Proceedings
Fifth Workshop on
Formal Languages and Analysis of
Contract-Oriented Software
(FLACOS'11)

Málaga, Spain, 22nd and 23rd September 2011

Edited by Ernesto Pimentel and Valentín Valero
Index

Preface
Ernesto Pimentel and Valentín Valero 1

Invited Presentation: Behaviour-Aware Generation of Mashups
Antonio Brogi 3

Invited Presentation: Contracts as Dynamic Condition Response Graphs
Thomas Hildebrandt 5

Timed Automata Semantics for Visual e-Contracts
Enrique Martínez, M. Emilia Cambronero, Gregorio Díaz and Gerardo Schneider 7

Distributed System Contract Monitoring
Adrian Francalanza Ph.D, Andrew Gauci M.Sc and Gordon Pace Ph.D 23

Handling Conflicts in Depth-First Search for LTL Tableau to Debug Compliance Based Languages
Francois Hantry and Mohand-Said Hacid 39

From Contracts in Structured English to CL Specifications
Seyed M. Montazeri, Nivir K.S. Roy and Gerardo Schneider 55

A Software Tool for Legal Drafting
Daniel Gorín, Sergio Mera and Fernando Schapachnik 71

Contract-Based Discovery in Sensor Web
Javier Cubo, Nadia Gámez, José Antonio Martin, Lidia Fuentes 87
Preface

This volume includes the proceedings of the 5th Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11). The FLACOS Workshops serve as annual meeting places to bring together researchers and practitioners working on language-based solutions to contract-oriented software development. High-level models of contracts are needed as a tool to negotiate contracts and provide services conforming to them. This Workshop provides language-based solutions to the above issues through formalization of contracts, design of appropriate abstraction mechanisms, and formal analysis of contract languages and software. Detailed information about the FLACOS 2011 Workshop can be found at http://flacos2011.lcc.uma.es/. The 5th edition of the FLACOS Workshop was organized by the University of Málaga. It took place in Malaga, Spain, during September 22-23, 2011.

The program of this edition consisted of 5 regular papers, selected by the following international Program Committee:

- B. Bjurling, SICS, Sweden
- A. Brogi, University of Pisa, Italy
- S. Graf, VERIMAG, France
- A. Lomuscio, Imperial College London, United Kingdom
- U. Montanari, University of Pisa, Italy
- O. Owe, University of Oslo, Norway
- G. Pace, University of Malta, Malta
- E. Pimentel, University of Málaga, Spain (program co-chair)
- A.P. Ravn, Aalborg University, Denmark
- W. Reisig, Humboldt University, Berlin, Germany
- G. Salaün, Grenoble INP - INRIA - LIG, France
- G. Schneider, University of Gothenburg, Sweden
- K. Sere, Åbo Akademi University, Finland
- V. Valero, University of Castilla-La Mancha, Spain (program co-chair)
- M. Wirsing, Ludwig-Maximilians University, Munich, Germany

The selected papers tackle different issues of great relevance for the FLACOS community: semantics of visual contracts, distributed monitoring of contracts, derivation of formal specifications from natural language descriptions of contracts, automated support for legal drafting, evolution of rules in contract and service-based systems. The workshop also included two invited presentations by Antonio Brogi and Thomas Hildebrandt.

The Organizing Committee was chaired by Ernesto Pimentel, from the University of Málaga, together with the following team: Carlos Canal, Javier Cámara, Javier Cubo, J. Antonio Martín, and Meriem Quederni.

We would like to thank all the members of the program committee for their great work during the review process, the authors for submitting papers, and the participants for attending the workshop in Málaga.

Ernesto Pimentel
Valentin Valero
Invited Presentation: 
Behaviour-Aware Generation of Mashups

Antonio Brogi (University of Pisa, Italy)

So called service behaviour — i.e., the (partial) order according to which service operations are invoked and offered — plays an important role in the generation of correct service compositions and mashups. In this talk we discuss how behavioural information can be simply included in service descriptions and fruitfully exploited to verify the correctness of service mashups. We will exemplify some of the advantages of including behavioural information in service descriptions by illustrating a concrete example of design of a plug-in for an existing service composition framework. The plug-in analyses service behaviour information — represented as simple constraints overs service operations — to support a behaviour-aware generation of mashups.
Invited Presentation:
Contracts as Dynamic Condition Response Graphs

Thomas Hildebrandt (IT University of Copenhagen, Denmark)

The declarative process language Dynamic Condition Response (DCR) Graphs has been developed in the Trustworthy Pervasive Healthcare Services (TrustCare) research project as a formalization of the workflow language employed by the industrial partner, Resultmaker. It serves both as a formal specification/contract language and as an executable workflow language. In the talk we present the formalism and give examples of its applications. In particular we present a recent technique for synthesizing a set of local, communicating DCR Graphs from a global process description. We will also present ongoing work on extending DCR Graphs with time, data and exceptions/compensations.
Timed Automata Semantics for Visual e-Contracts

Enrique Martínez, M. Emilia Cambronero, Gregorio Díaz
Department of Computer Science
University of Castilla-La Mancha
Albacete, Spain
{emartinez, gregorio, emicp}@dsi.uclm.es

Gerardo Schneider
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
Department of Informatics
University of Oslo, Norway
gersch@chalmers.se

C-O Diagrams have been introduced as a means to have a more visual representation of electronic contracts, where it is possible to represent the obligations, permissions and prohibitions of the different signatories, as well as what are the penalties in case of not 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 an ordering of states and edges in order to represent different deontic modalities.

1 Introduction

In the software context, the term contract has traditionally been used 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 done by Meyer in the context of the language Eiffel (programming-by-contracts, or design-by-contract) [10]. This notion of contracts basically relies on the Hoare’s notion of pre and post-conditions and invariants. Though this paradigm has proved 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, behavioural interfaces have been proposed to capture richer properties than simple pre and post-conditions [5]. Here it is possible to express contracts on the history of events, including causality properties. However, the approach is limited when it comes to contracts containing exceptional behaviour, since the focus is mainly on the interaction concerning expected (and prohibited) behaviour.

In the context of SOA, there are different service contract specification languages, like ebXML [4], WSLA [14], and WS-Agreement [13]. These standards and 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 behaviour is rather limited and the sub-languages used to specify, for instance, 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 [11] where a taxonomy is presented.
More recently, some researchers have investigated how to adapt deontic logic [9] to define (consistent) contracts targeted to software systems where the focus is on the normative notions of obligation, permission and prohibition, including sometimes exceptional cases (e.g., [12]). Independently of the application domain, there still is need to better fill the gap between a contract understood by non-experts in formal methods (for its use), its logical representation (for reasoning), and its internal machine-representation (for runtime monitoring, and to be manipulated by programmers). We see two possible ways to bridge this gap: i) to develop suitable techniques to get a good translation from contracts written in natural language into formal languages, and ii) to provide a graphical representation (and tools) to manipulate contracts at a high level, with formal semantics supporting automatic translation into the formal language. We take in this paper the second approach.

In [8] we have introduced C-O Diagrams, a graphical representation for contracts allowing the representation of complex clauses describing the obligations, permissions, and prohibitions of different signatories (as defined in deontic logic [9]), as well as reparations describing contractual clauses in case of not fulfillment of obligations and prohibitions. Besides, C-O Diagrams permit to define real-time constraints. In [7] some of the satisfaction rules needed to check if a timed automaton satisfies a C-O Diagram specification were defined. These rules were originally miscalled “formal semantics”. The goal of this paper is to further develop our previous work, in particular we present here a formal semantics for C-O Diagrams based on timed automata, extended with an ordering of states and edges.

The rest of the work is structured as follows: Section 2 presents C-O Diagrams and their syntax, Section 3 develops the formal semantics of C-O Diagrams, including its implementation in UPPAAL [6] and a small example. The work is concluded in Section 4.

