Specification and Verification of Normative Texts using C-O Diagrams

Gregorio Díaz, M. Emilia Cambronero, Enrique Martínez, and Gerardo Schneider

Abstract—C-O Diagrams have been introduced as a means to have a more visual representation of normative texts and electronic contracts, where it is possible to represent the obligations, permissions and prohibitions of the different signatories, as well as the penalties resulting from non-fulfillment of their obligations and prohibitions. In such diagrams we are also able to represent absolute and relative timing constraints. In this paper we present a formal semantics for C-O Diagrams based on timed automata extended with information regarding the satisfaction and violation of clauses in order to represent different deontic modalities. As a proof of concept, we apply our approach to two different case studies, where the method presented here has successfully identified problems in the specification.

Index Terms—Normative documents, electronic contracts, deontic logic, formal verification, visual models, timed automata, C-O Diagrams.

1 Introduction

In software context, the term *contract* has traditionally as a metaphor to represent limited kinds of "agreements" between software elements at different levels of abstraction. The first use of the term in connection with software programming and design was by Meyer, in the context of the language Eiffel (*programming-by-contracts*, or *design-by-contract*) [1], and relied on Hoare's notion of pre and post-conditions and invariants. Though this paradigm has proven to be useful for developing object oriented systems, it seems to have shortcomings for novel development paradigms such as service-oriented computing and component-based development. These new applications have a more involved interaction and therefore require a more sophisticated notion of contracts.

As a response, behavioral interfaces have been proposed to capture richer properties more than simple pre and post-conditions [2]. With this specifications, it is possible to express contracts concerning the history of events, including causality properties. However, this approach is limited when it comes to contracts containing exceptional behavior, since the focus is mainly on the interaction concerning expected (and prohibited) behavior.

In the context of Service Oriented Architectures (SOA), there are several service contract specification languages, such as ebXML [3], WSLA [4], and WS-Agreement [5]. These standardized specification languages suffer from one or more of the following problems: they are restricted to bilateral contracts, lack formal semantics (so it is difficult to reason about them), their treatment of functional behavior is rather limited and the sub-languages used to specify security constraints are usually limited to small application-specific domains. The lack of suitable languages for contracts in the context of SOA is a clear conclusion of the survey [6] where a taxonomy is presented.

Some researchers have investigated variants of *deontic logic* [7] to specify different aspects of software systems. Deontic logic is concerned (among other things) with the formalization of moral and legal obligations, permissions, and prohibitions, as well as their interrelation and properties. Formalizing such notions is not an easy task, as witnessed by the extensive research conducted by the deontic community both from the philosophical and the logical point of view [8]. From the computer science and software engineering perspective, however, the focus is on legal and not moral aspects. Therefore, in this paper we use deontic logic as a source of inspiration to define formal languages in order to specify contracts where (legal) obligations, permissions, and prohibitions, as well as events/consequences resulting from violations of

- *Gregorio Díaz, M. Emilia Cambronero and Enrique Martínez are with the Department of Computer Science, University of Castilla-La Mancha, Albacete, Spain.*
  E-mail: {gregorio.diaz, MEmilia.Cambronero, }@uclm.es and enrogeen@gmail.com.
- *Gerardo Schneider is with the Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden.*
  E-mail: gerardo@cse.gu.se

obligations and prohibitions are of importance.

To further close the gap between contracts and its representation, we consider that three criteria must be met: a) the representation must be usable and understandable for non-expert users, b) the logic behind this representation must provide reasoning techniques and c) the internal machine-codification must be easy manipulated by programmer and allow runtime monitoring. We envision two possible ways to accomplish this: i) the development of suitable techniques to get a proper translation from contracts written in natural language into formal languages, or ii) the development of a graphical representation (and tools) to manipulate contracts at a high level, with formal semantics supporting automatic translation into the formal language. In this paper we take the second approach.

Previously we have introduced *C-O Diagrams* [9], a graphical representation not only for electronic contracts (*e-contracts*) but also for the specification of any kind of normative text (Web service composition behavior, software product lines engineering, requirements engineering, etc.). *C-O Diagrams* allow for the representation of complex clauses describing the obligations, permissions, and prohibitions of different signatories (as defined in deontic logic), as well as *reparations* describing contractual clauses in cases of non-fulfillment of obligations and prohibitions. Also, *C-O Diagrams* permit users to define real-time constraints. In [10], some of the authors presented a set of satisfaction rules to check whether or not a timed automaton satisfies a *C-O Diagram* specification. These rules were necessary but not sufficient.

One of the main motivations for using a graphical contract representation is to ease the manipulation and understanding of contracts by non-specialized users. This idea is supported by many authors in this field, including [11] and [12]. To support the above we have presented in [9] an evaluation of *C-O Diagrams* based on user-based tests, with the purpose of comparing the understandability of both *C-O Diagrams* and textual notations for e-contracts. Results showed, at least in this study, that a visual representation model was more usable than its equivalent textual form.

The goal of this paper is to further develop our previous work, in particular we present a formal semantics for *C-O Diagrams* based on timed automata. In order to capture the normative concepts of permission, obligation and prohibition, as well as the penalties in cases of certain violations, we provide an extension of timed automata. This extension is needed, among other reasons, as it is not easy to represent the obligation to do something and the penalty for not doing it (this cannot be simply represented in timed automata as a branching, as this would be understood as an *or* where both branches have the same priority).

We use UPPAAL [13] to implement the obtained timed automata, and its model checker to verify properties about *C-O Diagrams*. As a proof of concept we present two case studies: one in the field of Web service composition and another in the field of requirements engineering.

The paper is structured as follows: background is shown in Section 2, Section 3 presents *C-O Diagrams* and their syntax, Section 4 develops the formal semantics of *C-O Diagrams* and Section 5 explains the implementation of the resulting timed automata in UPPAAL. Section 6 presents a case study of an *Online Auctioning Process* and Section 7 presents a case study of the engineering requirements of an *Adaptive Cruise Control* system. Finally, the related work is discussed in Section 8 and conclusions and future work are commented in Section 9.

2 Background

In this section we define the background formalism used to verify *C-O Diagrams*, namely timed automata, and a subset of Timed Computation Tree Logic (TCTL), which is used by the UPPAAL model checker to define the properties to be verified.

2.1 Timed Automata

A timed safety automaton, or simply *timed automaton* (TA) [14] is essentially a finite automaton extended with real-valued variables. These variables model the logical clocks in the system, and are initialized to zero when the system is started. They then increase their value synchronously as time elapses, at the same rate. In the model there are also clock constraints, which are guards on the edges that are used to restrict the behavior of the automaton, since a transition represented by an edge can only be executed when the clock values satisfy the guard condition. Transitions are not forced to be executed when their guards are true, the automaton can stay at a location without executing any transitions, unless an invariant condition is associated with that location. In this case, the automaton may remain at that same location as long as the invariant condition is satisfied. Additionally, the execution of a transition can be used to reset some clocks.

In what follows we consider a finite set of real-valued variables $\mathcal{C}$ ranging over by $x, y, \ldots$ standing for clocks, a finite set of integer-valued variables $\mathcal{V}$,

ranging over by $v, w, \ldots$ and a finite alphabet Σ ranging over by $a, b, \ldots$ standing for actions. We will use letters $r, r', \ldots$ to denote sets of clocks. We will denote by *Assigns* the set of possible assignments, $Assigns = \{v := expr \mid v \in \mathcal{V}\}$, where *expr* are arithmetic expressions using naturals and variables. Letters $s, s' \ldots$ will be used to represent a set of assignments. A *guard*, or *invariant condition* is a conjunctive formula of atomic constraints of the form: $x \sim n$, $x - y \sim n$, $v \sim n$ or $v - w \sim n$, for $x, y \in \mathcal{C}$, $v, w \in \mathcal{V}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. The set of guards or invariant conditions will be denoted by $\mathcal{G}$, ranging over by $g, g', \ldots$

Definition 1: (Timed Automata) A *timed automaton* is a tuple (N, n_0, E, I), where

- N is a finite set of locations (nodes).
- $n_0 \in N$ is the initial location.
- $E \subseteq N \times \mathcal{G} \times \Sigma \times \mathcal{P}(Assigns) \times 2^{\mathcal{C}} \times N$ is the set of edges, where the subset of *urgent edges* is called $E_u \subseteq E$.
- $I : N \to \mathcal{G}$ is a function that assigns invariant conditions to locations. □

From now on, we will write $n \xrightarrow[s]{g,a,r} n'$ to denote $(n, g, a, s, r, n') \in E$, and $n \xrightarrow[s]{g,a,r}_u n'$ when $(n, g, a, s, r, n') \in E_u$.

The semantics of a timed automaton is defined as a state transition system, where each state represents a location, a clock valuation and a variable valuation. We use the following notation: letters $u, z, \ldots$ will represent clock valuations, i.e., functions that assign non-negative real values to clocks, $u, z : \mathcal{C} \to \mathbb{R}_0^+$. By $u \in g$ we will represent that the clock valuation u makes g to be true, where we assume that when g is empty $u \in g$ is true, and by $u + d$ the clock valuation that takes u and increases the value of every clock by d. Letters $v, w, \ldots$ will be used to represent variable valuation, i.e., functions that assign non-negative integer values to variables, $v, w : \mathcal{V} \to \mathbb{Z}_0^+$.

Definition 2: (Timed Automaton Semantics) Let $\mathcal{A} = (N, n_0, E, I)$ be a timed automaton. The semantics of $\mathcal{A}$ is defined as the *timed labeled transition system* $(Q, q_0, \to)$, where:

- $Q \subseteq N \times {\mathbb{R}_0^+}^{\mathcal{C}} \times {\mathbb{Z}_0^+}^{\mathcal{V}}$ (set of states).
- $q_0 = (n_0, \overline{0}, V_0) \in Q$, is the initial state, where $\overline{0}$ is the clock valuation that assigns every clock to zero and V_0 is the variable valuation that assigns every variable to its initial value.
- $\to \subseteq (Q \times \mathbb{R}_0^+ \times \mathrm{Q}) \cup (\mathrm{Q} \times \Sigma \times \mathrm{Q})$ (delay and action transitions).

Delay transitions are of the form (q, d, q'), for $d \in \mathbb{R}_0^+$, denoted by $q \xrightarrow{d} q'$, and are defined by the following rule:

- $(n, u, v) \xrightarrow{d} (n, u + d, v)$ if and only if $(u + d') \in I(n)$, for all $d' \leq d$, $d' \in \mathbb{R}_0^+$.

Action transitions are of the form (q, a, q'), for $a \in \Sigma$, denoted by $q \xrightarrow{a} q'$, and are defined by the following rule:

- $(n, u, v) \xrightarrow{a} (n', u', v')$ if and only if there is an edge $n \xrightarrow[s]{g,a,r} n'$, such that $u \in g$, $u'(x) = u(x)$ for all $x \notin r$, $u'(x) = 0$, for all $x \in r$, and $u' \in I(n')$. □

A concurrent system is usually modelled by a set of timed automata running in parallel. A *Network of Timed Automata* (NTA) is then defined as a set of timed automata that run simultaneously, using the same set of clocks, and synchronizing on the common actions. Then, we distinguish two types of actions: internal and synchronization actions. Internal actions can be executed by the corresponding automata independently, and they will be ranged over the letters $a, b \ldots$, whereas synchronization actions must be executed simultaneously by two automata. Synchronization actions are ranged over letters $m, m', \ldots$ and come from the synchronization of two actions $m!$ and $m?$, executed from two different automata[1]. The semantics of a network of timed automata is then defined straightforwardly, as a natural extension of Def. 2.

Definition 3: (Semantics of an NTA) Let $\mathcal{A}_i = (N_i, n_{0_i}, E_i, I_i)$, $i = 1, \ldots, k$ be a set of timed automata. A *state* or *configuration* of the NTA $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$, is a tuple $(\overline{n}, u, v)$, where $\overline{n} = (n_1, \ldots, n_k)$, with $n_i \in N_i$, u is a clock valuation for the clocks in the system, $u \in {\mathbb{R}_0^+}^{\mathcal{C}}$, and v is a variable valuation for the variables in the system, $v \in {\mathbb{Z}_0^+}^{\mathcal{V}}$. There are three rules defining the semantics of an NTA:

- $(\overline{n}, u, v) \xrightarrow{d} (\overline{n}, u + d, v)$ (delay rule) if and only if $u + d' \in I_i(n_i)$, for all $i = 1, \ldots, k$ and for all $d' \leq d$, $d' \in \mathbb{R}_0^+$.
- $(\overline{n}, u, v) \xrightarrow{a} (\overline{n'}, u', v')$ (internal action rule) if and only if there is an edge $n_i \xrightarrow[s]{g,a,r} n_i'$, for some $i \in \{1, \ldots, k\}$, such that $n_j' = n_j$, for all $j \neq i$, $u \in g$, $u'(x) = u(x)$ for all $x \notin r$, $u'(x) = 0$, for all $x \in r$, and $u' \in \bigwedge_{h=1,\ldots,k} I_h(n_h')$, and such that $v \in s$, $v'(w) = v(w)$ for all $w \notin s$, $v'(w) = s(w)$, for all $w \in s$.
- $(\overline{n}, u, v) \xrightarrow{m} (\overline{n'}, u', v')$ (synchronization rule) if and only if there exist i, j, $i \neq j$, such that:
  1) $n_h' = n_h$, for all $h \neq i, h \neq j$.
  2) There exist two transitions $n_i \xrightarrow[s_i]{g_i,m!,r_i} n_i'$ and $n_j \xrightarrow[s_j]{g_j,m?,r_j} n_j'$, such that $u \in g_i \wedge g_j$, $u'(x) =$

1. In the original definition the only internal action is τ, and synchronizations always yield internal actions.

$u(x)$, for all $x \notin r_i \cup r_j$, and $u'(x) = 0$, for all $x \in r_i \cup r_j$. And such that, $v \in s_i \wedge s_j$, $v'(w) = v(w)$ for all $w \notin s_i \cup s_j$, $v'(w) = s(w)$ for all $w \in s_i \cup s_j$.

3) $u' \in \bigwedge_{h=1,\ldots,k} I_h(n'_h)$. □

2.2 UPPAAL Model Checker

UPPAAL [13] is a tool for modelling, validation and verification of real-time systems. The validation part is performed by graphical simulations and the verification part by model checking. A system in UPPAAL consists of a network of concurrent processes, each of them modelled as a timed automaton.

Timed automata in UPPAAL have been extended with bounded integer variables and channel synchronization. Bounded integer variables can be read or written by means of expressions labeling the edges, and can be tested in guard conditions. Synchronization channels are declared as *chan c*. An edge labeled with *c!* synchronizes with another labeled *c?*. A synchronization pair is chosen nondeterministically if several combinations are enabled. UPPAAL also defines broadcast channels, which are declared as *broadcast chan c*. In a broadcast synchronization one sender *c!* can synchronize with an arbitrary number of receivers *c?*. If there are no receivers, then the sender can still execute the *c!* action, i.e. broadcast sending never blocks. Another kind of channel defined by UPPAAL are the urgent synchronization channels, which are declared as *urgent chan c*. Delays may not occur if a synchronization transition on an urgent channel is enabled.

A state of an NTA in UPPAAL is defined by the locations of each of the automata, and the clock and variable valuations, as defined in subsection 2.1. Thus, a UPPAAL system is modelled as a network of several timed automata in parallel, a set of variables, a set of clocks, which are part of the state, a set of channels and the variable types, that is $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$.

Another useful feature of UPPAAL are *templates*, that are parameterized generic declarations of timed automata which can later be instantiated.

The simulation step in UPPAAL consists of running the system to check that it works properly in normal conditions. Since simulation does not guarantee system correctness completely, we must use the verifier tool to check some properties of the system that are of interest, and that should be ensured in all conditions. For instance, we can check reachability properties, i.e. if a certain state is reachable or not. This is called *model checking* and it is basically an exhaustive search that covers all possible behaviors of the system.

Properties in UPPAAL are written in a formal language which is a subset of Timed Computation Tree Logic (TCTL) [15], where atomic expressions are location names, variables, and clocks from the modelled system.

