Austin, TX: IEEE; 2011. p. 17.

[20] Lyde S, Might M. Trends in Functional Programming. In: McCarthy J, editor. Control-Flow Analysis with SAT Solvers. vol. 8322 of Lecture Notes in Computer Science. Springer Berlin Heidelberg; 2014. p. 125–133.

# A. Simplified Contract Language

## A.1 Syntax

A contract, *Contract*, is defined as a list of clauses, $C$. Clauses are recursively defined so each clause in a contract may be a tree of clauses.

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

An action ($a$), is used to represent an event that may take place; $\Sigma$ is the integer set of actions. $O\langle a\rangle$ means that there is an obligation for action $a$ to take place. $\mathcal{V}$ is a set of variables and $\mathcal{N}$ is a set of names. We assume these sets, $\Sigma$, $\mathcal{V}$ and $\mathcal{N}$, are disjoint and global. $\mathbb{T}_r$ represent relative temporal values while $\mathbb{T}_a$ represent an absolute time stamp. Both these values are treated as natural numbers. There is no clause for negating

subcontracts. To express that something is not allowed the prohibition clause $F\langle a \rangle$ can be used.

## A.2 Structural Operational Semantics

**Steps** Progress is built up by steps. There are 3 kinds of steps, action, observation and delay steps. During an action step either an *action* takes place or an *observation* is made. During an observation a variable is updated. A delay step increases the time with 1 time unit (the semantics operate in discrete time). Clauses can also be evaluated in between transitions through simplifications.

A step is either:

- An action $a$: $\xrightarrow{a}$ or an observation $\xrightarrow{v=2}$
- A delay of 1 time unit: $\rightsquigarrow^1$

We use the arrow $\xrightarrow{\cdot}$ to mean either kind of step.

**Environment** There is an environment $\Gamma$ which contains:

- a map from actions to timestamps ($\Sigma \mapsto \mathbb{T}_a$)
- a map from names to clauses ($\mathcal{N} \mapsto C$)
- a map from variables to integers ($\mathcal{V} \mapsto \mathbb{Z}$)
- variable $t_0$ which stores current time in ticks.

The environment is updated as follows:

- action step $\xrightarrow{a}$ updates an integer variable named $a$: $\Gamma[a := t_0]$ (record the time the action was done)
- delay step $\rightsquigarrow^1$ increments an absolute clock $t_0$: $\Gamma[t_0 \mathrel{+}= 1]$

**Guards** Guards are defined as predicates over the environment $\Gamma$:

- $\mathsf{done}(a)$ is true if action $a$ has occured ($\Gamma[a] > -1$).
- $\mathsf{sat}(n)$ is true if name $n$ is mapped to $\top$ ($\Gamma[n] = \top$).
- $\mathsf{earlier}(t)$ is true iff $\Gamma[t_0] < t$ and $\mathsf{later}(t)$ is true iff $\Gamma[t_0] > t$.

### A.2.1 Contract

1. All clauses in the list are processed together
2. Environment is shared

### A.2.2 Top, bottom

Top is trivially satisfiable. Bottom is trivially unsatisfiable.

### A.2.3 Named

Label a clause with a name so that it can be used in a guard.

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

### A.2.4 Obligation

Action must be performed.

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

### A.2.5 Permission

Action may or may not be performed.

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

### A.2.6 Forbiddance (Prohibition)

Action must not be performed.

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

### A.2.7 Declaration (Assignment)

Assign a value to a variable. Evaluated immediately (no step).

$$\mathit{Decl}_{Var}\frac{D\langle v, x\rangle}{\top}\Gamma[v := \Gamma[x]] \qquad \mathit{Decl}_{Int}\frac{D\langle v, i\rangle}{\top}\Gamma[v := i]$$

### A.2.8 And refinement

Conjunction.