2 C-O Diagrams Description and Syntax

In Fig. 1 we show the basic element of C-O Diagrams. It is called a box and it is divided into four fields. On the left-hand side of the box we specify the conditions and restrictions. 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, on the center, is the main field of the box, and it is 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 permission), considering the clause eventually satisfied if this reparation is satisfied. Each box has also 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 is the performer of the action.
These basic elements of C-O Diagrams can be refined by using AND/OR/SEQ refinements, as shown in Fig. 2. The aim of these refinements is to capture the hierarchical clause structure followed by most contracts. An **AND-refinement** means that all the subclauses must be satisfied in order to satisfy the parent clause. An **OR-refinement** 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** means that the norm specified in the target box (SubClause2 in Fig. 2) must be fulfilled after satisfying the norm specified in the source box (SubClause1 in Fig. 2). 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 example, in the right-hand side of Fig. 2 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.

It is only considered the specification of **atomic actions** in the P field of the leaf boxes of our diagrams. 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 actions. In Fig. 3 we can see an example about how to model these compound actions through refinements, given two atomic actions a and b.

The **deontic norms** (obligations, permissions and prohibitions) that are applied over these actions can be specified in any box of our C-O Diagrams, affecting all the actions in the leaf boxes that are descendants of this box. If it is the case that the box where we specify the deontic norm is a leaf, the norm only affects the atomic action we have in this box. It is used an upper case “O” to denote an obligation, an upper case “P” to denote a permission, and an upper case “F” to denote a prohibition (forbidden). These letters are written in the top left corner of field P.

The composition of deontic norms is also achieved by means of the different refinements we have in C-O Diagrams. Thus, an AND-refinement corresponds to the **conjunction** operator “∧” between norms, an OR-refinement corresponds to the **choice** operator “+” between norms, and a SEQ-refinement corresponds to the **sequence** operator “;” between norms. For example, we can imagine having 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 before through the different kinds of refinement (Fig. 4). However, the specification of deontic norms in our diagrams must fulfill the following rule: exactly one deontic norm must be specified in each one 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.
norms. We have also that agents are only 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.

We have given here an abridged description of C-O Diagrams. A more detail description can be found in [8], including a qualitative and quantitative evaluation, and a discussion on related work.

**Definition 1 (C-O Diagrams Syntax)** We consider a finite set of real-valued variables \( e \) standing for clocks, a finite set of non-negative integer-valued variables \( v \), a finite alphabet \( \Sigma \) for atomic actions, a finite set of identifiers \( A \) for agents, and another finite set of identifiers \( N \) for names. The greek letter \( \varepsilon \) means that and expression is not given, i.e., it is empty.

We use \( C \) to denote the contract modelled by a C-O Diagram. The diagram is defined by the following EBNF grammar:

\[
C \quad ::= \quad (\text{agent, name, } g, tr, O(C_2), R) | \\
(\text{agent, name, } g, tr, P(C_2), \varepsilon) | \\
(\text{agent, name, } g, tr, F(C_2), R) | \\
(\varepsilon, name, g, tr, C_1, \varepsilon) \\
C_1 \quad ::= \quad C (\text{And } C) \, | \, C (\text{Or } C) \, | \, C (\text{Seq } C) \\
C_2 \quad ::= \quad a \, | \, C_3 (\text{And } C_3) \, | \, C_3 (\text{Or } C_3) \, | \, C_3 (\text{Seq } C_3) \\
C_3 \quad ::= \quad (\varepsilon, name, \varepsilon, \varepsilon, C_2, \varepsilon) \\
R \quad ::= \quad C \, | \, \varepsilon
\]

where \( a \in \Sigma \), \( \text{agent} \in A \) and \( \text{name} \in N \). Guard \( g \) is \( \varepsilon \) or a conjunctive formula of atomic constraints of the form: \( v \sim n \) or \( v - w \sim n \), for \( v, w \in \Sigma \), \( \sim \in \{ \leq, =, >, \geq \} \) and \( n \in \mathbb{N} \), whereas timed restriction \( tr \) is \( \varepsilon \) or a conjunctive formula of atomic constraints of the form: \( x \sim n \) or \( x - y \sim n \), for \( x, y \in e \), \( \sim \in \{ \leq, =, >, \geq \} \) 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 and Seq are the operators corresponding to the refinements we have in C-O Diagrams, AND-refinement, OR-refinement and SEQ-refinement, respectively.

The simplest contract we can have in C-O Diagrams is that composed 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 atomic 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 different refinements we have in C-O Diagrams. In the box where we have the refinement into \( C_1 \) we cannot specify an agent nor a reparation because these elements are always related to a single deontic norm, but we still can specify a guard \( g \) and a time restriction \( tr \) that affect all the deontic norms we combine.

Once we write a deontic operator in a box of our diagram, we have two possibilities as we can see in the specification of \( C_2 \): we can just 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 we have that the subboxes \( (C_3) \) cannot define a new deontic operator as it has already been defined in the parent box (affecting all the subboxes).
3 C-O Diagrams Semantics

The C-O Diagrams semantics is defined by means of a transformation into a Network of Timed Automata (NTA), that is defined as a set of timed automata \([1,2]\) that run simultaneously, using the same set of clocks and variables, and synchronizing on the common actions.

In what follows we consider a finite set of real-valued variables \(C\) ranged over by \(x,y,\ldots\) standing for clocks, a finite set of non-negative integer-valued variables \(\mathcal{V}\), ranged over by \(v,w,\ldots\) and a finite alphabet \(\Sigma\) ranged over by \(a,b,\ldots\) standing for actions. We will use letters \(r,r',\ldots\) to denote sets of clocks. We will denote by \(\text{Assigns}\) the set of possible assignments, \(\text{Assigns} = \{v := \text{expr} \mid v \in \mathcal{V}\}\), where \(\text{expr}\) 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 \(x,y \in C, v,w \in \mathcal{V}, \sim \in \{\leq,\geq,>,>,=\}\) and \(n \in \mathbb{N}\). The set of guard or invariant conditions will be denoted by \(\mathcal{G}\), ranged over by \(g,g',\ldots\).

**Definition 2** (Timed Automaton)
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, \(E \subseteq N \times \mathcal{G} \times \Sigma \times \mathcal{P}(\text{Assigns}) \times 2^{C} \times N\) is the set of edges, where the subset of urgent edges is called \(E_u \subseteq E\), and they will graphically be distinguished as they will have their arrowhead painted in white. \(I : N \rightarrow \mathcal{G}\) is a function that assigns invariant conditions (which could be empty) to locations.

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

In an NTA 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 \(a,b,\ldots\). Synchronization actions, however, must be executed simultaneously by two automata, and they will be ranged over letters \(m,m',\ldots\) and come from the synchronization of two actions \(m!\) and \(m?\), executed from two different automata. Due to the lack of space, we refer the reader to [3] for the definition of the semantics of timed automaton and NTA.

To specify the C-O Diagrams semantics, we add the definition of two orderings, \(\prec_N\) and \(\prec_E\), where:

- \(\prec_N\) is a (strict, partial) ordering on \(N\) where \(n \prec_N n'\) means that node \(n\) is better than node \(n'\).
- \(\prec_E\) is a (strict, partial) ordering on \(E\) where \(e \prec_E e'\) means that edge \(e\) is better than edge \(e'\).

We also add a violation set \(V(n)\) associated to each node \(n\) in \(N\), that is the set of contractual obligations and prohibitions that are violated in \(n\).