The properties are defined using local properties that are either true or false depending on a specific local configuration. The term *local* means that the property is checked in a specific automata state or configuration[2], according to the definition shown in Table 1.

Definition 4: (Local Property) Given an UPPAAL model $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$. A formula φ is a *local property* iff it is formed according to the syntactical rules shown in Table 1. □

In Def. 4 we have expressed the syntax of the temporal logic that UPPAAL uses. Now, let us see the definition of the five different property classes that UPPAAL may check.

Definition 5: (Temporal Properties) Let $M = \langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ be an UPPAAL model, $\{(\vec{n}, u, v)\}^K = (\vec{n}, u, v)_0, (\vec{n}, u, v)_1, ..., (\vec{n}, u, v)_K$ be a sequence of configurations of length $K \in \mathbb{N} \cup \{\infty\}$, and let φ and ψ be local properties. Then the *trace semantics* of M, written $\tau(M)$, is the set of timed traces, as defined in [16]. We define *TCTL temporal properties* to be those TCTL formulae where the main operator is one of the following: $A[]$, $A <>$ and $-- >$. See Table 2 to see the semantics of such formulae, where $\vDash_{loc}$ is a satisfaction relation stating that a configuration $\{(\vec{n}, u, v)_i\}$ satisfies the local property φ, that is, the proposition stated in φ is fulfilled in $\{(\vec{n}, u, v)_i\}$, with $i \in 0 \ldots K$. □

The temporal property operators dual to $A[\,]$ and $A <>$ are defined as follows:

$$M \vDash E <> \varphi \quad \text{iff} \quad \neg(M \vDash A[\,]\ \text{not}(\varphi))$$
$$M \vDash E[\,]\ \varphi \quad \text{iff} \quad \neg(M \vDash A <> \text{not}(\varphi)).$$

For example, let us consider a property that specifies the situation in which a client has to increase his credit card balance if it is zero. In UPPAAL we use the *lead to* operator ($-- >$) since it allows to express that if a a certain state is reached then another given state will be reached later. The above property could be expressed by the following TCTL

2. The state of a NTA is defined in Def. 3.

TABLE 1
Local TCTL Properties

$\varphi ::=$ deadlock		
$\mid$	$A.n$	for $A \in \vec{A}$ and $n \in N_A$
$\mid$	$x \bowtie c$	for $x \in Clocks, \bowtie \in \{<, <=, ==, >=>\}, c \in \mathbb{Z}$
$\mid$	$x - y \bowtie c$	for $x, y \in Clocks, \bowtie \in \{<, <=, ==, >=>\}, c \in \mathbb{Z}$
$\mid$	$v \bowtie w$	for $v, w \in Vars \bigcup \mathbb{Z}, \bowtie \in \{<, <=, !=, ==, >=>\}$
$\mid$	(φ_1)	for φ_1 a local property
$\mid$	not φ_1	for φ_1 a local property
$\mid$	φ_1 or φ_2	for φ_1, φ_2 logical properties (logical OR)
$\mid$	φ_1 and φ_2	for φ_1, φ_2 logical properties (logical AND)
$\mid$	φ_1 imply φ_2	for φ_1, φ_2 logical properties (logical implication)

TABLE 2
Temporal TCTL Properties

$M \vDash A[\,]\ \varphi$	iff	$\forall\{(\vec{n}, u, v)\}^K \in \tau(M).\ \forall k \leq K.\ (\vec{n}, u, v)_k \vDash_{loc} \varphi$
$M \vDash A <> \varphi$	iff	$\forall\{(\vec{n}, u, v)\}^K \in \tau(M).\ \exists k \leq K.\ (\vec{n}, u, v)_k \vDash_{loc} \varphi$
$M \vDash \varphi --> \psi$	iff	$\forall\{(\vec{n}, u, v)\}^K \in \tau(M).\ \forall k \leq K$ $(\vec{n}, u, v)_k \vDash_{loc} \varphi \Rightarrow \exists k' \geq k.\ (\vec{n}, u, v)_{k'} \vDash_{loc} \psi$

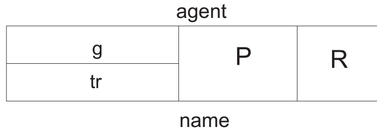

Fig. 1. Box structure

formula:

$$CreditCard.balance == 0 \ --> \ Client.Increase(CreditCard.balance).$$

The property will be satisfied when after the credit card balance is zero, the client increases the credit card balance.

3 *C-O Diagrams*: SYNTAX

In this section we first present an intuitive description of *C-O Diagrams*, we proceed with the diagrams' formal syntax, and we finish with a discussion on the different kind of time constraints we can represent in *C-O Diagrams*.

3.1 General Description

In Fig. 1 we show the basic element of *C-O Diagrams*. It is called a **box** and represents a contract clause. It is divided into four fields. On the left-hand side of the box we specify the conditions and constraints. The *guard* **g** specifies the conditions under which the contract clause must be taken into account (boolean expression). The *time restriction* **tr** specifies the time frame during which the contract clause must be satisfied (deadlines, timeouts, etc.). The *propositional content* **P**, in the center, is the main field of the box used to specify normative aspects (obligations, permissions and prohibitions) that are applied over actions, and/or the specification of the actions themselves. The last field of these boxes, on the right-hand side, is the *reparation* **R**. This reparation, if specified by the contract clause, is a reference to another contract that must be satisfied in case the main norm is not satisfied (a *prohibition* is violated or an *obligation* is not fulfilled, there is no reparation for *permissions*). Each box also has a name and an agent. The *name* is useful both to describe the clause and to reference the box from other clauses, so it must be unique. The *agent* indicates who the performer of the action is.

Example 1: Let us consider the description of the operation of a ***coffee machine*** containing, among others, the following clauses:

- "The coffee machine *must* deliver coffee after payment in less than *one minute*", that is an **obligation** including a **deadline**.
- "The client *has the option* to choose coffee with milk", that is an example of **permission**.
- "The client *should not pay* with coins different from Euros or Dollars", that is an example of **prohibition**.
- "The coffee machine *must* deliver milk if *coffee with milk has been chosen*", that is a **obligation** applied only if a **condition** is fulfilled.
- "The coffee machine *must* refund money if *coffee is not delivered*", that is a **reparation** to

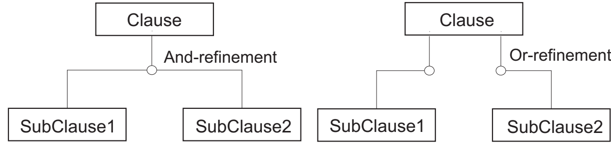

Fig. 2. AND/OR refinements in *C-O Diagrams*

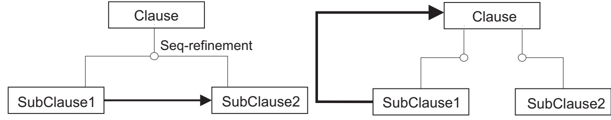

Fig. 3. SEQ refinement and repetition in *C-O Diagrams*

the **obligation** of delivering coffee. □

The basic element of *C-O Diagrams* can be refined by using AND/OR/SEQ refinements. The aim of these refinements is to capture the hierarchical clause structure followed by most contracts. An **AND-refinement** (left-hand side of Fig. 2) means that all the subclauses must be satisfied in order to satisfy the parent clause. An **OR-refinement** (right-hand side of Fig. 2) means that it is only necessary to satisfy one of the subclauses in order to satisfy the parent clause, so as soon as one of its subclauses is fulfilled, we conclude that the parent clause is fulfilled as well. A **SEQ-refinement** (left-hand side of Fig. 3) means that the norm specified in the target box (*SubClause2* in Fig. 3) must be fulfilled after satisfying the norm specified in the source box (*SubClause1* in Fig. 3). By using these structures we can build a hierarchical tree with the clauses defined by a contract, where the leaf clauses correspond to the atomic clauses, that is, to the clauses that cannot be divided into subclauses. There is another structure that can be used to model **repetition**. This structure is represented as an arrow going from a subclause to one of its ancestor clauses (or to itself), meaning the repetitive application of all the subclauses of the target clause after satisfying the source subclause. For instance, in the right-hand side of Fig. 3, we have an **OR-refinement** with an arrow going from *SubClause1* to *Clause*. It means that after satisfying *SubClause1* we apply *Clause* again, but not after satisfying *SubClause2*.

Only the specification of *actions* in the **P** field of the leaf boxes of our diagrams is considered. The composition of actions can be achieved by means of the different kinds of refinement. In this way, an AND-refinement can be used to model *concurrency* "&" between actions, an OR-refinement can be used to model a *choice* "+" between actions, and a SEQ-refinement can be used to model *sequence* ";" of

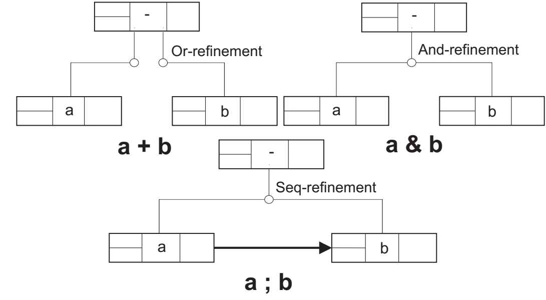

Fig. 4. Compound actions in *C-O Diagrams*

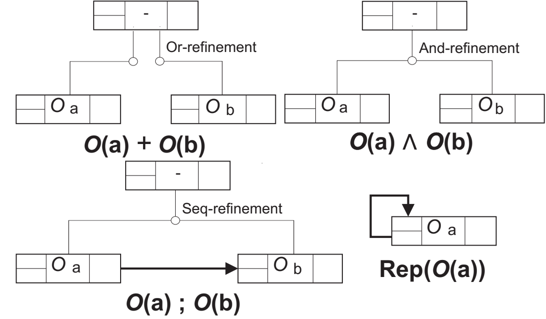

Fig. 5. Composition of norms in *C-O Diagrams*

actions. In Fig. 4 we can see an example of how to model these compound actions through refinements, given two actions *a* and *b*. The composition via an AND-refinement captures the case where two or more clauses must be satisfied in order to consider that the root clause has been satisfied as well. In reference to our example in the figure, the same result is achieved if action *a* is performed first and then action *b* or viceversa[3]. On the other hand , an OR-refinement just needs one of its subclauses to be satisfied to consider the root clause satisfied. The example shows that either the performance of action *a* or action *b* is sufficient to consider the root clause satisfied. The semantic behind this OR-refinement includes both internal and external choices since it is possible to capture a situation where an external agent of a system, e.g. a user, chooses the action to be performed. Therefore, a *C-O Diagram* can capture a contract between several parties from a generic point of view including both types of choices.

The *deontic norms* (obligations, permissions and prohibitions) that are applied over these actions can be specified in any box of our *C-O Diagrams*,

3. We take the point of view of timed automata theory, that is, clocks only evolve via the updates, guards and invariants over transitions and states. This point of view leads us to semantically describe concurrency as an interleaving of the concurrent actions using several consecutive transitions with a compatible set of updates, guards and invariants.

affecting all the actions in the leaf boxes that are descendants of this box. Whenever we use a leaf box to specify the deontic norm, the norm only affects the action we have in this box. The upper-case "*O*" denotes an obligation, an upper case "*P*" denotes a permission, and an upper case "*F*" denotes a prohibition (forbidden). These letters are written in the top left corner of the field **P**.

The composition of deontic norms is achieved by means of the already mentioned refinements. Thus, an AND-refinement corresponds to the *conjunction* operator "$\wedge$" between norms, an OR-refinement corresponds to the *choice* operator "$+$" between norms, and a SEQ-refinement corresponds to the *sequence* operator ";" between norms.

Example 2: Let us consider a leaf box specifying the obligation of performing an action a, written as $O(a)$, and another leaf box specifying the obligation of performing an action b, written as $O(b)$. These two norms can be combined in the three different ways mentioned above through the different kinds of refinement (Fig. 5). Considering the ***coffee machine*** example, we can suppose that action *a* corresponds to "payment (pay)" and action *b* corresponds to "No_different_Coins (diffcoins)", so if we consider the combination of both obligations with an AND-refinement, it is specified that both obligations have to be satisfied in any order. Fig. 6 shows the complete *C-O Diagram* for the coffee machine example. The main clauses are combined by using a SEQ-refinement, that is, 'Coffee_Selection", "Payment_In_Coins" and "Pouring" clauses. Each one is composed of several subclauses by different types of refinements. For instance, the clause "Coffee_Selection" is composed of two subclauses "Coffee_With_Milk" and "Coffee", by an OR-refinement. These clauses represent the permission of selecting coffee with or without milk, respectively, and it is the agent client who decides the desired type of coffee. The clause "Payment_In_Coins" is the combination of two subclauses: "Payment" and "No_Diff_Coins", by an AND-refinement. The first depicts the payment obligation, and the second represents the prohibition of using different types of coins. Finally, the clause "Pouring" is composed of two subclauses: "Pour_Coffee&Milk" and "Pour_Coffee", by an OR-refinement. "Pour_Coffe&Milk" depicts the obligation of delivering the coffee, where the time restriction $t_{Payment} < 60$ indicates that the coffee machine must deliver the coffee in 60 seconds after the payment. This clause also has a guard, Opt==Milk, which indicates that the coffee machine must deliver coffee with milk as this was the selected option. The clause "Pour_Coffee" is similar to the latter, representing that the selected option was coffee without milk. These latter clauses have a reparation "R1", when the obligation of pouring coffee is not satisfied, consisting of returning money to the client. Finally if the process ends successfully, the contract is applied again to serve a next client (repetition refinement of the root clause). □

TABLE 3
C-O Diagrams syntax

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

3.2 Syntax

When defining *C-O Diagrams*, there are some syntactic constraints that must be taken into account. First, exactly one deontic norm must be specified in each of the branches of our hierarchical tree, i.e., we cannot have an action without a deontic norm applied over it and we cannot have deontic norms applied over other deontic norms. Also, *agents* must only be specified in the boxes where a deontic norm is defined, being each agent associated to a concrete deontic norm. Finally, the *repetition* of both, actions and deontic norms, can be achieved by means of the repetition structure we have in *C-O Diagrams*.

Definition 6: (*C-O Diagrams* Syntax). Let us consider a finite set of real-valued variables $\mathcal{C}$ standing for clocks, a finite set of non-negative integer-valued variables $\mathcal{V}$, a finite alphabet Σ for actions, a finite set of identifiers $\mathcal{A}g$ for agents, and another finite set of identifiers $\mathcal{N}$ for names. Thus, an action $a \in \Sigma$ is a set of assignments and resets over variables $\mathcal{V}$ and clocks $\mathcal{C}$, that is, $a = \{v := expr | v \in \{\mathcal{V} \cup \mathcal{C}\}\}$. ϵ represent the empty expression. We use C to denote the contract modelled by a *C-O Diagram*. The syntax of a diagram is defined by the EBNF grammar shown in Table 3, where $a \in \Sigma$, $agent \in \mathcal{A}g$ and $name \in \mathcal{N}$. A guard g is ϵ or a conjunctive formula of atomic constraints of the form: $v \sim n$ or $v - w \sim n$, for $v, w \in \mathcal{V}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$, whereas a time restriction tr is ϵ or a conjunctive formula of atomic

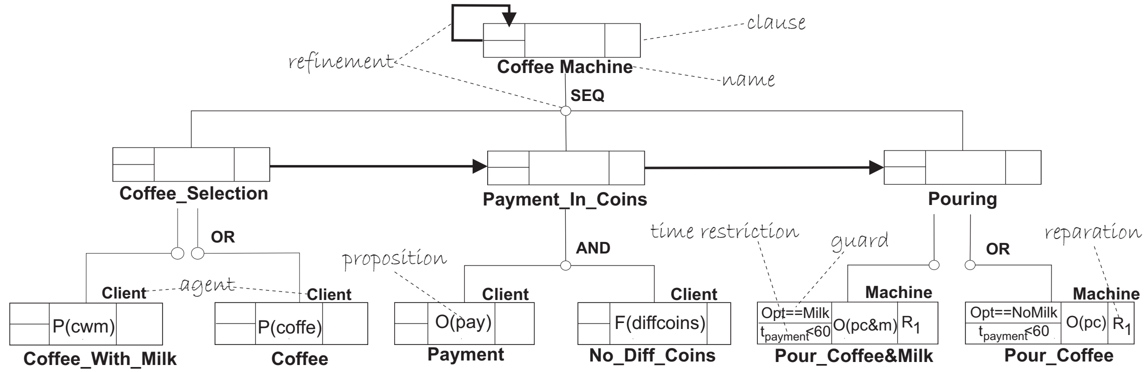

Fig. 6. Coffee Machine C-O Diagram

constraints of the form: $x \sim n$, for $x \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. O, P and F are the deontic operators corresponding to obligation, permission and prohibition, respectively, where $O(C_2)$ states the obligation of performing C_2, $F(C_2)$ states prohibition of performing C_2, and $P(C_2)$ states the permission of performing C_2. And, Or, Seq and Rep are the operators corresponding to the refinements we have in *C-O Diagrams*, AND-refinement, OR-refinement, SEQ-refinement and REP-refinement, respectively. □

The first three lines of the grammar shown in Table 3 define that the simplest contract we can have in *C-O Diagrams* is that consisting of only one box including the elements $agent$ and $name$. Optionally, we can specify a guard g and a time restriction tr. We also have a deontic operator (O, P or F) applied over an action a, and in the case of obligations and prohibitions it is possible to specify another contract C as a reparation.