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

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

### A.2.9 Or refinement

Disjunction.

$$Or_{\text{Thru}} \frac{C_1 \dot{\to} C_1' \qquad C_2 \dot{\to} C_2'}{Or\langle C_1, C_2\rangle \dot{\to} Or\langle C_1', C_2'\rangle}$$

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

### A.2.10 Seq refinement

Sequence.

$$Seq_{\text{Thru}} \frac{C_1 \dot{\to} C_1'}{Seq\langle C_1, C_2\rangle \dot{\to} Seq\langle C_1', C_2\rangle}$$

$$Seq_{\text{Top}} \frac{Seq\langle C_1, C_2\rangle}{C_2} \mathsf{isTop}(C_1) \qquad Seq_{\text{Bot}} \frac{Seq\langle C_1, C_2\rangle}{\bot} \mathsf{isBot}(C_1)$$

### A.2.11 Reparation

Note: the first clause, $C_r$, is the reparation.

$$Rep_{\text{Thru}} \frac{C \dot{\to} C'}{Rep\langle C_r, C\rangle \dot{\to} Rep\langle C_r, C'\rangle}$$

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

### A.2.12 Wait/After

#### Wait

Wait a relative amount of time.

$$\mathit{Wait}_1 \frac{}{\mathit{Wait}\langle 1, C\rangle \rightsquigarrow^1 C} \qquad \mathit{Wait}_0 \frac{\mathit{Wait}\langle 0, C\rangle}{C}$$

$$\mathit{Wait}_{Del} \frac{}{\mathit{Wait}\langle z, C\rangle \rightsquigarrow^1 \mathit{Wait}\langle z-1, C\rangle} z > 1$$

#### After

Wait until an absolute lower time bound. The bound must be checked straight away (not only after a delay step).

$$\mathit{After} \frac{\mathit{After}\langle t, C\rangle}{C} \Gamma \vdash t_0 \geq t$$

### A.2.13 Within/Before

#### Within

Inner clause must be satisfied within a relative amount of time.

$$\mathit{Within}_{\text{Thru}} \frac{C \xrightarrow{x} C'}{\mathit{Within}\langle z, C\rangle \xrightarrow{x} \mathit{Within}\langle z, C'\rangle}$$

$$\mathit{Within}_{Del} \frac{C \rightsquigarrow^1 C'}{\mathit{Within}\langle z, C\rangle \rightsquigarrow^1 \mathit{Within}\langle z-1, C'\rangle} z \geq 1$$

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

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

**Before**

Inner clause must be satisfied before an absolute upper bound timestamp.

$$\mathit{Before}_{\mathit{Thru}}\frac{C \stackrel{\cdot}{\to} C'}{\mathit{Before}\langle t, C\rangle \stackrel{\cdot}{\to} \mathit{Before}\langle t, C'\rangle}\Gamma \vdash t_0 < t$$

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

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

Note that:

$$\mathit{Before}\langle t, C\rangle \not\equiv \mathit{When}\langle \mathsf{earlier}(t), C\rangle$$

because *Before* will fail with $\bot$ on expiry.

### A.2.14 In/At

**In**

Like *Within* but we always wait until all time has passed before looking at result. You must satisfy the inner clause before the expiry, otherwise the whole thing fails.

$$\mathit{In}_{\text{ThruAct}}\frac{C \xrightarrow{x} C'}{\mathit{In}\langle z, C\rangle \xrightarrow{x} \mathit{In}\langle z, C'\rangle}z \geq 1 \qquad \mathit{In}_{\text{ThruDel}}\frac{C \rightsquigarrow^1 C'}{\mathit{In}\langle z, C\rangle \rightsquigarrow^1 \mathit{In}\langle z-1, C'\rangle}z \geq 1$$

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

**At**

An absolute version of *In* (the waiting version of *Before*).

$$\mathit{At}_{\mathit{Thru}}\frac{C \stackrel{\cdot}{\to} C'}{\mathit{At}\langle t, C\rangle \stackrel{\cdot}{\to} \mathit{At}\langle t, C'\rangle}\Gamma \vdash t_0 \leq t$$

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

### A.2.15 When

Guard which never expires.

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

### A.2.16 WhenWithin / WhenBefore

#### WhenWithin

A guard which expires (with $\top$) after a certain amount of relative time.

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

$$\mathit{WhenWithin}_{Exp}\frac{}{\mathit{WhenWithin}\langle 0, G, C\rangle \rightsquigarrow^{1} \top}\Gamma \nvdash G$$

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

#### WhenBefore

A guard which expires (with $\top$) after an absolute timestamp.