**Definition 3** (Violation Set) Let us consider the set of contractual obligations and prohibitions \(\mathcal{CN}\) ranged over \(cn, cn',\ldots\) standing for identifiers of obligations and prohibitions. We write \(n \not\models cn\) to express that obligation or prohibition \(cn\) is violated in node \(n\). Therefore, the violation set is defined as \(V(n) = \{cn \mid cn \in \mathcal{CN} \text{ and } n \not\models cn\}\).

Another set called satisfaction set \(S(n)\) is also associated to each node \(n\) in \(N\). This set is composed by the contractual obligations and prohibitions that have already been satisfied in \(n\).

**Definition 4** (Satisfaction Set) Let us consider the set of contractual obligations and prohibitions \(\mathcal{COF}\) ranged over \(cof, cof',\ldots\) standing for identifiers of obligations and prohibitions. We write \(n \models cof\) to express that obligation or prohibition \(cof\) has been satisfied in node \(n\) (we consider a prohibition satisfied in node \(n\) if it has not been violated and cannot be violated anymore because the time frame specified for the prohibition has expired). Hence, the satisfaction set is defined as \(S(n) = \{cof \mid cof \in \mathcal{COF} \text{ and } n \models cof\}\).
Once these two sets have been defined, we can formally define the **ordering on nodes** $\prec_N$, by comparing the violation sets and the satisfaction sets of the nodes, and the **ordering on edges** $\prec_E$, by comparing the violation sets and the satisfaction sets of the target nodes of the edges.

**Definition 5** (Ordering on Nodes) A node $n_1$ is better than another node $n_2$ if the violation set of $n_1$ is a proper subset of the violation set of $n_2$ or, if the violation sets are the same, a node $n_1$ is better than another node $n_2$ if the satisfaction set of $n_1$ is a proper superset of the satisfaction set of $n_2$. That is, $n_1 \prec_N n_2$ iff $(V(n_1) \subset V(n_2))$ or $(V(n_1) = V(n_2)$ and $S(n_1) \supset S(n_2))$.

**Definition 6** (Ordering on Edges) An edge $e_1$ is better than another edge $e_2$ if the source node is the same in both cases but the violation set of the target node of $e_1$ is a proper subset of the violation set of the target node of $e_2$ or, if the violation sets are the same, an edge $e_1$ is better than another edge $e_2$ if the satisfaction set of the target node of $e_1$ is a proper superset of the satisfaction set of the target node of $e_2$. Considering $e_1 = (n_1, g_1, a_1, s_1, r_1, n_1')$ and $e_2 = (n_2, g_2, a_2, s_2, r_2, n_2')$, $e_1 \prec_E e_2$ iff $(n_1 = n_2$ and $(V(n_1) \subset V(n_2'))$ or $(V(n_1') = V(n_2')$ and $S(n_1') \supset S(n_2'))$.

Finally, another set called **permission set** $P(n)$ is associated to each node $n$ in $N$. This set influences neither the ordering on nodes nor the ordering on edges, it is used just to record the permissions in the contract that have been made effective.

**Definition 7** (Permission Set) Let us consider the set of contractual permissions $CP$ ranged over $cp, cp', \ldots$ standing for identifiers of permissions. We write $n \models cp$ to express that permission $cp$ has already been made effective in node $n$. Then, the permission set is defined as $P(n) = \{ cp \mid cp \in CP \text{ and } n \models cp \}$.

Graphically, when we draw a timed automaton extended with these three sets, we write under each node $n$ between braces its violation set $V(n)$ on the left, its satisfaction set $S(n)$ on the centre and its permission set $P(n)$ on the right. In the initial node of the automata we build corresponding to C-O Diagrams these three sets are empty. By default, a node keeps in these sets the same content of the previous node when we compose the automata. Only in a few cases the content of these sets is modified (when an obligation or a prohibition is violated, an obligation or a prohibition is satisfied or a permission is made effective).

Concerning the **real-time restrictions** $tr$ specified in the contract, the two types of time restrictions we can have in C-O Diagrams must be translated in a different way for their inclusion into a timed automaton construction:

- A time restriction specified using **absolute time** must be specified in timed automata by rewriting the terms in which absolute time references occur. For that purpose we define a global clock $T \in \mathcal{C}$ that is never reset during the execution of the automata and, taking into account the moment at which the contract is enacted, we rewrite the absolute time references as deadlines involving clock $T$ and considering the smallest time unit needed in the contract. For example, 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 $(T \geq 5 \text{ and } T \leq 10)$.

- A time restriction specified using **relative time** must be specified in timed automata by introducing an additional clock to register the amount of time that has elapsed since another clause has been satisfied, resetting the additional clock value when this happens and specifying the deadline using it. 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 $C_{add} = \{ t_{name} \mid t_{name} \in \mathcal{C} \}$ including
Figure 5: Automata corresponding to a simple action \( a \) and to compound actions

A clock for every clause that is used as reference in the time restriction of at least another clause. For example, let us consider a contract with a clause that must be satisfied between 5 and 10 days after another clause \( \text{name1} \) has been satisfied. In this case we define an additional clock \( t_{\text{name1}} \) that is reset to zero when clause \( \text{name1} \) is satisfied \( (t_{\text{name1}} := 0) \) and the time restriction of the other clause is written as \( (t_{\text{name1}} \geq 5) \) and \( (t_{\text{name1}} \leq 10) \).

As a result, the set of clocks of the timed automata would be \( \mathcal{C} = \{ T \} \cup \mathcal{C}_{\text{add}} \). When we construct the timed automata corresponding to C-O Diagrams, we always consider \( (x \geq tr) \) and \( (x \leq tr) \) as the interval corresponding to the time restriction \( tr \) of the clause, where \( x \in \mathcal{C} \) is the clock used for its specification (\( x = T \) in the case of absolute time and \( x = t_{\text{name}} \) in the case of relative time, being \( \text{name} \) the clause used as reference), \( tr \in \mathbb{N} \) is the beginning of the interval and \( tr \in \mathbb{N} \) is the end of the interval \((tr \leq tr)\). If \( tr \) does not define the lower bound of the interval we take \( tr = 0 \), if \( tr \) does not define the upper bound of the interval we take \( tr = \infty \), and if \( tr = \varepsilon \) we take \( tr = 0 \), \( tr = \infty \) and \( tr = T \).

Once we have given these extensions of the definition of timed automata and we have explained how the different kinds of time restriction can be expressed, considering all the different elements we can specify in a C-O Diagram, we can define the transformation of the diagrams into timed automata by induction using several transformation rules.

**Definition 8 (C-O Diagrams Transformation Rules: Part I)**

1. An atomic action in a C-O Diagram, that is, \( (\varepsilon, \text{name}, \varepsilon, \varepsilon, a, \varepsilon) \) corresponds to the timed automaton \( A = (N_A, n_{0_A}, E_A, I_A) \), where:
   - \( N_A = \{ a_{\text{init}}, a_{\text{end}} \} \).
   - \( n_{0_A} = a_{\text{init}} \).
   - \( E_A = \{ a_{\text{init}} \xrightarrow{a} a_{\text{end}} \} \).
   - \( I_A = \emptyset \).
   The violation \( V \), satisfaction \( S \) and permission \( P \) sets are not modified, so \( V(a_{\text{init}}) = V(a_{\text{end}}) \), \( S(a_{\text{init}}) = S(a_{\text{end}}) \) and \( P(a_{\text{init}}) = P(a_{\text{end}}) \). This timed automaton can be seen in Fig. 5(A).