We use C_1 to define a more complex contract where we combine different deontic norms by means of any of the 4 different refinements.

C_2 represents actions under deontic operators: we can simply write a simple action a in the box, being the deontic operator applied only over it, or we can refine this box in order to apply the deontic operator over a compound action. In this case, the subboxes (the grammar for C_3) cannot define a new deontic operator, as it has already been defined in the parent box (affecting all the subboxes).

Example 3: Let us consider a simple contract specifying that a client has the obligation of paying a certain amount of money. In Fig. 7 three possible situations are depicted. The first, shown in $Example3_1$, depicts the payment obligation with a reparation contract C'_{3_1} (Fig. 7 bottom left), which could be a penalty over the amount of money. The second, as shown by $Example3_2$, where

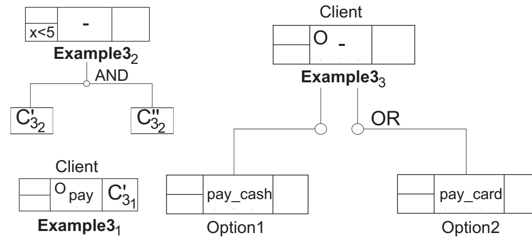

Fig. 7. Syntax examples

$C_{3_2} := (\epsilon, Example3_2, \epsilon, x < 5, C'_{3_2}\ And\ C''_{3_2}, \epsilon)$ is the composed contract specifying that contract C'_{3_2} and contract C''_{3_2} must be satisfied in order to satisfy C_{3_2} within 5 time units (Fig. 7 top left), where C'_{3_2} and contract C''_{3_2} could be the payment obligation and the obligation of receiving a ticket, respectively. Finally, $Example3_3$ models the contract $C_{3_3} := (Client, Example3_3, \epsilon, \epsilon, O(C'_{3_3}\ Or\ C''_{3_3}), \epsilon)$, where we have that $C'_{3_3} := (\epsilon, Option1, \epsilon, \epsilon, pay_cash, \epsilon)$ and $C''_{3_3} := (\epsilon, Option2, \epsilon, \epsilon, pay_card, \epsilon)$, is a contract specifying for a client the obligation of paying by cash or by credit card (Fig. 7, right). □

The update of variables and clocks is performed by actions, that is, *an action* specifies a set of assignments and resets over these data. For instance, in Example 3 the action pay_card may include the modification of the variable *balance*: $balance := balance - spent_amount$.

3.3 Real-Time Restrictions

C-O Diagrams present the capability to treat temporal references. This aspect of a contract is considered in a diagram via the tr field of a clause. This field is specified using a time constraint representing deadlines and timeouts. tr encodes a temporal restriction specified by means of a conjunction of inequalities as presented above. To capture these constraints properly, it is necessary to describe with accuracy any explicit or implicit time references.

For instance, it is possible to refer to a specific time and/or date, "July 10th, 2003, 20:00". However this reference can be more subtle "5 days after the payment". To capture both kinds of time references, or any other a user may want to specify we consider the following three types of restrictions:

- A time restriction specified using **absolute time** must be specified by rewriting the terms in which absolute time references occur. For that purpose we define a global clock T capturing the time passage starting at the moment when the contract is enacted. Time references are then captured as deadlines involving clock T and considering the smallest time unit needed in the contract.
- A time restriction specified using **relative time** must be specified by introducing an additional clock to register the amount of time that has elapsed since another clause has been satisfied. We call this clock t_{name}, where $name$ is the clause used as reference for the specification of the time restriction. Therefore, we define a set of additional clocks $C_{add} = \{t_{name} \mid t_{name} \in \mathcal{R}_0^+\}$ for all $name$ in $\mathcal{N}$.
- A time restriction specified using clocks defined by the **user**, C_{user}.

As a result, the set of clocks of the timed automaton would be $\mathcal{C} = \{T\} \cup C_{add} \cup C_{user}$.

Example 4: For **absolute time** let us consider a clause that must be satisfied between the *5th of November* and the *10th of November*, and that the contract containing this clause is enacted the *31st of October*. If we suppose that *days* is the smallest time unit used in the contract for the specification of real-time restrictions, the time restriction of this clause is written as $(T \geq 5)\, and\, (T \leq 10)$. In the case of **relative time** let us consider a contract with a clause $name_1$ that must be satisfied between *5* and *10* days after another clause $name_2$ has been satisfied. In this case we define an additional clock t_{name_2} and the time restriction of the clause, $name_1$, is written as $(t_{name_2} \geq 5)\, and\, (t_{name_2} \leq 10)$. □

4 *C-O Diagrams*: Semantics

C-O Diagrams specify concepts, that due to their high level of abstraction, are difficult to capture with a timed automata. For instance, it is difficult to capture the obligation to perform an action and its reparation, since timed automata cannot make the difference between branching as an option and branching due to violation. Therefore, the *C-O Diagrams* semantics is defined by means of a translation into an extended version of a *Network of Timed Automata* (NTA). In this section we first present an extension of timed automata suitable as a semantic domain for *C-O Diagrams*, we then discuss some issues concerning timing constraints, and we finish by showing the transformation rules to translate our diagrams into such timed automata.

4.1 NTA extended with normative clauses

Informally, *C-O Diagrams* are a graphic representation of norms applied to actions which are enacted under certain conditions and timing constraints. *C-O Diagrams* could be seen as the high level representation of timed automata where the normative concepts of obligation, permission and prohibition over actions are sets of propositional variables showing that in certain states these concepts are to be satisfied or not. In the following, we give a definition of an extended version of timed automata to capture the semantics of *C-O Diagrams*.

Definition 7: (Network of Timed Automaton extended with Normative Clauses) Let $\mathcal{A}_i = (N_i, n_{0_i}, n_{n_i}, E_i, I_i)$, $i = 1, \ldots, k$ be a family of timed automata where a new node $n_{n_i} \in N_i$ is added to define the final node of an automaton.[4] A *C-O NTA* is a set $(\mathcal{A}_1, ..., \mathcal{A}_k, \mathcal{N})$, where $\mathcal{N}$ is a set of clause names. A *state* of a *C-O NTA* s is a tuple $(\bar{n}, u, v, Cla)$, where $(\bar{n}, u, v)$ is as defined in Def. 3, and $Cla = (Vio, Sat, Per)$ where Vio, Sat and Per are sets of propositional variables ranged over $\mathcal{N}$. The set of all states is $\mathcal{S}$ and the *initial state* is $s_0 = (\bar{n_0}, u, v, Cla_0)$, where Cla_0 is empty. □

We will not expand here on the formal semantics of such automata, but just mention the intuition of the new added sets. Intuitively, when using *C-O NTA* as a semantic domain for *C-O Diagrams*, the clauses names defined in $\mathcal{N}$ will correspond to those names defined in the diagrams. The sets Vio, Sat and Per encodes the clauses that have been violated, satisfied or permitted at a certain state s of a *C-O NTA*. Therefore these sets are empty at the initial state s_0.

The set Vio consists of the names of the obligation and prohibition clauses violated at this state. On the other hand, the set Sat consists of the obligation and prohibition clauses satisfied at the state. That is, if we have a contract where the root clause is refined via an AND-refinement of two clauses $(agent_j, name_j, tr_j, g_j, O(C_j), R_j)$ and $(agent_k, name_k, tr_k, g_k, F(C_k), R_k)$ representing an obligation and a prohibition respectively, the sets

4. An extra parameter is added to the classical automaton definition to establish which is the final node. This parameter is used in the transformation rules for compositional purposes.

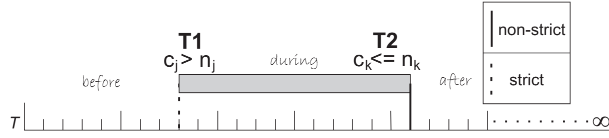

Fig. 8. Time interval with the lower and upper bounds, $T1$ and $T2$.

Vio and Sat may consist of any combination of $name_j$ and $name_k$ including the empty set.

The set Per encodes the permissions that have been made effective, that is, the clause names representing permission that in the sequence of the automata have been already executed. In case of a contract defined as a permission $(agent, name, tr, g, P(C), \epsilon)$, the set Per may consist of the clause name, $name$, or being empty.

4.2 Time restrictions and guards

Before explaining how the full translation of *C-O Diagrams* into C-O NTA is done, we discuss some issues concerning time constraints and guards.

When a clause $(agent, name, g, tr, C, R)$ is translated into a *C-O NTA*, the time restriction tr field specifies the time constraints under which the clause must be enacted. This time constraint encodes a time interval as depicted in Fig. 8. This time interval is characterized by the lower and upper bounds $T1$ and $T2$, which can be either strict or non-strict.

Definition 8: (Time interval for tr) Let tr be a time restriction given in a clause $(agent, name, g, tr, C, R)$ of a *C-O Diagram* defined as a conjunction of inequalities $c_1 \sim n_1 \wedge c_2 \sim n_2 \wedge \ldots \wedge c_n \sim n_m$ where $c_i \in \mathcal{C}$, $n_i \in \mathbb{Z}$ and $\sim \in \{\leq, <, =, >, \geq\}$ $(\forall i \in \{1, \ldots, m\})$. The *time interval* for tr is the interval given by the inequalities $T1 := (c_j \sim_1 n_j)$ and $T2 := (c_k \sim_2 n_k)$, where $T1$ and $T2$ are the greatest lower bound and the lowest upper bound of the time restriction with $\sim_1 \in \{>, \geq, =\}$ and $\sim_2 \in \{\leq, <, =\}$, respectively. □

It is important to take into account the moment when a clause is activated, i. e., when the clause is made effective in the automata execution sequence. If this happens before time $T1$ then time elapses till reaching $T1$, in which case the time constraint is satisfied (up till $T2$). If the activation occurs during the interval then the time constraint is immediately satisfied. In both cases, once the execution sequence is activated, the maximum time to accomplish the clause is bounded by $T2$, which implies that any subclause given by C is constrained by this upper bound. This fact is captured by the addition of $T2$ to all the invariants of the states encoding these subclauses. Finally, if the activation occurs after $T2$, then the clause cannot be made effective since the interval has been *missed* (that is, the time constraints cannot be satisfied).

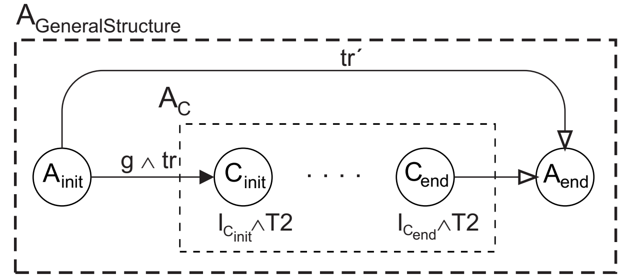

Fig. 9. *C-O NTA* general structure of a clause.

With respect to the guards (g), it is worth noting that the variables on which the guards depend on, may be updated by some other parallel automaton. Therefore, a clause can be active respecting the time constraint tr but the guard could be false. In this case time could elapse till the time interval is missed, or the guard could eventually become true due to further updates.

A generic scheme of the *C-O NTA* corresponding to a clause is depicted in Fig. 9[5], where some of the issues discussed above concerning time restrictions and guards are visualized. The two first cases, activation before or during the interval, are captured with the transition labeled with $g \wedge tr$, where the contract C is enacted with the addition of $T2$ to its invariants. The last situation, when the time interval is missed, is represented by the transition labeled with tr', accounting for the *missed deadline* (it is assumed here that the transition does not satisfy g). We formally define the notion of missed deadline in what follows.

Definition 9: (Missed interval tr') Given a *C-O NTA* $\mathcal{A}$, and tr, c_k, n_k and T_2 as defined in Def. 8, we define a *missed deadline* for transition tr to be the following time constraint:

$$tr' := \begin{cases} (c_k \geq n_k) & \text{iff } T2 \equiv (c_k < n_k) \\ (c_k > n_k) & \text{iff } T2 \equiv (c_k \leq n_k) \end{cases}$$

□

We next introduce the transformation rules adapting this general structure to the different situations given by the different types of clauses.

4.3 Transformation Rules

In the previous subsections we have presented our extension of timed automata suitable to be used

5. In Fig. 9 and all subsequent figures, we use white arrowheads to denote urgent edges whereas dark arrowheads depict non-urgent edges (as defined in Def. 1).

as a semantic target model for *C-O Diagrams*. We have made explicit some considerations about time restrictions and guards. We present now how *C-O Diagrams* are translated into *C-O NTA*. The translation is done by induction using several transformation rules given by the function trf that accepts as a parameter a *C-O Diagram* and gives as a result a *C-O NTA*.

Definition 10: (Transformation function trf) The function trf : *C-O Diagrams* $\rightarrow$ *C-O NTA* transforms a *C-O Diagram* (as given by the EBNF grammar in Def. 6) into a *C-O NTA*. The function is given by the rewriting system introduced in Tables 4–6. □

These tables are divided in two columns. In the left column we can see the transformation function for a given clause, and in the right column the graphical representation of the *C-O NTA* obtained for this clause. Note that we do not represent the sets Vio, Sat and Per in the visual representation given in the tables. This is due for sake of readability, but also because the transformation is done recursively and the content of these sets will depend on the further application of the transformation function to subclauses of the given clause.

The first table shows the *C-O NTAs* for six different expressions including the empty action, simple actions and composition of actions via conjunction, disjunction and sequence refinements. This table is explained in further detail in the next subsection. Table 5 shows the *C-O NTA* obtained from the transformation applied to the deontic clauses, i.e., obligation, prohibition and permission clauses (see subsection 4.3.2). Finally, subsection 4.3.3 deals with Table 6, which shows the resulting automata for the composition of deontic clauses via repetition, conjunction, disjunction and sequence refinements.

4.3.1 Actions

Table 4 shows the transformation rules for an empty action, an action and conjunction, disjunction and sequence of actions given by the EBNF syntax corresponding to the classes C_2 and C_3 of Table 3. The first, rule **(1)**, is the empty action depicted by two states, init and end, which are not connected. This empty action is used to capture cases where no action should be performed. For instance, a reparation from an obligation or prohibition has not been specified, that is, they cannot be repaired and therefore if the automata follows such a path, the system will end in a deadlock. Rule **(2)** is for the case of a simple action not under any deontic modality. The action consist of two states connected by a transition where the action is performed and therefore the clause name is added to the satisfaction set to reflect this fact (s_1). Furthermore, the clock t_{name} is reset (r_1) allowing other clauses to reference this clock in their temporal constraints tr. Rule **(2) bis** is applied to the case when a name for the clause is not provided, for instance, in the clause $(agent, name, g, tr, O(a), R)$ the transformation over the action defined as $trf(a)$ is then translated as $trf((agent, \epsilon, \epsilon, \epsilon, a, \epsilon))$.