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

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

## A.3 Predicates

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

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

$$\mathsf{notTop}(C) := \neg\mathsf{isTop}(C)$$

$$\mathsf{notBot}(C) := \neg\mathsf{isBot}(C)$$

# B. Case Study Results

## B.1 Unit Test Cases using Trace set up

```
--- 1 ---
All actions just in time, passed course should be satisfied

[regCourse.7, submit1.10, acceptedAss1.17, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is satisfied.

  Test: Sat

--- 2 ---
All actions in time, all nemed clauses should be satisfied

[regCourse.7, submit1.10, acceptedAss1.17, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> PassCourse.status == SAT && (PassExam
    .status == SAT && (InCourse.status == SAT && (
    RegisteredExam.status == SAT && (Ass1.status == SAT && Ass2
    .status == SAT))))
 -- Formula is satisfied.

  Test: Sat

--- 3 ---
Missed regExam deadline -> not passed exam.

[regCourse.7, submit1.10, acceptedAss1.17, submit2.30, regExam
    .45, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]
```

```
Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 4 ---
RegExam before in course -> not passed exam.

[regExam.7, regCourse.7, submit1.10, acceptedAss1.17, submit2
    .30, regExam.45, reSubmit2.48, acceptedAss2.54, takeExam
    .60, passExam.80]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 5 ---
RegExam before in course -> not registered for exam.

[regExam.4, regCourse.7, submit1.10, acceptedAss1.17, submit2
    .30, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> RegisteredExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 6 ---
No regExam action -> not registered for exam.

[submit1.10, acceptedAss1.17, submit2.30, regExam.45,
    reSubmit2.48, acceptedAss2.54, takeExam.60, passExam.80]


Verifying formula 1: E<> RegisteredExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 7 ---
Late submission of ass1 -> not passed ass1
```

```
[regCourse.7, submit1.11, acceptedAss1.17, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 8 ---
Late grading of ass1 and no resubmission -> not passed ass1

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 9 ---
Grading of ass1 same time step after -> not passed ass1 (At
    moves on first when deadline for submission expires)

[regCourse.7, submit1.10, acceptedAss1.10, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 10 ---
Grading of ass1 1 time step after submission -> passedAss1

[regCourse.7, submit1.10, acceptedAss1.11, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is satisfied.

  Test: Sat
```

```
--- 11 ---
Grading of ass1 same time step but before submission -> not
    passed ass1

[regCourse.7, acceptedAss1.10, submit1.10, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.54, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 12 ---
No submission of assignment 1 -> not passed ass1

[regCourse.7, acceptedAss1.17, submit2.30, regExam.44,
    reSubmit2.48, acceptedAss2.54, takeExam.60, passExam.80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 13 ---
Not passed assignment 1 -> not passed course

[regCourse.7, acceptedAss1.17, submit2.30, regExam.44,
    reSubmit2.48, acceptedAss2.54, takeExam.60, passExam.80]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 14 ---
Late first submission of ass2 -> not passed ass2

[regCourse.7, submit1.10, acceptedAss1.18, submit2.31, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is NOT satisfied.
```

```
  Test: Sat

--- 15 ---
No first submission of ass2 -> not passed ass2

[regCourse.7, submit1.10, acceptedAss1.18, regExam.44,
    reSubmit2.48, acceptedAss2.55, takeExam.60, passExam.80]


Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 16 ---
Late second submission of ass2 -> not passed ass2

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.49, acceptedAss2.55, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 17 ---
Late second grading of ass2 -> not passed ass2

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.56, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 18 ---
Immediate secound grading of ass2 -> passed ass2 (Before moves
    on at once)

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.48, takeExam.60, passExam
    .80]
```

```
Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is satisfied.

  Test: Sat

--- 19 ---
Grading of ass2 same timestep as but before subission -> not
    passed ass2

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, acceptedAss2.48, reSubmit2.48, takeExam.60, passExam
    .80]


Verifying formula 1: E<> Ass2.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 20 ---
Not passed ass2 -> not passed course

[regCourse.7, submit1.10, acceptedAss1.18, submit2.31, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .80]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 21 ---
Late registration for course -> not in cource

[regCourse.8, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .80]


Verifying formula 1: E<> InCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 22 ---
No registration for course -> not in cource
```

```
[submit1.10, acceptedAss1.18, submit2.30, regExam.44,
    reSubmit2.48, acceptedAss2.55, takeExam.60, passExam.80]


Verifying formula 1: E<> InCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 23 ---
Late registration for course -> not passed exam

[regCourse.8, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .80]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 24 ---
No registration for course -> not passed ass1

[submit1.10, acceptedAss1.18, submit2.30, regExam.44,
    reSubmit2.48, acceptedAss2.55, takeExam.60, passExam.80]


Verifying formula 1: E<> Ass1.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 25 ---
No registration for course -> not passed course

[submit1.10, acceptedAss1.18, submit2.30, regExam.44,
    reSubmit2.48, acceptedAss2.55, takeExam.60, passExam.80]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 26 ---
Late exam -> not passed exam
```

```
[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.61, passExam
    .80]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 27 ---
Early exam -> not passed exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.59, passExam
    .80]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 28 ---
Not taking exam -> not passed exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, passExam.80]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 29 ---
Taking exam to early -> not passed Course

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.59, passExam
    .80]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 30 ---
```

```
Accepting exam same day as but before exam -> not passing exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, passExam.60, takeExam
    .60]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat

--- 31 ---
Accepting exam same day as but after exam -> passed exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .60]


Verifying formula 1: E<> PassExam.status == SAT
 -- Formula is satisfied.

  Test: Sat

--- 32 ---
Late grading exam -> not passing exam

[regCourse.7, submit1.10, acceptedAss1.18, submit2.30, regExam
    .44, reSubmit2.48, acceptedAss2.55, takeExam.60, passExam
    .81]


Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is NOT satisfied.

  Test: Sat
```