2. A compound action in a C-O Diagram where an AND-refinement is used to compose actions, that is, \( (\varepsilon, \text{name}, \varepsilon, \varepsilon, C_1 \And C_2 \And \ldots \And C_n, \varepsilon) \) corresponds to the cartesian product of the automata corresponding to each one of the subcontracts. Let us consider \( A, B, \ldots, Z \) the automata corresponding to the subcontracts \( C_1, C_2, \ldots, C_n \) (the actions specified in these subcontracts can be atomic actions or other compound actions). The resulting automaton \( A \And B \And \ldots \And Z \) corresponds to the cartesian product of these automata, that is, \( A \And B \And \ldots \And Z \). Again, the violation \( V \), satisfaction \( S \) and permission \( P \) sets are not modified, so they are the same in all the nodes. This composition of timed automata is shown graphically in Fig. 5(B).
(3) A **compound action** in a C-O Diagram where an **OR-refinement** is used to compose actions, that is, \((\varepsilon, \text{name}, \varepsilon, C_1 \text{ Or } C_2 \text{ Or } \ldots \text{ Or } C_n, \varepsilon)\) corresponds to a new automaton in which the automata corresponding to each one of the subcontracts is considered as an alternative. Let us consider \(A, B, \ldots, Z\) the automata corresponding to the subcontracts \(C_1, C_2, \ldots, C_n\) (the actions specified in these subcontracts can be atomic actions or other compound actions). The resulting automaton **OR** preserves the structure of the automata we are composing but adding a new initial node \(\text{OR}_{\text{init}}\) and connecting this node by means of urgent edges performing no action to the initial nodes of \(A, B, \ldots, Z\) (\(A_{\text{init}}, B_{\text{init}}, \ldots, Z_{\text{init}}\)). It is also added a new ending node \(\text{OR}_{\text{end}}\) and urgent edges performing no action from the ending nodes of \(A, B, \ldots, Z\) (\(A_{\text{end}}, B_{\text{end}}, \ldots, Z_{\text{end}}\)) to this new ending node. Let \(A = (N_A, n_{0_A}, E_A, I_A), B = (N_B, n_{0_B}, E_B, I_B), \ldots, Z = (N_Z, n_{0_Z}, E_Z, I_Z)\). The resulting automaton is therefore \(OR = (N_{\text{OR}}, n_{0_{\text{OR}}}, E_{\text{OR}}, I_{\text{OR}})\), where:

- \(N_{\text{OR}} = N_A \cup N_B \cup \ldots \cup N_Z \cup \{\text{OR}_{\text{init}}, \text{OR}_{\text{end}}\}\)
- \(n_{0_{\text{OR}}} = \text{OR}_{\text{init}}\)
- \(E_{\text{OR}} = E_A \cup E_B \cup \ldots \cup E_Z \cup \{\text{OR}_{\text{init}} \rightarrow \text{u } A_{\text{init}}, \text{OR}_{\text{init}} \rightarrow \text{u } B_{\text{init}}, \ldots, \text{OR}_{\text{init}} \rightarrow \text{u } Z_{\text{init}}\}\)
- \(I_{\text{OR}} = I_A \cup I_B \cup \ldots \cup I_Z\).

The violation (V), satisfaction (S) and permission (P) sets are not modified, so they are the same in all the nodes. This composition of timed automata is shown graphically in Fig. 5(C).

(4) A **compound action** in a C-O Diagram where a **SEQ-refinement** is used to compose actions, that is, \((\varepsilon, \text{name}, \varepsilon, C_1 \text{ Seq } C_2 \text{ Seq } \ldots \text{ Seq } C_n, \varepsilon)\) corresponds to a new automaton in which the automata corresponding to each one of the subcontracts are connected in sequence. Let us consider \(A, B, \ldots, Z\) the automata corresponding to the subcontracts \(C_1, C_2, \ldots, C_n\) (the actions specified in these subcontracts can be atomic actions or other compound actions). The resulting automaton **SEQ** preserves the structure of the automata we are composing, adding no extra nodes. We only connect with an urgent edge performing no action the ending node of each automaton in the sequence \((A_{\text{end}}, B_{\text{end}}, \ldots, Y_{\text{end}})\) with the initial node of the next automaton in the sequence \((B_{\text{init}}, C_{\text{init}}, \ldots, Z_{\text{init}})\). This rule is not applied in the cases of \(A_{\text{init}}\) (as there is not previous ending node to connect) and \(Z_{\text{end}}\) (as there is not following initial node to connect). Let \(A = (N_A, n_{0_A}, E_A, I_A), B = (N_B, n_{0_B}, E_B, I_B), \ldots, Z = (N_Z, n_{0_Z}, E_Z, I_Z)\). The resulting automaton is therefore **SEQ** = \((N_{\text{SEQ}}, n_{0_{\text{SEQ}}}, E_{\text{SEQ}}, I_{\text{SEQ}})\), where:

- \(N_{\text{SEQ}} = N_A \cup N_B \cup \ldots \cup N_Z\)
- \(n_{0_{\text{SEQ}}} = \text{A}_{\text{init}}\)
- \(E_{\text{SEQ}} = E_A \cup E_B \cup \ldots \cup E_Z \cup \{A_{\text{end}} \rightarrow \text{u } B_{\text{init}}, B_{\text{end}} \rightarrow \text{u } C_{\text{init}}, \ldots, Y_{\text{end}} \rightarrow \text{u } Z_{\text{init}}\}\)
- \(I_{\text{SEQ}} = I_A \cup I_B \cup \ldots \cup I_Z\).

Again, the violation (V), satisfaction (S) and permission (P) sets are not modified, so they are the same in all the nodes. This composition of timed automata is shown graphically in Fig. 5(D).

Until now, we have seen how the automata corresponding to the different actions (atomic or compound) specified in a C-O Diagram are constructed and we have seen that these translations do not modify the content of any of the sets (violation, satisfaction or permission). Next, we define the transformation rules specifying how these “action” automata are modified when we apply a deontic norm (obligation, permission or prohibition) over the actions in the C-O Diagram.

**Definition 7 (C-O Diagrams Transformation Rules: Part II)**
(5) The application of an obligation, a permission or a prohibition over an action in a C-O Diagram, i.e., \( \langle \text{agent}, \text{name}, \text{g}, \text{tr}, \text{O/P/F} \rangle \) corresponds to an automaton where the obligation/prohibition of performing the action specified in the subcontrakt \( C \) can be skipped, fulfilled or violated, whereas the permission of performing the action can be skipped, made effective or not made effective. Let us consider \( A = (N_A, n_0, E_A, I_A) \) the automaton corresponding to \( C \), being \( A_{\text{init}} \) the initial node and \( A_{\text{end}} \) the ending node. The resulting automaton \( D(A) \), where \( D \in \{ \text{O/P/F} \} \), preserves the structure of the automaton \( A \) but adding a new ending node \( A_{\text{time}} \) including the obligation over the action in its violation set, the prohibition over the action in its satisfaction set or nothing if a permission over the action is considered. If guard condition \( g \not\in \varepsilon \), we add another ending node \( A_{\text{skip}} \) where the violation, satisfaction and permission sets are not modified. We also include the obligation over the action in the satisfaction set of \( A_{\text{end}} \), the prohibition over the action in the violation set of \( A_{\text{end}} \) or the permission over the action in the permission set of \( A_{\text{end}} \). An invariant \( x \leq t_2 + 1 \) is added to each node of \( A \) except \( A_{\text{end}} \) and each edge performing one of the actions in this automaton is guarded with \( (x \geq t_1) \) and \( (x \leq t_2) \) and action performed by agent. New edges with \( x = t_2 + 1 \) and no action performed are added from each node of \( A \) except \( A_{\text{end}} \) to the new node \( A_{\text{time}} \) and, if guard condition \( g \not\in \varepsilon \), an urgent edge from \( A_{\text{init}} \) to \( A_{\text{skip}} \) is also added guarded with the guard condition of the clause negated \( \sim g \). Finally, if \( t_{\text{name}} \in \varepsilon \), all the edges reaching \( A_{\text{end}} \) reset \( t_{\text{name}} \) in the cases of obligation and permission, whereas all the edges reaching \( A_{\text{time}} \) reset \( t_{\text{name}} \) in the case of prohibition. Considering the more complex case, where \( g \not\in \varepsilon \) and \( t_{\text{name}} \in \varepsilon \), and having that \( g_1 \equiv (x \geq t_1) \) and \( (x \leq t_2) \) and \( g_2 \equiv x = t_2 + 1 \), the resulting automaton is therefore \( D(A) = (N_{D(A)}, n_{D_0(A)}, E_{D(A)}, I_{D(A)}) \), where:

- \( N_{D(A)} = N_A \cup \{ A_{\text{time}}, A_{\text{skip}} \} \).
- \( n_{D_0(A)} = A_{\text{init}} \).

\[
E_{D(A)} = \{ A_{\text{init}} \xrightarrow{g} A_{\text{skip}} \} \cup \left\{ \begin{array}{l}
\text{if } D = \text{O} \quad n \xrightarrow{g_1, \text{agent}(a)} n' \mid n \xrightarrow{a} n' \in E_A \text{ and } n' \neq A_{\text{end}}, \\
\text{if } D = \text{P} \quad n \xrightarrow{g_1, \text{agent}(a)} n' \mid n \xrightarrow{a} n' \in E_A \text{ and } n' \neq A_{\text{end}}, \\
\text{if } D = \text{F} \quad n \xrightarrow{g_2, \text{agent}(a)} n' \mid n \xrightarrow{a} n' \in E_A,
\end{array} \right.
\]

\[
I_{D(A)} = I_A \cup \{ I(n) \equiv x \leq t_2 + 1 \mid n \in N_A - \{ A_{\text{end}} \} \}.
\]

The resulting timed automata are shown graphically in Fig.6 where (A) corresponds to obligation, (B) corresponds to permission and (C) corresponds to prohibition. We consider one of the atomic actions included in the subcontrakt \( C \).

We can see that the above constructions can include a reparation contract \( R \) in the cases of obligation and prohibition. If this reparation is defined, we have to construct the automaton corresponding to the reparation contract and integrate this automaton as part of the automaton we have generated for the obligation or prohibition. This reparation contract removes the obliged or prohibited clause \( \text{name} \) from the violation set of the corresponding automaton, as we can see in Fig.5(D).

**Definition 7** (C-O Diagrams Transformation Rules: Part III)
A contract is breached. In the case of permission, as no reparation is defined, we have that the node violating the norm is a final node of the whole automaton construction where the ending nodes to this new node. Notice that in the case of obligation and prohibition, if there is no reparation possible, first we need to have only one ending node in the automata corresponding to the different obligations or prohibitions. Therefore, we add a new ending node in these automata and urgent edges from the old node we also have that the satisfaction set and the permission set are different from the ones we have in the initial node of the reparation because we have to include in the satisfaction set all the permissions that have been made effective in the reparation contract. Let us consider the obligation automaton \( O(A) \) be the automaton corresponding to the obligation automaton \( O(A) \) together with the reparation automaton \( R \). Considering the node with name in its violation \( (A_{\text{vio}}) \) set as the initial node of the reparation automaton \( (R_{\text{init}}) \). In the ending node of the reparation automaton \( (R_{\text{end}}) \) name is removed from the violation set, as the violation has been repaired. In this node we also have that the satisfaction set and the permission set are different from the ones we have in the initial node of the reparation because we have to include in the satisfaction set all the obligations and prohibitions satisfied in the reparation contract, and in the permission set all the permissions that have been made effective in the reparation contract. Let us consider \( D(A) = (N_{D(A)}, n_{0D(A)}, E_{D(A)}, I_{D(A)}) \), where \( D \in \{O, F\} \), and \( R = (N_{R}, n_{0R}, E_{R}, I_{R}) \). The resulting automaton is therefore \( D(A)_R = (N_{D(A)_R}, n_{0D(A)_R}, E_{D(A)_R}, I_{D(A)_R}) \), where:

- \( N_{D(A)_R} = N_{D(A)} \cup N_R - \{R_{\text{init}}\} \).
- \( N_{0D(A)_R} = A_{\text{init}} \).
- \( E_{D(A)_R} = E_{D(A)} \cup \{n \xrightarrow{g,D} n' | n \xrightarrow{g,R} n' \in E_R \text{ and } n \neq R_{\text{init}}\} \cup \{A_{\text{vio}} \xrightarrow{g,D} n' | n \xrightarrow{g,R} n' \in E_R \text{ and } n = R_{\text{init}}\} \).
- \( I_{D(A)_R} = I_{D(A)} - \{I(A_{\text{vio}})\} \cup \{I(n) | n \in N_R - \{R_{\text{init}}\}\} \cup \{I(A_{\text{vio}}) = I(R_{\text{init}})\} \).

Finally, we have to define the rules about how the automata corresponding to different deontic norms are composed when we have a composition of deontic norms in our C-O Diagram. To make this composition possible, first we need to have only one ending node in the automata corresponding to the different deontic norms. Therefore, we add a new ending node in these automata and urgent edges from the old ending nodes to this new node. Notice that in the case of obligation and prohibition, if there is no reparation defined, the node violating the norm is a final node of the whole automaton construction where the contract is breached. In the case of permission, as no reparation is defined, we have that \( P(A)_R = P(A) \).

**Definition 7 (C-O Diagrams Transformation Rules: Part IV)**

(7) Let \( D(A)_R = (N_{D(A)_R}, n_{0D(A)_R}, E_{D(A)_R}, I_{D(A)_R}) \), where \( D \in \{O, P, F\} \), be the automaton corresponding to an obligation, a prohibition or a permission in a C-O Diagram, specifying a reparation \( R \neq \varepsilon \) in the two first cases. The corresponding automaton with only one ending node, that we call \( A_{\text{final}} \),
and preserves the violation, satisfaction and permission sets of the previous node, is \( D(A)^r = (N_{D(A)^r}, n_{0,D(A)^r}, E_{D(A)^r}, I_{D(A)^r}) \), where:

- \( N_{D(A)^r} = N_{D(A)} \cup \{ A_{final} \} \).
- \( n_{0,D(A)^r} = n_{0,D(A)} \).
- \( E_{D(A)^r} = E_{D(A)} \cup \{ A_{skip} \rightarrow_a A_{final} \} \) if \( D = O \)
  \[ \begin{align*}
  & \begin{cases}
  A_{end} \rightarrow_a A_{final} & \text{if } D = O \\
  R_{end} \rightarrow_a A_{final} & \text{if } D = P \\
  A_{time} \rightarrow_a A_{final} & \text{if } D = F
  \end{cases}
  \end{align*} \]
- \( I_{D(A)^r} = I_{D(A)} \).

Therefore, the composition of the automata corresponding to different deontic norms is defined by three additional transformation rules.