Three other rules are defined to describe the transformations given for clauses consisting of conjunction, disjunction and sequence of actions. The conjunction of actions, rule **(3)**, is defined via the Cartesian product[6] of all subclauses defined within the expression, that is, the Cartesian product of all the resulting *C-O NTA*. Whereas, a disjunction of actions, rule **(4)**, is defined as a pairwise disjoint of actions, that is, the possibility to accomplish one among different actions. Therefore, we define this transformation rule by an automata consisting in two states OR_{init} and OR_{end} connected to all the initial and final states of the corresponding automata of its actions, respectively. Regarding the sequence of actions, rule **(5)**, the first step taken is the transformation of each action into its automaton. Afterwards, the first action is appended to the second one in a succession until the last action. Thus, initial node of the first action becomes the initial node for the sequence and the last node of the last action is connected to the last node created for the sequence structure in order to modify the satisfaction set and reset the clause clock.

Example 5: Let us consider a new version of the ***coffee machine*** example, where the coffee machine could deliver tea and cappuccino in addition to coffee. Actions "delivers coffee" (Del_Coffee), "delivers tea" (Del_Tea) and "delivers cappuccino" (Del_Cap) are composed by an OR-refinement (only one of them is performed). The automaton obtained by applying rule **(4)** is shown in Fig. 10. □

4.3.2 Deontic clauses

Until now, we have seen how the automata corresponding to different actions (simple or compound) specified in a *C-O Diagram* are constructed, and we have seen that these translations only modify the content of the satisfaction set. We will see now not only how this set is modified, but also the violation and permission sets. We define the transformation rules specifying how these automata are compound when we apply a deontic norm (obligation, permission or prohibition) over the actions in the *C-O*

6. The Cartesian product of automata is defined as any combination of the transitions belonging to each automata [17].

TABLE 4
Semantic for actions and composition of actions.

Transformation Rules	C-O NTA
$trf(\epsilon) = \{(\{n_{init}, n_{end}\}, n_{init}, n_{end}, \emptyset, \emptyset), \emptyset\}$ **(1)**	$A_{EmptyAction}$: n_{init}, n_{end}
$trf((agent, name, \epsilon, \epsilon, a, \epsilon)) =$ $\{(N_A, n_{init}, n_{end}, E_A, \emptyset), \{name\}\}$, where: $N_A = \{n_{init}, n_{end}\}$ and $E_A = \{n_{init} \xrightarrow[s_1]{agent.a, r_1} n_{end}\}$. **(2)**	$A_{AtomicAction}$: n_{init} —agent.a , r_1 / s_1→ n_{end}
$trf((agent, \epsilon, \epsilon, \epsilon, a, \epsilon)) =$ $\{(N_A, n_{init}, n_{end}, E_A, \emptyset), \{name\}\}$, where: $N_A = \{n_{init}, n_{end}\}$ and $E_A = \{n_{init} \xrightarrow{agent.a} n_{end}\}$. **(2)bis**	$A_{AtomicActionBis}$: n_{init} —agent.a→ n_{end}
$trf((\epsilon, name, \epsilon, \epsilon, C_3^1\, And\, C_3^2\, And\, \dots\, And\, C_3^n, \epsilon)) =$ $\{(N_{And}, CP_{init}, And_{end}, E_{And}, I_{And}), \mathcal{N}_{And}\}$, where: $N_{And} = N_{CP} \cup \{And_{end}\}$, $E_{And} = E_{CP} \cup \{CP_{end} \xrightarrow[s_1]{r_1}_u And_{end}\}$, $I_{And} = I_{CP}$, $(N_{CP}, CP_{init}, CP_{end}, E_{CP}, I_{CP}) =$ $trf(C_3^1) \times trf(C_3^2) \times \dots \times trf(C_3^n)$ and $\mathcal{N}_{And} = \{\{name\} \cup \mathcal{N}_{C_3^1} \cup \mathcal{N}_{C_3^2} \dots \cup \mathcal{N}_{C_3^n}\}$ $\times$ is the Cartesian product of the automata. **(3)**	A_{And}: $A_{C_3^1}$ (C^1_{3init} ..... C^1_{3end}) X $A_{C_3^2}$ (C^2_{3init} ..... C^2_{3end}) X ..... X $A_{C_3^n}$ (C^n_{3init} ..... C^n_{3end}) —r_1, s_1→ And_{end}
$trf((\epsilon, name, \epsilon, \epsilon, C_3^1\, Or\, C_3^2\, Or\, \dots\, Or\, C_3^n, \epsilon)) =$ $\{(N_{OR}, OR_{init}, OR_{end}, E_{OR}, I_{OR}), \mathcal{N}_{\mathcal{OR}}\}$, where: $N_{OR} = N_{C_3^1} \cup N_{C_3^2} \cup \dots \cup N_{C_3^n} \cup \{OR_{init}, OR_{end}\}$, $E_{OR} = E_{C_3^1} \cup E_{C_3^2} \cup \dots \cup E_{C_3^n} \cup \{OR_{init} \longrightarrow_u C^1_{3_{init}}$, $OR_{init} \longrightarrow_u C^2_{3_{init}}, \dots, OR_{init} \longrightarrow_u C^n_{3_{init}}\} \cup$ $\{C^1_{3_{end}} \xrightarrow[s_1]{r_1}_u OR_{end}, C^2_{3_{end}} \xrightarrow[s_1]{r_1}_u OR_{end}, \dots C^n_{3_{end}} \xrightarrow[s_1]{r_1}_u OR_{end}\}$, $I_{OR} = I_{C_3^1} \cup I_{C_3^2} \cup \dots \cup I_{C_3^n}$, $trf(C_3^1) = \{(N_{C_3^1}, n0_{C_3^1}, E_{C_3^1}, I_{C_3^1}), \mathcal{N}_{C_3^1}\}$, $trf(C_3^2) = \{(N_{C_3^2}, n0_{C_3^2}, E_{C_3^2}, I_{C_3^2}), \mathcal{N}_{C_3^2}\}, \dots$ $trf(C_3^n) = \{(N_{C_3^n}, n0_{C_3^n}, E_{C_3^n}, I_{C_3^n}), \mathcal{N}_{C_3^n}\}$ and $\mathcal{N}_{OR} = \{\{name\} \cup \mathcal{N}_{C_3^1} \cup \mathcal{N}_{C_3^2} \dots \cup \mathcal{N}_{C_3^n}\}$. **(4)**	A_{OR}: OR_{init} → $A_{C_3^1}$ (C^1_{3init} ..... C^1_{3end}), $A_{C_3^2}$ (C^2_{3init} ..... C^2_{3end}), ..., $A_{C_3^n}$ (C^n_{3init} ..... C^n_{3end}) —r_1, s_1→ OR_{end}
$trf((\epsilon, name, \epsilon, \epsilon, C_3^1\, Seq\, C_3^2\, Seq \dots Seq\, C_3^n, \epsilon)) =$ $\{(N_{SEQ}, C^1_{2_{init}}, Seq_{end}, E_{SEQ}, I_{SEQ}), \mathcal{N}_{SEQ}\}$, where: $N_{SEQ} = \{Seq_{end}\} \cup N_{C_3^1} \cup N_{C_3^2} \cup \dots \cup N_{C_3^n}$, $E_{SEQ} = E_{C_3^1} \cup E_{C_3^2} \cup \dots \cup E_{C_3^n} \cup \{C^1_{3_{end}} \longrightarrow_u C^2_{3_{init}}$, $C^2_{3_{end}} \longrightarrow_u C^3_{3_{init}}, \dots, C^{n-1}_{3_{end}} \longrightarrow_u C^n_{3_{init}}\} \cup$ $\{C^n_{3_{end}} \xrightarrow[s_1]{r_1}_u Seq_{end}\}$, $I_{SEQ} = I_{C_3^1} \cup I_{C_3^2} \cup \dots \cup I_{C_3^n}$, $trf(C_3^1) = \{(N_{C_3^1}, C^1_{2_{init}}, C^1_{2_{end}}, E_{C_3^1}, I_{C_3^1}), \mathcal{N}_{C_3^1}\}$, $trf(C_3^2) = \{(N_{C_3^2}, C^2_{2_{init}}, C^2_{2_{end}}, E_{C_3^2}, I_{C_3^2}), \mathcal{N}_{C_3^2}\}, \dots$ $trf(C_3^n) = \{(N_{C_3^n}, C^n_{2_{init}}, C^n_{2_{end}}, E_{C_3^n}, I_{C_3^n}), \mathcal{N}_{C_3^n}\}$ and $\mathcal{N}_{SEQ} = \{\{name\} \cup \mathcal{N}_{C_3^1} \cup \mathcal{N}_{C_3^2} \dots \cup \mathcal{N}_{C_3^n}\}$. **(5)**	A_{SEQ}: $A_{C_3^1}$ (Seq_{init} C^1_{3init} ..... C^1_{3end}) → $A_{C_3^2}$ (C^2_{3init} ..... C^2_{3end}) → ... → $A_{C_3^n}$ (C^n_{3init} ..... C^n_{3end}) —r_1, s_1→ Seq_{end}

where $r_1 = \{t_{name}$ if $t_{name} \in \mathcal{C}$ else $\emptyset\}$ and $s_1 = add(Sat, name)$.

TABLE 5
Semantic for deontic clauses.

Transformation Rules	C-O NTA
$trf((agent, name, g, tr, O(C_2), R)) =$ $\{(N_O, O_{init}, O_{end}, E_O, I_O), \mathcal{N}_O\}$, where: $N_0 = N_{C_2} \cup N_R \cup \{O_{init}, O_{end}\}$, $E_O = E_{C_2} \cup E_R \cup \{O_{init} \xrightarrow{\neg g \wedge tr'}_{u} O_{end}\} \cup$ $\{O_{init} \xrightarrow{g \wedge tr} C_{2_{init}}, C_{2_{end}} \xrightarrow[s_1]{r_1}_{u} O_{end}\} \cup$ $\{O_{init} \xrightarrow[s_2]{g \wedge tr'} R_{init}, R_{end} \xrightarrow[s_1]{r_1}_{u} O_{end}\}$, $I_O = I_R \cup \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_2}\}$, $\mathcal{N}_O = \{\{name\} \cup \mathcal{N}_{C_2} \cup \mathcal{N}_R\}$, $trf(C_2) = \{(N_{C_2}, C_{2_{init}}, C_{2_{end}}, E_{C_2}, I_{C_2}), \mathcal{N}_{C_2}\}$ and $trf(R) = \{(N_R, R_{init}, R_{end}, E_R, I_R), \mathcal{N}_R\}$. **(6)**	$A_{Obligation}$, A_{C2}, C_{2init}, C_{2end}, $I_{C_{2init}} \wedge T2$, $I_{C_{2end}} \wedge T2$, $g \wedge tr$, r_1, s_1, O_{init}, $\neg g \wedge tr'$, O_{end}, A_R, $g \wedge tr'$, s_2, R_{init}, R_{end}, r_1, s_1
$trf((agent, name, g, tr, F(C_2), R)) =$ $\{(N_F, F_{init}, F_{end}, E_F, I_F), \mathcal{N}_F\}, where:$ $N_F = N_{C_2} \cup N_R \cup \{F_{init}, F_{end}\}$, $E_F = E_{C_2} \cup E_R \cup \{F_{init} \xrightarrow{\neg g \wedge tr'}_{u} F_{end}\} \cup$ $\{F_{init} \xrightarrow[s_1]{g \wedge tr', r_1} F_{end}, F_{init} \xrightarrow{g \wedge tr} C_{2_{init}}\} \cup$ $\{C_{2_{end}} \xrightarrow[s_2]{}_{u} R_{init}, R_{end} \xrightarrow[s_1]{r_1}_{u} F_{end}\}$, $I_F = I_R \cup \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_2}\}$, $\mathcal{N}_F = \{\{name\} \cup \mathcal{N}_{C_2} \cup \mathcal{N}_R\}$, $trf(C_2) = \{(N_{C_2}, C_{2_{init}}, C_{2_{end}}, E_{C_2}, I_{C_2}), \mathcal{N}_{C_2}\}$ and $trf(R) = \{(N_R, R_{init}, R_{end}, E_R, I_R), \mathcal{N}_R\}$. **(7)**	$A_{Forbidden}$, $\neg g \wedge tr'$, $g \wedge tr', r_1$, s_1, A_{C2}, F_{init}, $g \wedge tr$, C_{2init}, C_{2end}, F_{end}, $I_{C_{init}} \wedge T2$, $I_{C_{2end}} \wedge T2$, A_R, s_2, r_1, s_1, R_{init}, R_{end}
$trf((agent, name, g, tr, P(C_2), \epsilon)) =$ $\{(N_P, P_{init}, P_{end}, E_P, I_P), \mathcal{N}_P\}$, where: $N_P = N_{C_2} \cup N_R \cup \{P_{init}, P_{end}\}$, $E_P = E_{C_2} \cup E_R \cup \{P_{init} \xrightarrow{\neg g \wedge tr'}_{u} P_{end}\} \cup$ $\{P_{init} \xrightarrow{g \wedge tr'} P_{end}, P_{init} \xrightarrow{g \wedge tr} C_{2_{init}}, C_{2_{end}} \xrightarrow[s_3]{r_1}_{u} P_{end}\}$, $I_P = I_R \cup \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_2}\}$, $\mathcal{N}_P = \{\{name\} \cup \mathcal{N}_{C_2} \cup \mathcal{N}_R\}$, $trf(C_2) = \{(N_{C_2}, C_{2_{init}}, C_{2_{end}}, E_{C_2}, I_{C_2}), \mathcal{N}_{C_2}\}$ and $trf(R) = \{(N_R, R_{init}, R_{end}, E_R, I_R), \mathcal{N}_R\}$. **(8)**	$A_{Permission}$, $\neg g \wedge tr'$, A_{C2}, P_{init}, $g \wedge tr$, C_{2init}, C_{2end}, r_1, s_3, P_{end}, $I_{C_{2init}} \wedge T2$, $I_{C_{2end}} \wedge T2$, $g \wedge tr'$

where $r_1 = \{t_{name}$ if $t_{name} \in \mathcal{C}$ else $\emptyset\}$, $s_1 = add(Sat, name)$ and $s_2 = add(Vio, name)$.

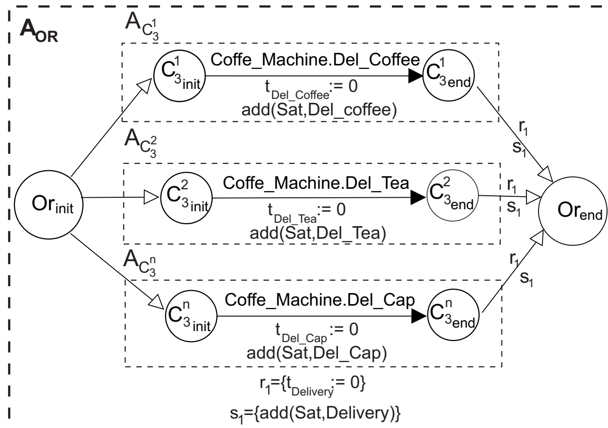

Fig. 10. Example of an OR-refinement of actions

Diagram introduced by the first three lines of Table 3 for the contract class C. We show in Table 5 the *C-O NTA* for the transformation of obligation, prohibition and permission. The obligation of an action or a set of actions, rule **(6)**, refined by the above rules, is defined by an automaton with two nodes, O_{init} and O_{end}, and several transitions connecting these nodes with the nodes of its subclause(s) or the reparation contract. Three different paths are defined for an obligation: the regular behavior, the exceptional behavior when the obligation is not fulfilled and the skipping of the obligation if the guard does not hold. Let us examine this with an example.

Example 6: (Tenant example) Let us assume that

Tenant

sign == true	O(Pay)	ε
T ≥ 1 ∧ T < 7		

Payment

Fig. 11. Clause example of a tenant.