**Passed**: 32 / 32

## B.2 Unit Test Cases using Doer set up

```
Verifying formula 1: E<> PassCourse.status == SAT
 -- Formula is satisfied.

Verifying formula 2: PassExam.status == VIO --> PassCourse.
    status != SAT
 -- Formula is satisfied.
```

```
Verifying formula 3: Ass1.status == VIO --> PassCourse.status
   != SAT
 -- Formula is satisfied.

Verifying formula 4: InCourse.status == VIO --> RegisteredExam
   .status != SAT
 -- Formula is satisfied.

Verifying formula 5: InCourse.status == VIO --> Ass1.status !=
    SAT
 -- Formula is satisfied.

Verifying formula 6: RegisteredExam.status == VIO --> PassExam
   .status != SAT
 -- Formula is satisfied.

Verifying formula 7: done[submit1] == -1 --> Ass1.status !=
   SAT
 -- Formula is satisfied.

Verifying formula 8: done[acceptedAss1] == -1 --> Ass1.status
   != SAT
 -- Formula is satisfied.

Verifying formula 9: E<> done[submit1] == -1 && Ass1.status ==
    SAT
 -- Formula is NOT satisfied.

Verifying formula 10: done[submit1] > 1 --> Ass1.status != SAT
 -- Formula is satisfied.

Verifying formula 11: E<> done[submit1] <= 0 && Ass1.status !=
    SAT
 -- Formula is satisfied.

Verifying formula 12: E<> done[submit1] == 1 && Ass1.status !=
    SAT
 -- Formula is satisfied.

Verifying formula 13: E<> done[submit1] > done[acceptedAss1]
   && Ass1.status != SAT
 -- Formula is satisfied.

Verifying formula 14: done[regCourse] > 1 --> InCourse.status
   != SAT
 -- Formula is satisfied.

Verifying formula 15: done[regExam] == -1 --> RegisteredExam.
```

```
   status != SAT
 -- Formula is satisfied.

Verifying formula 16: E<> done[regExam] <= 0 && RegisteredExam
   .status != SAT
 -- Formula is satisfied.

Verifying formula 17: E<> done[regExam] > done[regCourse] &&
   RegisteredExam.status != SAT
 -- Formula is satisfied.

Verifying formula 18: done[passExam] == -1 --> PassExam.status
    != SAT
 -- Formula is satisfied.

Verifying formula 19: done[passExam] <= 3 --> PassExam.status
   != SAT
 -- Formula is satisfied.

Verifying formula 20: E<> done[passExam] > 4 && PassExam.
   status != SAT
 -- Formula is satisfied.

Verifying formula 21: E<> done[passExam] == 4 && PassExam.
   status == SAT
 -- Formula is satisfied.
```

**Passed**: 21 / 21