**Definition 7 (C-O Diagrams Transformation Rules: Part V)**

(8) If several norms are composed by an **AND-refinement**, that is, we have specified the diagram \((\varepsilon, \text{name}, g, \text{tr}, C_1, \text{And} C_2, \text{And} \ldots \text{And} C_n, \varepsilon)\), their composition corresponds to a network of automata in which we consider all the norms we are composing in parallel. Let us consider \( \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n \) the automata corresponding to the norms we are composing. The resulting network of automata preserves the structure of the automata we are composing, adding to each one of them the additional nodes and edges necessary for synchronization (these nodes are called \( C_{init} \) and \( C_{final} \) in the first automaton, \( C_{syn} \) and \( C_{syn'} \); \( i = 1, \ldots, n - 1 \) in the other automata). Before its initial node, each automaton synchronizes with the other automata and it synchronizes again after its final node by means of urgent channels (\( m_1, m_2, \ldots, m_{n-1} \)). In the first automaton we add another node \( C_{skip} \) if guard condition of the parent clause \( g \neq \varepsilon \) and an urgent edge from \( C_{init} \) to this new node guarded with the guard condition negated (\( \neg g \)). In the final node of the first automaton the violation, satisfaction and permission sets are the union of the sets resulting in each one of the automata running in parallel, so we have that \( V_{final} = V_1 \cup V_2 \cup \ldots \cup V_n \), \( S_{final} = S_1 \cup S_2 \cup \ldots \cup S_n \) and \( P_{final} = P_1 \cup P_2 \cup \ldots \cup P_n \). If time restriction of the parent clause \( tr \neq \varepsilon \), we consider this additional time restriction in all the composed automata together with their own time restrictions. Let \( \varepsilon_1 = (N_{\varepsilon_1}, n_{0,\varepsilon_1}, E_{\varepsilon_1}, I_{\varepsilon_1}) \), \( \varepsilon_2 = (N_{\varepsilon_2}, n_{0,\varepsilon_2}, E_{\varepsilon_2}, I_{\varepsilon_2}) \), \ldots, \( \varepsilon_n = (N_{\varepsilon_n}, n_{0,\varepsilon_n}, E_{\varepsilon_n}, I_{\varepsilon_n}) \). Considering the case where \( g \neq \varepsilon \) and \( tr \neq \varepsilon \), and having that \( E_{\varepsilon_1*}, E_{\varepsilon_2*}, \ldots, E_{\varepsilon_n*} \) are the sets of edges considering time restriction \( tr \) together with their own time restriction, the resulting network of automata is therefore \( \varepsilon^{*} = (N_{\varepsilon^{*}}, n_{0,\varepsilon^{*}}, E_{\varepsilon^{*}}, I_{\varepsilon^{*}}) \), \( i = 1, \ldots, n \) where:

- \( N_{\varepsilon^{*}} = N_{\varepsilon_i} \cup \) \( \begin{cases}
  C_{init}, C_{final}, C_{skip} & \text{if } i = 1 \\
  C_{i_{syn}}, C_{i_{syn'}}, C_{i-1_{syn}}, C_{i-1_{syn'}} & \text{if } i = 2, \ldots, n - 1 \\
  C_{i_{1_{syn}}}, C_{i-1_{syn'}} & \text{if } i = n
  \end{cases} \)

- \( n_{0,\varepsilon^{*}} = \) \( \begin{cases}
  C_{init} & \text{if } i = 1 \\
  C_{i-1_{syn}}, C_{i_{1_{syn}}'} & \text{if } i = 2, \ldots, n
  \end{cases} \)

- \( E_{\varepsilon^{*}} = E_{\varepsilon_i} \cup \) \( \begin{cases}
  C_{init} \xrightarrow{m_1} C_{i_{syn}}, C_{init} \xrightarrow{m_1} C_{i_{syn'}}, C_{i-1_{syn}}, C_{i-1_{syn'}}, & \text{if } i = 1 \\
  C_{i_{syn}} \xrightarrow{m_1} C_{i_{syn'}}, C_{i_{syn}} \xrightarrow{m_1} C_{i_{syn'}}, & \text{if } i = 2, \ldots, n - 1 \\
  C_{i_{1_{syn}}}, C_{i_{1_{syn'}}} & \text{if } i = n
  \end{cases} \)

- \( I_{\varepsilon^{*}} = I_{\varepsilon_i} \cup \{ I(n) \equiv x \leq t2 + 1 \} n \in N_{\varepsilon_i} - \{ C_{final} \} \).
(9) If several norms are composed by an OR-refinement, that is, we have specified the diagram \( (\mathcal{E}, \text{name}, g, \text{tr}, C_1 \lor C_2 \lor \ldots \lor C_n, \mathcal{E}) \), their composition corresponds to an automaton in which the automata corresponding to each one of the norms is considered as an alternative. Let us consider \( \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n \) the automata corresponding to the norms we are composing. The resulting automaton OR* preserves the structure of the automata we are composing, adding two nodes called \( C_{\text{init}} \) and \( C_{\text{final}} \). We define an urgent edge performing no action for each one of the norms we are composing connecting \( C_{\text{init}} \) with the initial node of the automaton corresponding to the norm and we also define an urgent edge performing no action for each one of the norm we are composing connecting the final node of its automaton with \( C_{\text{final}} \). We add another node \( C_{\text{skip}} \) if guard condition of the parent clause \( g \neq \mathcal{E} \) and an urgent edge from \( C_{\text{init}} \) to this new node guarded with the guard condition negated \( (\neg g) \). In the final node of this new structure we keep the violation, satisfaction and permission sets of the previous final node, so we have that \( V_{\text{final}} = V_1 [V_2 | \ldots | V_n, S_{\text{final}} = S_1 | S_2 | \ldots | S_n \) and \( P_{\text{final}} = P_1 | P_2 | \ldots | P_n \). If time restriction of the parent clause \( \text{tr} \neq \mathcal{E} \), we consider this additional time restriction in all the composed automata together with their own time restrictions. Let \( \mathcal{E}_1 = (N_{\mathcal{E}_1}, n_{\mathcal{E}_1}, E_{\mathcal{E}_1}, I_{\mathcal{E}_1}), \mathcal{E}_2 = (N_{\mathcal{E}_2}, n_{\mathcal{E}_2}, E_{\mathcal{E}_2}, I_{\mathcal{E}_2}), \ldots, \mathcal{E}_n = (N_{\mathcal{E}_n}, n_{\mathcal{E}_n}, E_{\mathcal{E}_n}, I_{\mathcal{E}_n}) \). Considering the case where \( g \neq \mathcal{E} \) and \( \text{tr} \neq \mathcal{E} \), and having that \( E_{\mathcal{E}_1}, E_{\mathcal{E}_2}, \ldots, E_{\mathcal{E}_n} \) are the sets of edges considering time restriction \( \text{tr} \) together with their own time restriction, the resulting automaton is therefore \( \text{OR*} = (N_{\text{OR*}}, n_{\text{OR*}}, E_{\text{OR*}}, I_{\text{OR*}}) \), where:

- \( N_{\text{OR*}} = N_{\mathcal{E}_1} \cup N_{\mathcal{E}_2} \cup \ldots \cup N_{\mathcal{E}_n} \cup \{ C_{\text{init}}, C_{\text{final}}, C_{\text{skip}} \} \).
- \( n_{\text{OR*}} = C_{\text{init}} \).
- \( E_{\text{OR*}} = E_{\mathcal{E}_1} \cup E_{\mathcal{E}_2} \cup \ldots \cup E_{\mathcal{E}_n} \cup \{ C_{\text{init}} \rightarrow_u C_{\text{final}}, \text{init} \rightarrow_u C_{2\text{init}}, \ldots \} \).
- \( C_{\text{init}} \rightarrow_u C_{\text{init}} \cup \{ C_{\text{final}} \rightarrow_u C_{\text{final}} \}, \text{init} \rightarrow_u C_{\text{final}} \cup \{ C_{\text{final}} \rightarrow_u C_{\text{final}} \}, \ldots \).
This composition of timed automata is shown graphically in Fig. 7(B).