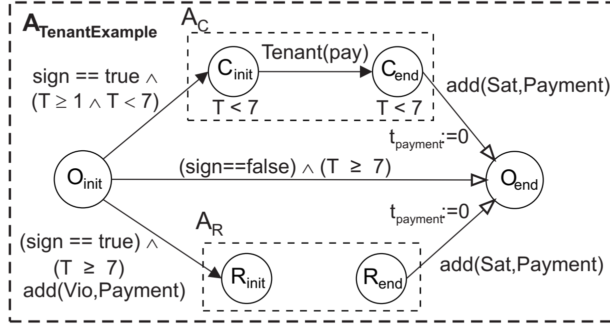

Fig. 12. Automaton of the tenant example.

when renting an apartment there is an obligation to pay the rent between the 1st and the 6th day of each month. We have a boolean condition (the signature of a renting contract has been done), an action (to pay the rent), and a temporal restriction tr with a lower bound (1st day of each Mont) and an upper bound (7th day of the month). This example corresponds to the box in Fig. 11, defined as $(tenant, Payment, (sign == true), ((T \geq 1) \wedge (T < 7)), O(pay), \epsilon)$. □

The translation of the above into *C-O NTA* is depicted in Fig. 12. The automaton shows three branches from the initial state. The first, at the top, is the regular execution, that is, the tenant has signed the contract and between the 1st and the 6th day of the month she pays the rent. The second representing the exceptional case where the tenant does not pay in time, that is, it is the 7th day and she did not pay the rent, then the reparation clause R is enabled. Note that in this case no reparation clause has been defined. The third, in the middle of the figure, corresponds to the situation where the tenant has not signed the contract.

Similar to the obligation case, the translation of a prohibition clause, rule **(7)**, results in an automaton with three branches from the initial state: i) regular behavior in the middle stating that the guard holds and the upper bound $T2$ of the temporal restriction has been reached and that the forbidden behavior has not been executed; ii) the exceptional behavior at the bottom where the guard holds and the forbidden behavior has been performed during the interval specified in tr then the reparation is enabled; iii) the situation where the guard does not hold and the the time to perform the forbidden action has elapsed at the top, i.e., the clock has reached the upper bound $T2$. Finally, rule **(8)** (Table 5) shows the translation of a permission. Here we can see three possibilities, the regular behavior in the middle where the action(s) permitted are performed, the situation where during the interval tr the permitted behavior is not executed at the bottom, and, at the top, where the guard does not hold and the upper time limit has been reached and therefore the permitted action(s) are skipped.

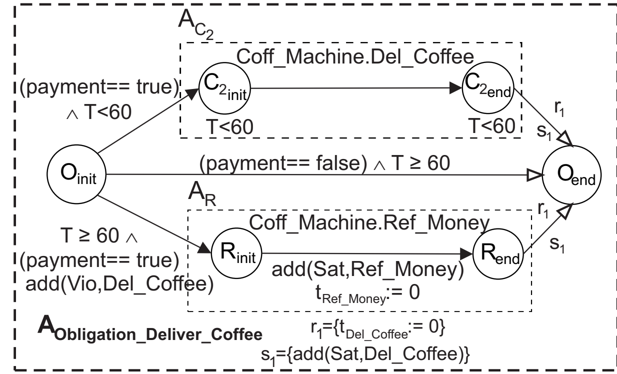

Fig. 13. Example of a Reparation to a violation

The following example shows an extract of an obligation clause of the coffee machine example of Fig. 6 where the reparation R_1 is specified in case the obligation is not performed.

Example 7: Let us consider the clause “The coffee machine *must* deliver coffee after payment in less than *one minute*” (*Del_Coffee*). Furthermore, if together with the obligation of delivering coffee we consider the reparation clause “The coffee machine refunds the money if *coffee is not delivered*” (*Ref_Money*), we get the following:

$$(Coffee_Machine, Del_Coffee, \epsilon, (T < 60), O(Del_Coffee), R_1)$$
$$R_1 = (Coffee_Machinet, Ref_Money, \epsilon, \epsilon, O(Ref_Money), \epsilon).$$

The resulting automaton (by rule **(6)**) is shown in Fig. 13, where we have that the transition connecting nodes O_{init} with R_{init} adds the clause name *Del_Coffee* to the violation set to state that this clause has been violated, but after been repaired by the *Ref_Money* action in the transition connecting R_{end} with O_{end} this clause name is added to the satisfaction set as well. Therefore, we can have two situations: first, the coffee is delivered and then the *Del_Coffee* is added to the satisfaction set and, second, the coffee is not delivered and the name

of the clause appears in the violation set but the money is refunded and the name of the clause then appears in the satisfaction set too. This means that if a clause appears at the same time in the violation and in the satisfaction sets it is because the clause has been repaired allowing us to keep track of the different situations taken place in the automata. □

This example allows us to see how different violation and satisfaction sets are modified in an obliged clause when a reparation is defined. On the other hand, in prohibitions we see the opposite situation, if the actions defined in the prohibition are executed, then the prohibition is breached and therefore the clause name is added to the violation set. But still, if a reparation is defined and successfully performed then after its execution the clause name is added to the satisfaction set. Therefore, if a prohibited action appears in both sets it means that it has been repaired. Notice that, in case of permissions there is no way to breach the contract since a permission cannot be violated and therefore it cannot appear in a violation set.

4.3.3 Composition of Clauses

We define now the rules corresponding to the composition of deontic norms. These automata are obtained by the transformation rules **(9)–(12)** (Table 6) corresponding to the fourth line of the class C and class C_1 of the *C-O Diagrams*syntax shown in Table 3.

The first rule of this table, rule **(9)** shows the transformation for a repetition where a transition connects the ending node again with the initial one. The second transformation, rule **(10)** encodes the conjunction of several deontic norms via a parallel structure where the equivalent automaton of each different subclause is connected with its predecessor and successor automaton via a synchronization action m_i. The third rule **(11)** encode the disjunction of deontic norms. This rule shows a similar structure as the rule **(4)**, but presents a significant difference since some transitions and invariants are labeled with g, tr, tr' and $T2$ to encode the guard and time restriction of the clause. The last rule of this table, rule **(12)**, shows the sequence refinement that is similar to the structure shown in rule **5** with some modifications to take into account guards and invariants.

Example 8: Let us consider a composition using an AND-refinement of the clauses "The coffee machine *must* deliver coffee after payment in less than *one minute*" (*Del_Coffee*), and "The coffee machine *must* deliver milk after payment in less than *one minute*" (*Del_Milk*). The resulting *C-O NTA* (applying rule **(10)**) is shown in Fig. 14, where T is the clock used to control the deadline and *m1* is the urgent channel used to synchronize both automata. The corresponding expression is:

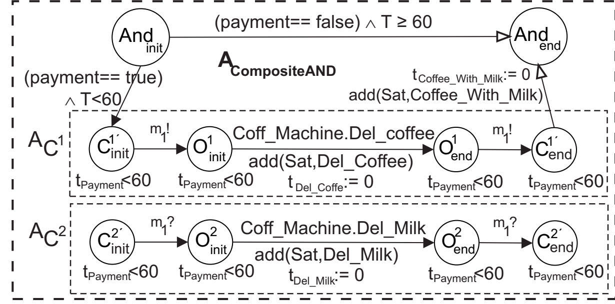

Fig. 14. Example of an AND-refinement of obligations

$$(\epsilon, Coffe_With_Milk, g, tr, C_1 And C_2, \epsilon), with:$$
$$g = \{(payment == true)\}, tr = \{t_{payment} < 60\},$$
$$C_1 = (Coffee_Machine, Del_Coffee, \epsilon, \epsilon, O(Del_Coffee), \epsilon) and$$
$$C_2 = (Coffee_Machinet, Del_Milk, \epsilon, \epsilon, O(Del_Milk), \epsilon).$$

□

5 UPPAAL IMPLEMENTATION

The implementation of *C-O NTAs* in UPPAAL is quite straightforward. There are only a few implementation issues that need further explanation:

1) As there is no way in UPPAAL of directly expressing that an edge without synchronization should be taken without delay, that is, there are no urgent edges, we have to encode this behavior. For this purpose we consider the modelling pattern proposed in [18]. The encoding of urgent edges introduces an extra automaton, that we call $Urgent$, with a single location and a self loop. The self loop synchronizes on an urgent channel that we call urg_edge. An edge can now be made urgent by performing the complementary action.
2) The performance of actions by agents is implemented in two different ways. Actions that interact with the physical environment, like pressing a button, are translated by means of boolean variables in UPPAAL. We define a boolean variable called $agent_action$ for each of the actions considered in the contract. These variables are initialized to **false** and, when an action is performed by an agent on one of the edges, we update the value of the corresponding variable to **true**. The second type concerns actions that modify the value of the variables defined in the diagram. For instance,

TABLE 6
Semantic for compositions of clauses.

Transformation Rules	C-O NTA
$trf((\epsilon, \epsilon, g, tr, Rep(C), \epsilon)) =$ $\{(N_C, C_{init}, C_{end}, E_{Rep}, I_C), \mathcal{N}_C\}$, where: $E_{Rep} = E_C \cup \{C_{end} \xrightarrow{g,tr}_u C_{init}\}$ and $trf(C) = \{(N_C, C_{init}, C_{end}, E_C, I_C), \mathcal{N}_C\}$. **(9)**	
$trf((\epsilon, name, g, tr, C^1 \; And \; C^2 \; And \; \ldots \; And \; C^n, \epsilon)) =$ $\{(N'_{C^1}, C^{1'}_{init}, C^{1'}_{end}, E'_{C^1}, I'_{C^1}),$ $(N'_{C^1}, C^{2'}_{init}, C^{2'}_{end}, E'_{C^2}, I'_{C^2}) \ldots,$ $(N'_{C^n}, C^{n'}_{init}, C^{n'}_{end}, E'_{C^n}, I'_{C^n}), \mathcal{N}_{and}\}$, where: $\forall i \in 1 \leq i \geq n$ $N'_{C^i} = \{C^{i'}_{init}, C^{i''}_{init}, C^{i'}_{end}, C^{i''}_{end}\} \cup N_{C^i},$ $E'_{C^i} = \{C^i_{init} \xrightarrow{\neg g \wedge tr'}_u C^i_{end}\} \cup$ $\{C^{i'}_{init} \xrightarrow{m_{i-1}?} C^{i''}_{init}, C^{i''}_{init} \xrightarrow{m_i!} C_{i_{init}}\} \cup$ $\{C^{i'}_{end} \xrightarrow{m_{i-1}?} C^{i''}_{end}, C^{i''}_{end} \xrightarrow{m_i!} C^i_{end}\},$ $I_{C_i} = \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_i}\},$ $\mathcal{N}_{and} = \{\{name\} \cup \mathcal{N}_{C_1} \cup \mathcal{N}_{C_2} \ldots \cup \mathcal{N}_{C_n}\},$ $trf(C_1) = \{(N_{C_1}, C_{1_{init}}, C_{1_{end}}, E_{C_1}, I_{C_1}), \mathcal{N}_{C_1}\},$ $trf(C_2) = \{(N_{C_2}, C_{2_{init}}, C_{2_{end}}, E_{C_2}, I_{C_2}), \mathcal{N}_{C_2}\} \ldots$ $trf(C_n) = \{(N_{C_n}, C_{n_{init}}, C_{n_{end}}, E_{C_n}, I_{C_n}), \mathcal{N}_{C_n}\}.$ **(10)**	
$trf((\epsilon, name, g, tr, C^1 \; Or \; C^2 \; Or \; \ldots \; Or \; C^n, \epsilon)) =$ $\{(N_{OR}, OR_{init}, OR_{end}, E_{OR}, I_{OR}), \mathcal{N}_{\mathcal{OR}}\}$, where: $N_{OR} = N_{C^1} \cup N_{C^2} \cup \ldots \cup N_{C^n} \cup \{OR_{init}, OR_{end}\},$ $E_{OR} = E_{C^1} \cup E_{C^2} \cup \ldots \cup E_{C^n} \cup$ $\{OR_{init} \xrightarrow{\neg g \wedge tr'}_u OR_{end}, OR_{init} \xrightarrow[g \wedge tr]{} C^1_{init},$ $OR_{init} \xrightarrow[g \wedge tr]{} C^2_{init}, \ldots OR_{init} \xrightarrow[g \wedge tr]{} C^n_{init},$ $C^1_{end} \xrightarrow[s_1]{r_1}_u OR_{end} C^2_{end} \xrightarrow[s_1]{r_1}_u OR_{end},$ $C^n_{end} \xrightarrow[s_1]{r_1}_u OR_{end}\},$ $I_{OR} = I_{C^{1'}} \cup I_{C^{2'}} \cup \ldots \cup I_{C^{n'}},$ $\forall i \in 1 \leq i \geq n \; I_{C^{i'}} = \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_i}\},$ $trf(C^1) = \{(N_{C^1}, n_{0_{C^1}}, E_{C^1}, I_{C^1}), \mathcal{N}_{C^1}\},$ $trf(C^2) = \{(N_{C^2}, n_{0_{C^2}}, E_{C^2}, I_{C^2}), \mathcal{N}_{C^2}\}, \ldots$ $trf(C^n) = \{(N_{C^n}, n_{0_{C^n}}, E_{C^n}, I_{C^n}), \mathcal{N}_{C^n}\},$ $\mathcal{N}_{OR} = \{\{name\} \cup \mathcal{N}_{C^1} \cup \mathcal{N}_{C^2} \ldots \cup \mathcal{N}_{C^n}\}.$ **(11)**	
$trf((\epsilon, name, g, tr, C^1 \; Seq \; C^2 \; Seq \ldots Seq \; C^n, \epsilon)) =$ $\{(N_{SEQ}, Seq_{init}, Seq_{end}, E_{SEQ}, I_{SEQ}), \mathcal{N}_{SEQ}\}$, where: $N_{SEQ} = \{Seq_{init}, Seq_{end}\} \cup N_{C^1} \cup N_{C^2} \cup \ldots \cup N_{C^n},$ $E_{SEQ} = E_{C^1} \cup E_{C^2} \cup \ldots \cup E_{C^n} \cup$ $\{Seq_{init} \xrightarrow{\neg g \wedge tr'}_u Seq_{end}, Seq_{init} \xrightarrow{g \wedge tr} C^1_{init}\} \cup$ $\{C^1_{end} \longrightarrow_u C^2_{init}, C^2_{end} \longrightarrow_u C^3_{init}, \ldots,$ $C^{n-1}_{end} \longrightarrow_u C^n_{init}, C^n_{end} \xrightarrow[s_1]{r_1}_u Seq_{end}\},$ $I_{SEQ} = I_{C^{1'}} \cup I_{C^{2'}} \cup \ldots \cup I_{C^{n'}},$ $\forall i \in 1 \leq i \geq n \; I_{C^{i'}} = \{I(n) \equiv I(n) \wedge T2, \forall n \in N_{C_i}\},$ $trf(C^1) = \{(N_{C^1}, n_{0_{C^1}}, E_{C^1}, I_{C^1}), \mathcal{N}_{C^1}\},$ $trf(C^2) = \{(N_{C^2}, n_{0_{C^2}}, E_{C^2}, I_{C^2}), \mathcal{N}_{C^2}\}, \ldots$ $trf(C^n) = \{(N_{C^n}, n_{0_{C^n}}, E_{C^n}, I_{C^n}), \mathcal{N}_{C^n}\},$ $\mathcal{N}_{OR} = \{\{name\} \cup \mathcal{N}_{C^1} \cup \mathcal{N}_{C^2} \ldots \cup \mathcal{N}_{C^n}\}.$ **(12)**	

where $r_1 = \{t_{name}$ if $t_{name} \in \mathcal{C}$ else $\emptyset\}$ and $s_1 = add(Sat, name)$.

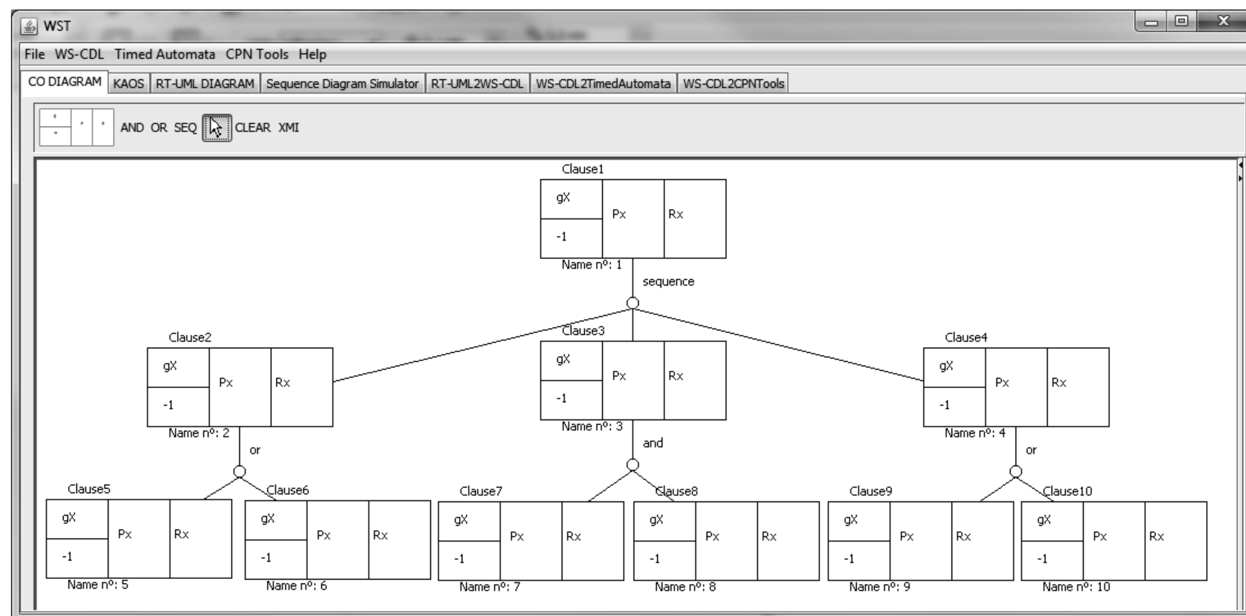

Fig. 15. WST plugin to model *C-O Diagrams*.

the deposit of a certain amount of money into a bank account results in an increase of the balance. In these cases, we define the boolean variable to state that the action has been performed and in the same transitions the referred variables are modified.

3) The violation, satisfaction and permission sets are implemented in UPPAAL by means of boolean arrays and constant integers with the names of the clauses of the contract containing obligations, prohibitions or permissions. We define an array V for violation, an array S for satisfaction, and an array P for permission, all initialized to **false**. The size of the arrays V and S are equal to the number of obligations and prohibitions in the contract, whereas the size of the array P is equal to the number of permissions. We also define constant integers with the name of the clauses containing obligations and prohibitions, initializing each one of them to a different value (from 0 to the size of the arrays V and S minus 1), and constant integers with the name of the clauses containing permissions, initializing each one of them to a different value (from 0 to the size of the array P minus 1). These constants are used as indexes in the arrays. When taking a transition where the target node contains at least one modified set (an obligation/prohibition is violated, an obligation/prohibition is satisfied or a permission is made effective), we update to **true** in the proper array the value of the index corresponding to the clause. In the case of reparations the index corresponding to the proper clause in V is set to **false**.

Example 9:

Let us consider a contract with only one obligation (*Clause_1*), one prohibition (*Clause_2*) and one permission (*Clause_3*). For the implementation of the corresponding NTA in UPPAAL we define a boolean array V and a boolean array S of size two (one obligation plus one prohibition), and a boolean array P of size one (only one permission). The constant integers $Clause_1 = 0$, $Clause_2 = 1$ and $Clause_3 = 0$ are also defined as indexes for the arrays. In this way, we can properly update these arrays. For instance, for a transition where the obligation (*Clause_1*) is satisfied, we update the array S with $S[Clause_1] = true$, and for a transition where the permission (*Clause_3*) is made effective, we update the array P: $P[Clause_3] = true$. □

A prototype of a plugin for WST [19] implementing the above transformation can be seen in Fig. 15.[7]

6 Case Study: Online Auctioning Process (OAP)

The case study presented in this section is inspired by the motivating example described in [20]. It consists of an *Online Auctioning Process* involving

7. Available at http://dsi.uclm.es/retics/wst. This plugin will be available in the next release of the tool, aiming to help users to model the system and perform the automatic translation.

TABLE 7
Norms of the *Online Auctioning Process* contract

N#	Agent	Modality	Action	Condition	Temporal Constraint
1	*Seller*	*Permission*	Can select an item to auction (a_1)	$\emptyset$	$\emptyset$
2	*Seller*	*Prohibition*	Auctions a fraudulent item (a_2)	$\emptyset$	One day after selection (t_2)
3	*Seller*	*Obligation*	Uploads valid information (a_3)	$\emptyset$	One day after selection (t_3)
4	*Auction S.*	*Obligation*	Publishes the auction (a_4)	$\emptyset$	One day after checked (t_4)
5	*Buyer*	*Permission*	Can place bids for the item (a_5)	$\emptyset$	Seven days after publication (t_5)
6	*Buyer*	*Obligation*	Pays the item by credit card (a_6)	Highest bid (g_1)	Three days after auction (t_6)
7	*Buyer*	*Obligation*	Pays the item by PayPal (a_7)	Highest bid (g_1)	Three days after auction (t_7)
8	*Seller*	*Obligation*	Sends the item to the *buyer* (a_8)	Highest bid (g_1)	Fourteen days after auction (t_8)
9	*Auction S.*	*Obligation*	Refunds the payment (a_9)	$\emptyset$	Seven days after violation of *C.8* (t_9)
10	*Auction S.*	*Permission*	Can penalize the *seller* (a_{10})	$\emptyset$	Seven days after violation of *C.8* (t_{10})

the interaction between three different agents: the *buyer*, the *seller*, and the *auction service*.

The online auctioning starts when a *seller* wants to auction an item. The *seller* has **one day** to upload valid information about the item he wants to sell taking into account that the sale of inadequate items such as counterfeit items or wild animals is forbidden. Once an item has been checked and found eligible for auction, the *auction service* also has **one day** to publish the auction of the item. After that, the *buyer* can place bids during **seven days**. When this period of time is over, if the bid placed by the *buyer* is the highest one, the activities concerning the payment and the shipment of the item start.

First, the *buyer* has **three days** to perform the payment, which can be done by means of credit card or PayPal. After the payment has been performed, the *seller* has **fourteen days** to send the item to the *buyer*. If the item is not received within this period of time, the *auction service* has **seven days** to refund the payment to the *buyer* and can penalize the *seller* in some way (for example, not allowing the *seller* to auction new items for a period of time). However, if the on time reception of the item by the *buyer* is acknowledged, the auction process is considered to have finished successfully.

In Table 7 we show a list of the obligations, permissions and prohibitions that can be inferred from the description of the process, where the obligation specified by clause **9** and the permission specified by clause **10** are a possible reparation for the violation of clause **8**. There are nine clauses specifying real-time constraints, clauses **2** till **10**, some sharing the same constraint.

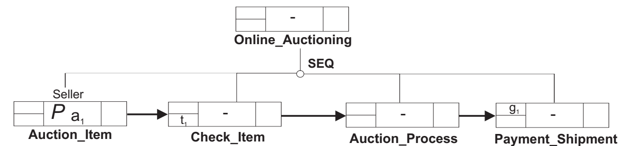

Fig. 16. Top-level of the *Online Auctioning Process*

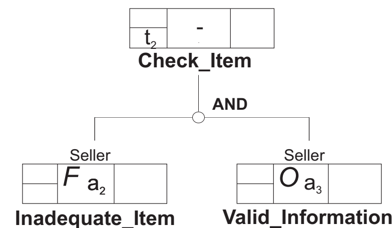

Fig. 17. Decomposition of clause *Check_Item*

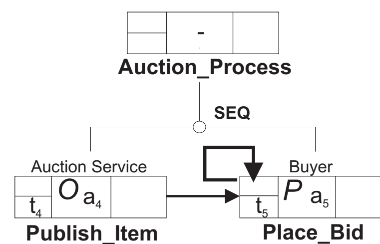

Fig. 18. Decomposition of clause *Auction_Process*

A problem with this textual specification is that the relationship between the different clauses is not clear, making any kind of analysis difficult. We use *C-O Diagrams* to clearly specify the relationship between the different clauses, but not so formal that an expert is needed.

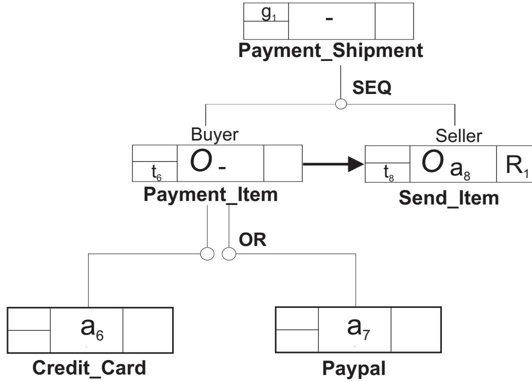

Fig. 19. Decomposition of clause *Payment_Shipment*

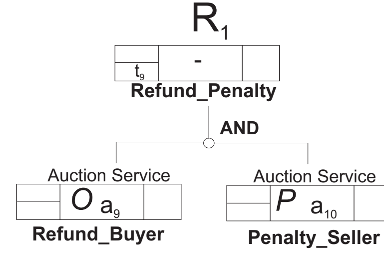

Fig. 20. Reparation of clause *Send_Item*

6.1 OAP *C-O Diagrams*

In what follows we model OAP using *C-O Diagrams* based on the description provided in Table 7. We use a_n to denote the action corresponding to clause number n in the table; the reparation for Clause **8** is called R_1. We also use t_i to denote the real-time constraints of Clause **i**, furthermore, the condition of placing the highest bid is denoted as g_1.

In Fig. 16 we show the top-level of the *C-O Diagram* we specify for the process, called *Online_Auctioning*, starting the sequence from the permission specified in clause **1** (*Auction_Item*). We have grouped the rest of the clauses shown in Table 7 into four more general clauses with a sequence relationship between them: 1) *Check_Item* decomposed via a conjunction of clauses **2** and **3**, 2) *Auction_Process* refined as a sequence of Clause (**4** and **5**), 3) *Payment_Shipment* other sequence of clauses where the first clause *Payment_Item* is refined at the same time with a disjunction of clauses **6** and **7** followed by clause **8** and 4) reparation **R_1** of clause **8** consisting of a conjunction of clauses **9** and **10**. These general clauses cover the different phases that we have been identified in the *Online Auctioning Process* bridging the gap between the informal description and the table. Figures 17–19 expand on the clauses shown in the top-level diagram depicted in Fig. 16. Finally, in Fig. 20 we show the diagram corresponding to reparation R_1 (*Refund_Penalty*). It includes the real-time constraint t_9, and it is decomposed into two subclauses by means of an AND-refinement.

6.2 OAP Timed Automata

For sake of clarity we do not show the textual representation of the translation of *C-O Diagrams* into *C-O NTAs*, and give their graphical representation instead. The automata are shown in Fig. 21, obtained by applying the transformation rules given in subsection 4.3. Every part of these automata is surrounded by a dashed box and, at the top of each box, we can see the name of each automaton preceded by the name of the equivalent clause, and in some cases, by the the type of deontic or refinement operator. These automata consist of three different automata running in parallel, the main automaton and two small automata surrounded by highlighted black dashed squares. Double circles are used to denote the initial node of each automaton. The top level of Fig. 16 is represented by a sequence in $A_{CompositeSEQ_Online_Auctioning}$. This is then refined into a sequence of subclauses C_{AI}, C_{CI}, C_{AP} and C_{PS}, which are translated to the automata $A_{Permission_Auction_Item}$, $A_{CompositeAND_Check_Item}$, $A_{CompositeSEQ_Auction_Process}$, and $A_{CompositeSEQ_Payment_Shipment}$, respectively. The automaton $A_{Permission_Auction_Item}$ consists of a permission of auctioning an item. The last transitions resets the clock $t_{Auction_Item}$ and adds the name of the clause to the permission set stating that this permission has been effective by, in this case, the seller.

The automata $A_{CompositeAND_Check_Item}$ is refined by a composition of the automata obtained from C_{II} and C_{VI}, that is, $A_{Forbidden_Inadequate_Item}$ and $A_{Obligation_Valid_Information}$. The first automaton shows that, if in the period of one day after the item has been selected an action a_2 is performed, then the prohibition is considered breached and the name of the clause $Inadequate_Item$ is added to the violation set. Note that, since no reparation is defined, once the contract is violated, there is no way to recover the system. On the other hand, if the action is not taken for this period of time the prohibition is considered accomplished and is added to the satisfaction set. The second automaton runs in parallel with the main automaton and starts when the transition from the former automata fires the transition labeled with $m_1!$ executed at the same

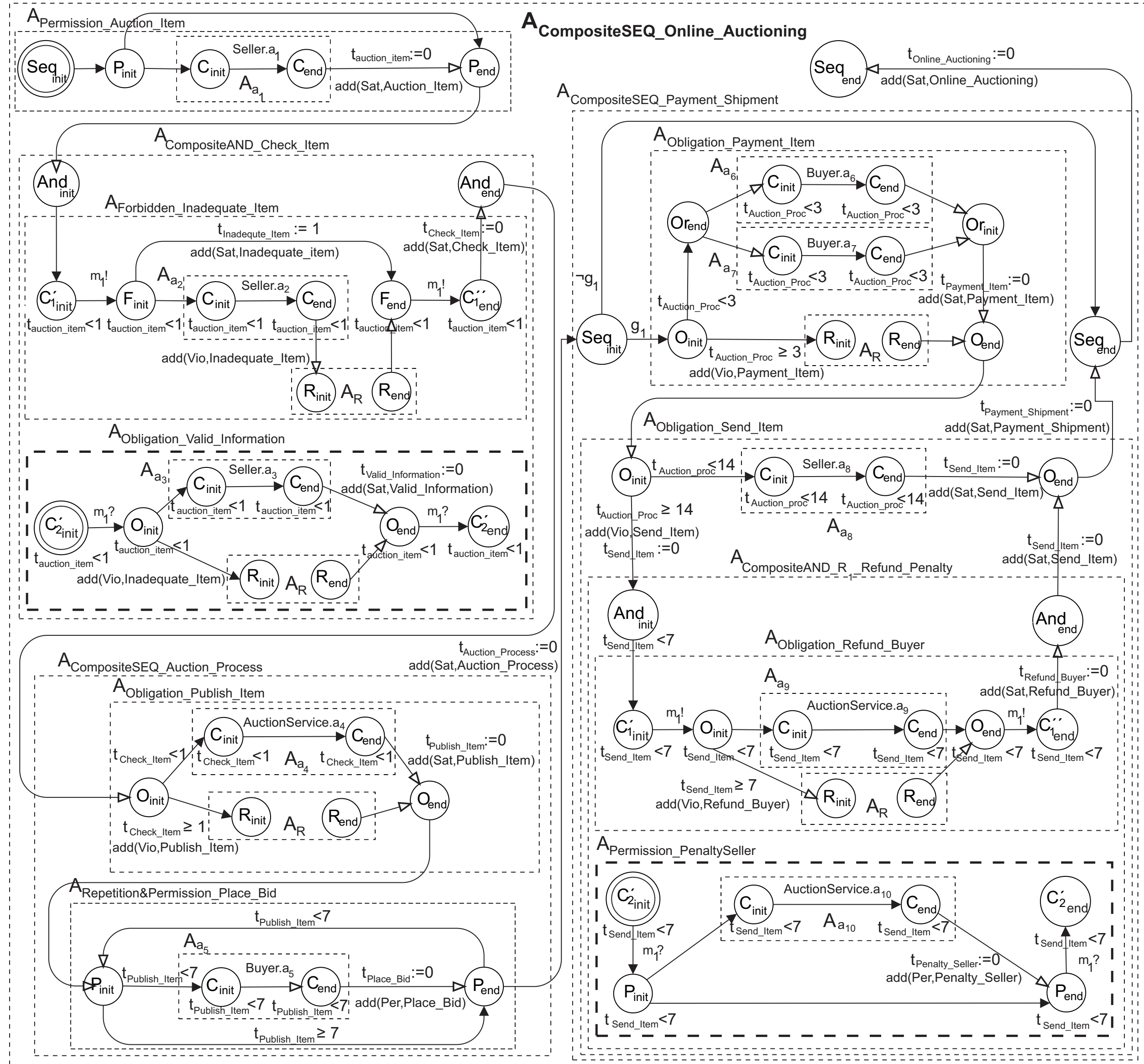

Fig. 21. Automata of the Online Auctioning Process.

time with the transition labeled with $m_1?$ of this other automaton. Once this automaton starts, it captures the obligation of providing valid information, action a_3 performed by the seller, with the same temporal window stated in the former prohibition.