(10) If several norms are composed by a SEQ-refinement, that is, we have specified the diagram $(ε, name, g, tr; C_1 Seq C_2 Seq … Seq C_n, ε)$, their composition corresponds to an automaton in which the automata corresponding to each one of the norms are connected in sequence. Let us consider $C_1, C_2, …, C_n$ the automata corresponding to the norms we are composing. The resulting automaton $SEQ*$ preserves the structure of the automata we are composing, adding just one extra node $C_{skip}$ if guard condition of the parent clause $g ≠ ε$ and an urgent edge from $C_{init}$ to this new node guarded with the guard condition negated ($¬g$). We connect with an urgent edge performing no action the ending node of each automaton in the sequence $(C_{final1}, C_{final2}, …, C_{finaln−1})$ with the initial node of the next automaton $(C_{init2}, C_{init3}, …, C_{initn})$. This rule is not applied in the cases of $C_{init}$ (as there is not previous ending node to connect) and $C_{nfinal}$ (as there is not following initial node to connect). In the initial node of each one of the composed automata we preserve the violation, satisfaction and permission sets of the previous final node. If time restriction of the parent clause $tr ≠ ε$, we consider this additional time restriction in all the composed automata together with their own time restrictions. Let $C_1 = (N_{c1}, n_{init}, E_{c1}, I_{c1})$, $C_2 = (N_{c2}, n_{init}, E_{c2}, I_{c2})$, …, $C_n = (N_{cn}, n_{init}, E_{cn}, I_{cn})$. Considering the case where $g ≠ ε$ and $tr ≠ ε$, and having that $E_{c1} +, E_{c2} +, …, E_{cn} +$ are the sets of edges considering time restriction $tr$ together with their own time restriction, the resulting automaton is $SEQ* = (N_{SEQ*}, n_{SEQ*}, E_{SEQ*}, I_{SEQ*})$, where:

- $N_{SEQ*} = N_{c1} ∪ N_{c2} ∪ … ∪ N_{cn} ∪ \{C_{skip}\}$,
- $n_{SEQ*} = C_{init}$,
- $E_{SEQ*} = E_{c1} + ∪ E_{c2} + ∪ … ∪ E_{cn} + ∪ \{C_{init} \xrightarrow{\text{seq}} C_{skip}, C_{final} \xrightarrow{\text{seq}} C_{init}\}$,
- $I_{SEQ*} = I_{c1} ∪ \{I(n) \equiv x ≤ r1 + 1 | n ∈ N_{c1} − \{C_{final}\}\} ∪ I_{c2} ∪ \{I(n) \equiv x ≤ r2 + 1 | n ∈ N_{c2} − \{C_{final}\}\} ∪ … ∪ I_{cn} ∪ \{I(n) \equiv x ≤ rn + 1 | n ∈ N_{cn} − \{C_{final}\}\}$.

This composition of timed automata is shown graphically in Fig. 7(C).

3.1 Implementation in UPPAAL

The implementation of the NTAs we have obtained in UPPAAL is quite straightforward as both, the NTA formalism considered by the tool and the NTA formalism that we have considered, are very similar. There are only a few implementation points that need a more detailed explanation:

- First, as there is no way in UPPAAL of directly expressing that an edge without synchronisation should be taken without delay, that is, there are no urgent edges, we have to find an alternative way of encoding this behaviour. For this purpose we consider the modelling pattern proposed in [3].

The encoding of urgent edges introduces an extra automaton, that we call Urgent, with a single location and a self loop. The self loop synchronises on an urgent channel that we call arg_edge. An edge can now be made urgent by performing the complimentary action.

- The performance of actions by agents is implemented by means of boolean variables in UPPAAL. We define a boolean variable called agent_action for each one of the actions considered in the contract. These variables are initialized to false and, when one of the actions is performed by an agent in one of the edges, we update the value of the corresponding variable to true.

- Finally, 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 of them initialized to $\text{false}$. The size of the arrays $V$ and $S$ is 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 $\text{true}$ in the proper array the value of the index corresponding to the clause. In the case of repairing an obligation/prohibition violation, the index corresponding to the proper clause in $V$ is set to $\text{false}$.

### 3.2 Example: Online Auctioning Process

Let us consider part of a contract about an online auctioning process. It specifies that at the beginning of the process the seller has one day to upload valid information about the item he wants to sell, being forbidden the sale of inadequate items such as replicas of designers items or wild animals. We can identify in this specification an obligation, a prohibition and a real-time constraint affecting both norms. In the representation of this contract as a C-O Diagram, that can be seen in the left-hand side of Fig. 8, we have a main clause $\text{Check Item}$ including the time restriction one day, denoted as $t_1$. This main clause is decomposed by means of an AND-refinement, having on the one hand the clause with the prohibition, called $\text{Inadequate Item}$ and denoting the action as $a_2$, and on the other hand the clause with the obligation, called $\text{Valid Information}$ and denoting the action as $a_3$.

By following the C-O Diagrams semantics, we can obtain an NTA corresponding to the contract. Its implementation in the UPPAAL tool can be seen in the right-hand side of Fig. 8 having two automata running in parallel, one corresponding to the prohibition and the other one corresponding to the obligation. Now we can take advantage of all the mechanisms for simulation and formal verification provided by the tool to model-check the contract specification. As this is just a small part of a contract, the properties we can verify here are quite obvious. However, this verification process can be very useful over big contracts, verifying properties such as the violation of clauses when a time constraint expires, the possibility of satisfying the contract without violating any clause, etc.

For example, in the current NTA we can check the property that if the seller takes more than one day ($t_1 > 1$) to upload valid information about the item, the clause $\text{Valid Information}$ is always violated. This property is written as follows in the UPPAAL verifier:

$$A_{n_1} \text{and } t_1 > 1 \implies \neg V[\text{Valid Information}]$$

And we obtain that this property is satisfied.
4 Conclusions

In this work we have developed a formal semantics for C-O Diagrams based on timed automata extended with an ordering of states and edges in order to represent the different deontic modalities. We have also seen how these automata can be implemented in UPPAAAL in order to model-check the contract specification, and a small example has been provided.

As future work, we are working on several case studies in order to prove the usefulness of our approach to model-check the specification of complex contracts with real-time constraints. With these case studies we also want to check the complexity of the contracts we can deal with. Finally, we are working on the improvement of the satisfaction rules defined in [7] and their relationship with the C-O Diagrams formal semantics.

References

Distributed System Contract Monitoring

Adrian Francalanza Andrew Gauci Gordon J. Pace
Department of Computer Science, University of Malta ∗
{adrian.francalanza|agau0006|gordon.pace}@um.edu.mt

The use of behavioural contracts, to specify, regulate and verify systems, is particularly relevant to runtime monitoring of distributed systems. System distribution poses major challenges to contract monitoring, from monitoring-induced information leaks to computation load balancing, communication overheads and fault-tolerance. We present mDP, a location-aware process calculus, for reasoning about monitoring of distributed systems. We define a family of Labelled Transition Systems for this calculus, which allow formal reasoning about different monitoring strategies at different levels of abstractions. We also illustrate the expressivity of the calculus by showing how contracts in a simple contract language can be synthesised into different mDP monitors.

1 Introduction

As systems continue to grow in size and complexity, the use of behavioural contracts is becoming crucial in specifying, regulating and verifying correctness. Various notions of contracts have been used, but most prevailing variants enable the regulation of the behaviour of a system, possibly with consequences in case of violations. Such contracts can then be used in multiple ways, from system validation and verification, to conflict analysis of the contract itself. One important use of contracts is in runtime monitoring: system traces are analysed at runtime to ensure that any contract violating behaviour is truncated before it leads to any further consequences, possibly applying reparations to recover from anomalous states.

More and more systems are deployed in a distributed fashion, whether out of our choice or necessity. Distribution poses major design challenges for runtime monitoring of contracts, since monitors themselves can be distributed, and trace analysis can be carried out remotely across location. This impacts directly various aspects of the system being monitored, from the security of sensitive information, to resource management and load balancing, to aspects relating to fault tolerance. Various alternative solutions have been presented in the literature, from fully orchestrated solutions where monitors are located at a central location, to statically distributed monitors where the contract monitor is statically decomposed into different components hosted at the location where system traces are generated.