Note that in the above we do not explicitly write transitions with the guard $\neg g \wedge tr'$ as the guard is always false (idem for the other automata).

For sake of space, we do not explain the rest of the translation. We finish by mentioning that the reparation of C_{SI}, i.e., R_1, is translated as $A_{Composite_AND_R_1}$. Note that before R_1 is enacted the name of the clause to be repaired, $Ship_Item$, is added to the violation set, and to the satisfaction set after being repaired.

6.3 OAP Validation

Using the automata defined above as a source, we implement the translation to the specific format supported by UPPAAL. The validation of the contract can be done by means of simulation, where we check that it behaves as expected. For this purpose we use the simulator included in the UPPAAL tool to make the following validations:

- We run the system manually, selecting the transitions to be executed at each step.
- We let the system run automatically; the transitions are therefore executed randomly.

- Whenever the model checker gives a counter-example, we analyze it by running the simulator with the given trace.

6.4 OAP Verification

We use the verifier to check the following kind of properties. i) Obligations and prohibitions are respected, or the corresponding penalties are applied otherwise. We do this by writing *safety* properties expressed in UPPAAL via the temporal operators $A[]\varphi$ (in all paths and in every state the property φ holds), and $A <> \varphi$ (in all paths there is a state where the property φ holds). ii) The contract allows all permissions to be exercised. We do this by checking *reachability* properties, like $E <> \varphi$ (there exists at least one state where the property φ holds), and $E[]\ \varphi$ (the property φ always holds for at least one path). iii) The order between the different clauses follows the order established in the contract, for instance the property ψ must always follow (not necessarily right after) property φ, for which we write $\varphi -- > \psi$ (*unbounded response*).

Note that the counter-examples given by UPPAAL are given in terms of the underlying automata making it difficult for a non-specialized user to understand them. Though this is a shortcoming, we believe that the user may still be able to understand the trace by checking the sequence given against the specification given in the *C-O Diagram*. The query language used for the specification of properties as explained above is a simplified version of TCTL [18] (cf. Section 2.2).

We describe in what follow some of the results concerning the verification of the properties we have checked.

- It is possible to reach a state where the item has been sent to the *buyer*, i.e. that the obligation $Send_Item$ has been satisfied. In UPPAAL:

$$E <> \ (S[Send_Item] == true)$$

  This property is **satisfied**.
- The item is sent to the *buyer* ($Send_Item$) only if the payment has been performed before ($Payment_Item$). In UPPAAL:

$$A[]\ S[Send_Item] == true \\ imply\ S[Payment_Item] == true$$

  This property is **satisfied**.
- The item has been checked ($Check_Item$), if the *auction service* takes more than one day to publish the auction of the item ($t_{Check_Item} > 1$), the clause $Publish_Item$ is violated. In UPPAAL:

$$(S[Check_Item] == true\ and\ t_{Check_Item} > 1) \\ -- > V[Publish_Item] == true$$

  This property is **satisfied**.
- There exists a maximal path in which none of the main obligations and prohibitions of the contract are violated. In UPPAAL:

$$E[]\ (V[Inadequate_Item] == false \\ and\ \ V[Valid_Information] == false \\ and\ \ V[Publish_Item] == false \\ and\ \ V[Payment_Item] == false \\ and\ \ V[Send_Item] == false)$$

  This property is also **satisfied**.

We have also used UPPAAL to check that some undesirable behaviors never happen (that is, we expect that the properties are not satisfied). We have checked the following properties:

- The obligation of sending the item to the *buyer* ($Send_Item$) should be satisfied if the payment has not been performed ($Payment_Item$). In UPPAAL:

$$A[]\ S[Send_Item] == true\ imply \\ S[Payment_Item] == false$$

  This property is **not satisfied**, as expected.
- The obligation of sending the item to the *buyer* ($Send_Item$) and the obligation of refunding the payment to the *buyer* ($Refund_Buyer$) should not be satisfied at the same time, as the refund has to be done only if the item is not sent. In UPPAAL:

$$E <> \ (S[Send_Item] == true \\ and\ \ S[Refund_Buyer] == true)$$

  This property is **not satisfied**, as expected.
- It is an undesired behavior that both the obligation of paying the item ($Payment_Item$) and the obligation of refunding the payment ($Refund_Buyer$) are always not satisfied at the same time. In UPPAAL:

$$A[]\ (S[Payment_Item] == false \\ or\ \ S[Refund_Buyer] == false)$$

  This property is **not satisfied** as expected.

Another interesting property that at first glance might appear to be satisfiable is that when the obligation of paying the item is satisfied ($Payment_Item$), either the obligation of sending

the item ($Send_Item$) or the obligation of refunding the payment ($Refund_Buyer$) must be eventually satisfied. In UPPAAL:

$$S[Payment_Item] == true --> (S[Send_Item] == true \;\; or \;\; S[Refund_Buyer] == true)$$

We run UPPAAL and unexpectedly got that the property is **not satisfied**. To understand the reason we run the trace provided by the verifier leading to the state where the property does not hold. In this trace we can see that automaton $A_{Obligation_Refund_Buyer}$ ends in node R_{init}. In this state both, the obligation of sending the item and the obligation of refunding the payment, have been violated. The result requires further analysis in order to understand whether the problem is in the original specification, in the *C-O Diagrams*, in our translation to *C-O NTA* or in the UPPAAL implementation. After a careful analysis, we propose two possible explanations describing where the problem lies. Given the current specification, and assuming that any non-explicit penalty associated to a violation implies a breach in the contract (and thus to be handled by other legal means), we can assume that the result of our verifier is correct, and no further changes in the original specification are needed. However, if we consider that the property should be satisfied, then the solution should be to modify the specification adding an explicit penalty to the obligation of refunding the payment (or by default a generic clause concerning the violation of any obligation or prohibition with no explicit penalty associated). In any case, we have found a potential breach in the original specification.

7 Case Study: Adaptive Cruise Control (ACC)

This case study deals with the requirements established for an engineering process regarding the correct operation of an *Adaptive Cruise Control* system. The system is concerned with the automatic control of the speed of a car, in some cases taking into account the distance to the vehicle ahead. Sensors are used to measure this distance.

The description of the system is as follows. The system starts working when it is inactive and the *driver* presses a button to activate the system. It can be activated with the current speed or with a previously stored speed. After being activated the *system* should not be deactivated before **10 milliseconds**, so it can perform all the safety checks before allowing deactivation. On the other hand, the *system* has

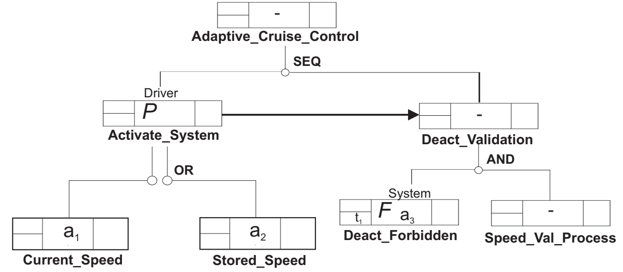

Fig. 22. Top-level of the *Adaptive Cruise Control*

to validate, within **5 milliseconds**, that the speed is in the permissible range. If the validation fails the *system* has **10 milliseconds** to notify that the speed is invalid, ending the activation process. If the validation does not fail, the *system* has to continue validating, within **5 milliseconds**, that no vehicle is within the minimum safety distance. Again, if the validation fails the *system* has **10 milliseconds** to notify that the distance is invalid, ending the activation process, but if the validation does not fail the *system* has to finish the activation process. At this moment there are two possibilities. If no vehicle is detected ahead, the *system* just displays within **5 milliseconds** the activation speed to be followed. However, if a vehicle within the minimum safety distance is detected, the *system* has **15 milliseconds** to reduce the engine torque and actuate the brakes in order to adapt the cruise speed to the speed of the vehicle ahead, and after that the *system* displays the speed adopted. Finally, after displaying the speed, the *driver* is allowed to deactivate the system at any moment.

7.1 ACC *C-O Diagrams*

In the following, we model this ACC contract with *C-O Diagrams* based on the specification given in Table 8. In Fig. 22 we show the top-level *C-O Diagram*, called *Adaptive_Cruise_Control*, starting the sequence from the permissions specified in clauses **1** and **2** (*Activate_System*), and composing the actions of activating the system with current speed or with a previously stored speed by means of an OR-refinement. After that, we have an AND-refinement considering the prohibition of deactivating the system as specified in clause **3** (*Deact_Forbidden*) and a general clause called *Speed_Val_Process* whose decomposition is explained in what follows.

The decomposition of clause *Speed_Val_Process* into subclauses can be seen in Fig. 23, where a SEQ-refinement is used in the decomposition. We have on the left-hand side the specification of the obligation specified in clause **5** (*Speed_Validation*).

TABLE 8
Norms of the *Adaptive Cruise Control* contract

N#	Agent	Modality	Action	Condition	Time
1	*Driver*	*Permission*	Can active the system with the current speed (a_1)	$\emptyset$	$\emptyset$
2	*Driver*	*Permission*	Can active the system with a previously stored speed (a_2)	$\emptyset$	$\emptyset$
3	*System*	*Prohibition*	Deactivates the system (a_3)	$\emptyset$	Ten milliseconds after activation (t_1)
4	*Driver*	*Permission*	Deactivates the system (a_4)	$\emptyset$	$\emptyset$
5	*System*	*Obligation*	Validates the car speed (a_5)	$\emptyset$	Five milliseconds after activation (t_2)
6	*System*	*Obligation*	Notifies invalid speed (a_6)	Invalid speed ($\neg g1$)	Ten milliseconds after speed validation (t_3)
7	*System*	*Obligation*	Validates distance (a_7)	Valid speed ($g1$)	Five milliseconds after speed validation (t_4)
8	*System*	*Obligation*	Notifies invalid distance (a_8)	$g1$ $\wedge$ Invalid distance ($\neg g2$)	Ten milliseconds after distance validation (t_5)
9	*System*	*Obligation*	Reduces engine torque (a_9)	$g1 \wedge g2 \wedge$ Vehicle ahead ($g3$)	Fifteen milliseconds after distance validation (t_6)
10	*System*	*Obligation*	Actuates brakes (a_{10})	$g1 \wedge g2 \wedge$ Vehicle ahead ($g3$)	Fifteen milliseconds after distance validation (t_7)
11	*System*	*Obligation*	Displays speed (a_{11})	$g1$ $\wedge$ Valid distance ($g2$)	Five milliseconds after setting speed (t_8)

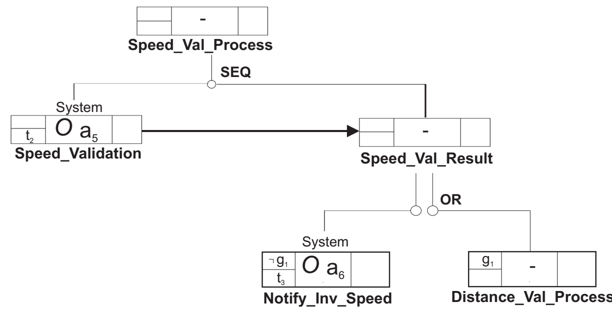

Fig. 23. Decomposition of *Speed_Val_Process*

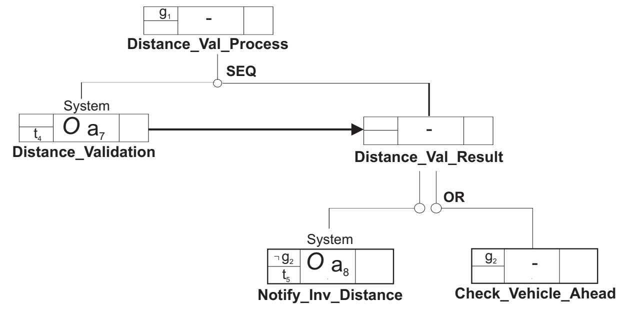

Fig. 24. Decomposition of *Distance_Val_Process*

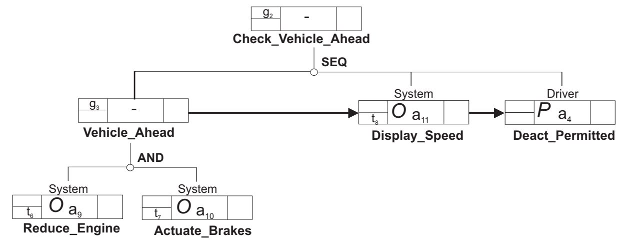

Fig. 25. Decomposition of *Check_Vehicle_Ahead*

On the right-hand side of the figure we have an OR-refinement of the obligation specified in clause **6** (*Notify_Inv_Speed*), applied if the speed is not valid, and a general clause called *Distance_Val_Process* which is applied otherwise.

The decomposition of clause *Distance_Val_Process* into subclauses is shown in Fig. 24, consisting of a SEQ-refinement. We have on the left-hand side the obligation specified in clause **7** (*Distance_Validation*), and on the right-hand side we have an OR-refinement consisting of the obligation specified in clause **8** (*Notify_Inv_Distance*), applied if the distance is not valid; the general clause *Check_Vehicle_Ahead* is applied otherwise.

Finally, Fig. 25 shows the decomposition of clause *Check_Vehicle_Ahead*, consisting of a SEQ-refinement. In this case the sequence starts with an AND-refinement considering the obligation specified in clause **9** (*Reduce_Engine*), and the obligation specified in clause **10** (*Actuate_Brakes*). These two obligations are applied only if a vehicle ahead is detected. After that, we have the obligation specified in clause **11** (*Display_Speed*), and at the end of the sequence we have the permission specified in clause **4** (*Deact_Permitted*).

7.2 ACC Validation and Verification

Once again, we obtain the network of timed automata corresponding to this contract by applying the transformation rules of the *C-O Diagrams* semantics and we implement these automata in the UPPAAL tool for the validation and verification of

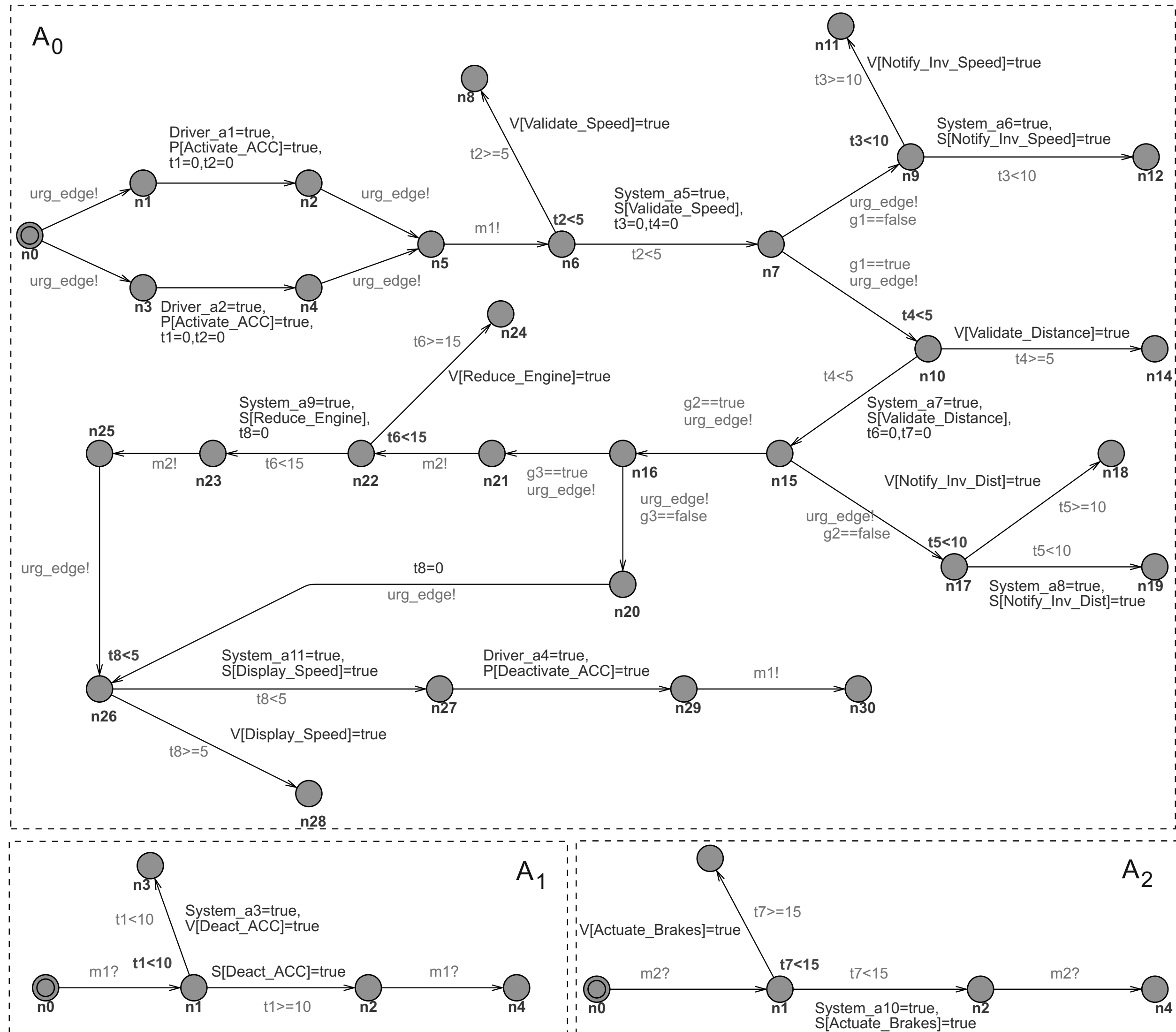

Fig. 26. Implementation of the ACC network of timed automata in UPPAAL

the contract, as in the OAP case study. A simplification of the automata implemented in UPPAAL can be seen in Fig. 26, where some nodes and invariants has been removed for the sake of readability.

As for the OAP case study, we use UPPAAL's simulator for validation purposes and its verifier to formally verify TCTL properties. We describe below the results of the verification of the properties we have checked.

- It is possible to reach a state where the system has been eventually activated and the speed adopted is displayed, that is, the obligation $Display_Speed$ has been satisfied. In UPPAAL:

$$E <> \quad S[Display_Speed] == true$$

  This property is **satisfied**.

- After the system has validated the speed ($S[Validated_Speed]$) it takes more than five milliseconds to validate the distance ($t4 > 5$), that is that the clause $Validate_Distance$ is violated. In UPPAAL:

$$S[Validate_Speed] == trueandt4 > 5 --> \\ V[Validate_Distance] == true$$

  This property is **satisfied**.

- There is a maximal path in which none of the main obligations and prohibitions of the contract has been been violated. In UPPAAL:

$$\begin{aligned} E[]\ \ (\ \ & V[Deact_ACC] == false \\ and\ \ & V[Validate_Speed] == false \\ and\ \ & V[Notify_Inv_Speed] == false \end{aligned}$$

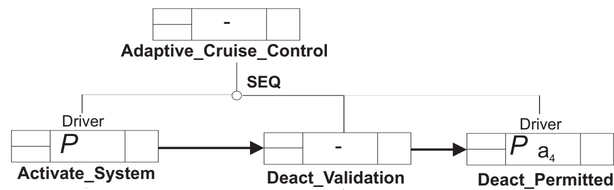

Fig. 27. Modification of the *Adaptive Cruise Control*

$$and \quad V[Validate_Distance] == false$$
$$and \quad V[Notify_Inv_Dist] == false$$
$$and \quad V[Reduce_Engine] == false$$
$$and \quad V[Actuate_Brakes] == false$$
$$and \quad V[Display_Speed] == false)$$

This property is **satisfied**.

- Finally, we have checked that the contract specification does not allow a state where both the permission and the prohibition of deactivating the system happen at the same time to be reached. In UPPAAL:

$$A[]\ not(P[Deactive_ACC]\ and\ S[Deact_ACC])$$

This property is **not satisfied**.

In the case of the last property we need to analyze the counter-example trace given by UPPAAL in order to determine where the problem lies (not only the reason but also at which level: original specification, *C-O Diagrams*, etc.). We identified that the problem was that the permission of deactivating the system could be enacted after all the activation process, and before the expiration of the prohibition of deactivation (this was a problem in the *C-O Diagrams*). This was solved by modifying the *C-O Diagrams* in such a way that the permission of deactivating the system be only enacted after the prohibition of deactivating has expired. The solution is shown in Fig. 27, where the $Deact_Permitted$ clause is now applied only after the $Deact_Validation$ process has finished (therefore the $Deact_Permitted$ clause is not a subclause of this process anymore). We have implemented the modified *C-O Diagrams*, and re-verified that all the properties are then satisfied.

8 Related Work

The use of deontic logic for reasoning about contracts is wide spread in the literature since it was first proposed in [8]. In [21] Governatori et al. provide a mechanism to check whether business processes are compliant with business contracts, using the logic FCL (based on deontic logic) to reason about contracts. In [22] Lomuscio et al. present an approach using WS-BPEL to specify both all possible behaviors of each service and the contractually correct behaviors, translating these specifications into automata supported by the MCMAS model checker to verify the behaviors automatically, whereas in [23] they consider a service composition in OWL-S and check with MCMAS if the composition fulfills properties written in a formal language based on epistemic and deontic logic (restricted to obligations). None of the above allow the specification of real-time constraints.

The approach followed by *C-O Diagrams* is inspired by the formal language $\mathcal{CL}$ [24]. In $\mathcal{CL}$ a contract is also expressed as a composition of obligations, permissions and prohibitions over actions, and the way of specifying reparations is as in our model. However, $\mathcal{CL}$ does not support either the specification of agents nor timing constraints natively, so they have to be encoded in the definition of the actions. In [25] Solaiman et al. show how relevant parts of contracts can be described by means of Finite State Machines, which are used to detect undesirable ambiguities, whereas we check if the contract satisfies some properties using UPPAAL. In [26] Desai et al. also automate reasoning about the correctness of the contract specification, but in this case representing contracts formally as a set of commitments.

None of the previous works provide a visual model for the definition of contracts. However, several other works define a meta-model for the specification of e-contracts for the purpose of their enactment or enforcement. In [27] Chiu et al. present a meta-model for e-contract templates written in UML, where a template consists of a set of contract clauses of three different types: obligations, permissions and prohibitions. These clauses are later mapped into Event Condition Action (ECA) rules for contract enforcement purposes, but the templates do not include any kind of reparation or recovery associated to the clauses. In [28] Krishna et al. proposed another meta-model based on entity-relationship diagrams used to generate workflows supporting e-contract enactment. This meta-model includes clauses, activities, parties and the possibility of specifying exceptional behavior, but not real-time constraints. In [29] Rouached et al. propose a contract layered model for modelling and monitoring e-contracts. It consists of a business entities layer, a business actions layer, and a business rules layer. These three layers specify the parties, the actions and the clauses of the contract respectively, including the conditions under which these clauses

are executed. However, real-time restrictions are not included and the specification of the clauses follows an operational, not a deontic, approach. In [30] Heckel and Lohmann propose to visualize contracts by graph transformation rules over UML data models; their focus is on testing not formal verification as we do.

Finally, *C-O Diagrams* implements many of the desirable properties, presented in [31], of a good formal language for normative texts.

9 Conclusions

In this work we have developed a formal semantics for *C-O Diagrams* based on an extension of timed automata. We have also presented an implementation for the tool UPPAAL, and have validated and verified the contract against temporal properties written in TCTL. We have applied our approach to two different case studies, where our method has successfully identified problems in their specifications.

We are currently working on the development of an interface to depict *C-O Diagrams*, and implementing the transformation presented in section 5. This tool is being integrated as a plugin of the tool WST [19].

An interesting research direction is to use GF [32] to relate *C-O Diagrams* with natural language. This might be done by means of using controlled natural languages as an intermediate language and by encoding *C-O Diagrams* into GF as has been recently done for the language CL [33].

Acknowledgments

Partially supported by: i) The Spanish government (co financed by FEDER founds) with project TIN2012-36812-C02-02. ii) The project "Reliable Multilingual Digital Communication: Methods and Applications", funded by The Swedish Research Council (*Vetenskapsrådet*, project nr. 2012-5746).

References

[1] B. Meyer, "Design by Contract," Interactive Software Engineering Inc., Tech. Rep. TR-EI-12/CO, 1986.

[2] J. Hatcliff, G. Leavens, k. Leino, P. Mller, and M. Parkinson, "Behavioral Interface Specification Languages," School of EECS, Univ. of Central Florida, Tech. Rep. CS-TR-09-01, 2009.

[3] "ebXML: Electronic Business using eXtensible Markup Language," www.ebxml.org.

[4] "WSLA: Web Service Level Agreements," www.research.ibm.com/wsla/.

[5] "Web Services Agreement Specification (WS-Agreement)," http://www.ogf.org/documents/GFD.107.pdf.

[6] J. C. Okika and A. P. Ravn, "Classification of soa contract specification languages," in *ICWS'08*. IEEE Computer Society, 2008, pp. 433–440.

[7] P. McNamara, "Deontic Logic," in *Handbook of the History of Logic*. North-Holland Publishing, 2006, vol. 7, pp. 197–289.

[8] G. H. V. Wright, "Deontic logic," *Mind*, vol. 60, pp. 1–15, 1951.

[9] E. Martínez, G. Díaz, M. E. Cambronero, and G. Schneider, "A Model for Visual Specification of e-Contracts," in *IEEE SCC'10*. IEEE Computer Society, 2010, pp. 1–8.

[10] E. Martínez, G. Díaz, and M. E. Cambronero, "Contractually Compliant Service Compositions," in *ICSOC 2011*, 2011, pp. 636–644.

[11] E. M. Haber, Y. E. Ioannidis, and M. Livny, "Foundations of visual metaphors for schema display," *J. Intell. Inf. Syst.*, vol. 3, no. 3/4, pp. 263–298, 1994.

[12] D. Harel, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514–530, 1988.

[13] K. G. Larsen, Z. Pettersson, and Y. Wang, "UPPAAL in a Nutshell," *STTT: International Journal on Software Tools for Technlogy Transfer*, vol. 1, no. 1–2, pp. 134–152, 1997.

[14] R. Alur and D. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126(2), pp. 183–235, 1994.

[15] W. Penczek and A. Plrola, *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach.*, ser. Studies in Computational Intelligence. Springer, 2006, vol. 20.

[16] A. David, M. O. Mller, and W. Yi, "Verification of uml statechart with real-time extensions," in *FASE 2002*, ser. LNCS, vol. 2306. Springer-Verlag, 2003, pp. 218–232.

[17] W. Drfler, "The cartesian composition of automata," *Theory Comput. Syst.*, vol. 11, pp. 239–257, 1978.

[18] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," in *SFM*, ser. LNCS, vol. 3185. Springer, 2004, pp. 200–236.

[19] M. E. Cambronero, G. Díaz, E. Martínez, V. V. Ruiz, and M. L. Tobarra, "Wst: a tool supporting timed composite web services model transformation," *Simulation*, vol. 88, no. 3, pp. 349–364, 2012.

[20] G. Decker, "Design and Analysis of Process Choreographies," Ph.D. dissertation, Hasso Plattner Institute, Uiversity of Postdam, 2009.

[21] *Compliance checking between business processes and business contracts*. IEEE Computer Society, 2006.

[22] A. Lomuscio, H. Qu, and M. Solanki, "Towards verifying contract regulated service composition," in *ICWS 2008*. IEEE Computer Society, 2008, pp. 254–261.

[23] A. Lomuscio, H. Qu, and M. Solanki, "Towards verifying compliance in agent-based web service compositions," in *AAMAS (1)*. IFAAMAS, 2008, pp. 265–272.

[24] C. Prisacariu and G. Schneider, "CL: An Action-based Logic for Reasoning about Contracts," in *WOLLIC'09*, ser. LNCS, vol. 5514. Springer, 2009, pp. 335–349.

[25] E. Solaiman, C. Molina-Jimenez, and S. Shrivastava, "Model Checking Correctness Properties of Electronic Contracts," in *ICSOC'03*, ser. LNCS, vol. 2910. Springer, 2003, pp. 303–318.

[26] N. Desai, N. C. Narendra, and M. P. Singh, "Checking correctness of business contracts via commitments," *AAMAS (2)*, pp. 787–794, 2008.

[27] D. Chiu, S. Cheung, and S. Till, "A Three-Layer Architecture for E-Contract Enforcement in an E-Service Environment," in *HICSS*, 2003, pp. 74–83.

[28] P. Krishna, K. Karlapalem, and A. Dani, "From Contract to E-Contracts: Modeling and Enactment," *Information Technology and Management*, vol. 6, no. 4, pp. 363–387, 2005.

[29] M. Rouached, O. Perrin, and C. Godart, "A Contract Layered Architecture for Regulating Cross-Organisational Business Processes," in *Business Process Management*, vol. 3649, 2005, pp. 410–415.

[30] R. Heckel and M. Lohmann, "Towards contract-based testing of web services," in *TACoS 2004*, ser. ENTCS, vol. 116, 2005, pp. 145–156.
[31] G. J. Pace and G. Schneider, "Challenges in the specification of full contracts," in *iFM'09*, ser. LNCS, vol. 5423, 2009, pp. 292–306.
[32] A. Ranta, *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications, 2011.
[33] K. Angelov, J. J. Camilleri, and G. Schneider, "A framework for conflict analysis of normative texts written in controlled natural language," *Journal of Logic and Algebraic Programming*, vol. 82, no. 5-7, pp. 216–240, 2013.

Gregorio Díaz is an associate professor in Computer Science at the University of Castilla-La Mancha since 2009 obtaining the tenure distinction in 2011. He obtained his PhD degree in 2006 and was an assistant professor for several years in the same university. His research goals are aimed to make software more reliable, more secure, and easier to design. His primary technical interests include software engineering and related areas, including contract specification, program monitoring, testing, and verification. His research combines strong theoretical foundations with realistic experimentation in the area of web services and cloud computing. More information is available on his homepage: http://www.dsi.uclm.es/personal/GregorioDiaz/.

Maria E. Cambronero is an associate professor in Computer Science at Castilla-La Mancha University, Spain, obtaining the tenure distinction in 2012. She received her PhD in 2009 from the University of Castilla-La Mancha for her work on modelling, validation and verification of real-timed web services systems. Her research interests include the application of theoretical foundations and software engineering for contracts specification and distributed real-timed systems, especifically, web services and cloud computing.

Enrique Martínez received the Computer Science degree from the University of Castilla-La Mancha in 2007 and the PhD degree from the same university in 2011. He is now working at private sector as a bank analyst programmer. His research interests include formal methods, electronic contracts and software engineering, and their application to service oriented architecture and software product lines. More information is available on his homepage: http://www.dsi.uclm.es/personal/47075509/index_english.html

Gerardo Schneider received the PhD degree in computer science from the University Joseph Fourier (thesis done at the VERIMAG laboratory), Grenoble (France), in 2002. From 2003 till 2009 he was a researcher at Uppsala University (Sweden), Irisa/INRIA Rennes (France), and the University of Oslo (Norway). Since July 2009, he has been an associate professor in computer science at the University of Gothenburg in Sweden. His research interests include software verification (real-time and runtime verification, model checking, testing), and the specification and analysis of contracts.