The primary contribution of this paper is a unified formal framework for studying different monitoring strategies. We present a location-aware calculus supporting explicit monitoring as a first class entity, and internalising behavioural traces at the operational level rather than at a meta-level. We show the expressivity of the calculus by using it to model different distributed system monitoring strategies from the literature. We also present a novel architecture in which contract monitors migrate across locations to keep information monitoring local, while limiting remote monitor instrumentation in certain situations. The versatility of the contract-supporting calculus is later illustrated by showing how it can model different instrumentation strategies. In particular, we show how behavioral contracts expressed using regular expressions can be automatically translated into monitors of different monitoring strategies.

∗The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship Scheme (Malta). The scholarship is part financed by the European Union European Social Fund.
The paper is organised as follows. In section 2, we outline the contract monitoring strategies for distributed systems from the literature. We present a monitoring distributed calculus in sections 3 and 4 and illustrate its use for monitoring behavioural contracts expressed as regular expressions in section 5. We discuss related work and conclude in section 6.

2 Monitoring Distributed Systems

Monitoring distributed systems is distinct from monolithic monitoring. These systems are usually characterised by the absence of a global clock when ordering events across location boundaries. They often consist of autonomous, concurrently executing subsystems communicating through message passing, each with its local memory, where communication across subsystems is considerably slower than local communication. The topology of such systems may sometimes change at runtime through the addition of new subsystems or the communication of private channels. Most internet-based and service-oriented systems, peer-to-peer systems and Enterprise Service Bus architectures [4] are instances of such systems.

These characteristics impinge on contract monitoring. For instance, the absence of a global clock prohibits precise monitoring for consequentiality contracts across locations [11]. Distribution impacts on information locality; subsystem events may contain confidential information which must not be exposed globally thereby requiring local monitoring. The possibility of distributing monitors also introduces concerns for monitor load balancing and monitor replication for fault tolerance.

Whether monitored contracts are known at compile time or else become known at runtime also affects distributed monitoring. Static contracts, ones which are fully known at compile time, are typically not expressive enough for distributed systems with dynamic topologies. Dynamic contracts, ones which are partially known at compile time, tend to be more appropriate for such systems. They are found in intrusion detection [8], where suspicious user behaviour is learnt at runtime, and in systems involving service discovery, where the chosen service may come with a fixed or negotiated contract upon discovery.

2.1 Classifying Distributed System Monitoring Approaches

Existing approaches for distributed system monitoring can be broadly classified into two categories: orchestration-based or choreography-based. Orchestration-based approaches relegate monitoring responsibility to a central monitor overhearing all necessary information whereas choreography-based typically distribute monitoring across the subsystems. Orchestration, used traditionally in monolithic systems, is relatively the simplest strategy and its centralisation facilitates the handling of dynamic contracts. The approach is however susceptible to data exposure when contacts concern private information; it also leads to considerable communication overhead across locations, and poses a security risk by exposing the monitor as a central point of attack. By contrast, choreography-based approaches push verification locally, potentially minimising data exposure and communication overhead. Communication between localised monitors is typically substantially less than that induced by the remote monitoring of a central monitor. Choreography is however more complex to instrument, as contracts need to be decomposed into coordinating local monitors, is more intrusive, burdening monitored subsystems with additional local computation, and is applicable only when the subsystems allow local instrumentation of monitoring code. Choreographed monitors are also instrumented upfront, which may lead to redundant local instrumentation in the case of consequential contracts; if monitoring at location $k$ is dependent on verification at location $l$, and the check at $l$ is never satisfied, upfront monitor instrumentation at $k$ is never needed.
Static orchestration verifying pre-determined contracts is a common approach, e.g., [3], where web-service compositions are monitored in an orchestrated fashion. By contrast, [11] uses orchestration to monitor for dynamic properties: web services are centrally monitored against BPMN workflow specifications, facilitating the verification of contracts (representing system properties) discovered at runtime on-the-fly. Extensive work has also been done in static choreography monitoring [13, 7, 10, 12, 14], where communication overhead is mitigated by breaking up contracts into parts which can be monitored independently and locally, synchronising between the monitors only when necessary. However, to the best of our knowledge, these approaches cannot fully handle dynamic contracts with runtime contract decomposition and distribution, nor do they tackle monitor-induced data exposure.

An alternative approach is that of using migrating monitors, which adequately supports dynamic contracts whilst still avoiding orchestration; in particular, it limits instrumentation of distributed monitors in cases where monitoring is dependent on computation. Using this approach, monitors reside where the immediate confidential traces reside, and migrate to other subsystems, possibly discovered at runtime, when information from elsewhere is required i.e., on a by-need basis. This enhanced expressivity also permits support for dynamic topologies and contracts learnt at runtime.

**Example 2.1.** Consider the hospital system contract:

A nurse will have access to a patient’s records after requesting them, as long as his or her request is approved by a doctor assigned to the patient.

We assume that (i) the nurse requests (and eventually accesses) the patient’s data from a handheld device, (ii) the information about which doctors have been assigned to which patients resides at the central site, and (iii) the patient’s information is stored on the doctors’ private clinic systems, where doctors can also allow nurses permission to access patients’ data.

A migrating monitor starts on the nurse’s system; upon receiving a patient-information request, it migrates to the hospital system, decomposes, and spreads to the systems of the patient’s assigned doctors to check for permissions allowing the nurse access to the records. Finally, if the permission is given, the decomposed monitors migrate back to the nurse’s device to check that the records are available. As with choreographed monitoring, and in contrast to orchestration, migrating monitors can ensure that monitoring is performed locally. The main difference is that instrumentation of monitors can be performed at runtime. For instance, when monitoring the hospital contract clause, no monitor is installed on a doctor’s system unless a nurse has made a request for information about a patient assigned to that doctor, which is less intrusive.

The added expressivity of migrating monitors requires a trust management infrastructure to ensure safe deployment of received monitors. Various solutions can be applied towards this end, from monitors signed by a trusted entity showing that they are the result of an approved contract negotiation procedure, to proof-carrying monitors which come with a proof guaranteeing what resources they access. This issue will not be discussed further here, but is crucial for the practicality of migrating monitors.

There are a number of issues relating to these different monitoring approaches that are unresolved. For instance, it is somewhat unclear, at least from a formal perspective, what added benefits migration brings to distributed monitoring. There are also issues relating to the monitoring of consequential properties across locations, which cannot be both sound and complete: in Example 2.1 by the time the monitor migrates to the doctor’s system, the doctor may have already approved the nurse’s request. Distribution precludes precise analysis of the relative timing of the traces, and one has the option of taking a worst or best case scenario, avoiding false positives or false negatives respectively. This problem is also prevalent in both orchestrated and choreographed approaches. We therefore require a common formal framework where all three approaches can be expressed. This would, in turn, permit rigorous analysis and evaluation with respect to these issues.
3 A Distributed Monitoring Language

We present mDPl, an adaptation of the distributed π-calculus in [9], where processes are partitioned across a flat organisation of locations; their behaviour is amenable to monitoring through traces, administered at a local level. The syntax, presented in Figure[1], assumes denumerable sets of channel names $c, d \in \text{CHANS}$, location names $l, k, h \in \text{LOCs}$, indices $n, m, o \in \text{IDX}$ and variables $x, y \in \text{VARS}$; identifiers $u, v$ range over \textsc{Ident}$ = \text{CHANS} \cup \text{LOCs} \cup \text{IDX} \cup \text{VARS}$, with identifier list $v_1, \ldots, v_n$ denoted as $\bar{v}$.

\begin{align*}
S, R \in \text{Sys} & \quad ::= \quad k \llbracket P \rrbracket \mid S \parallel R \mid \text{new} \, c. S \\
P, Q \in \text{Proc} & \quad ::= \quad \text{stop} \mid u!v. P \mid u?x. P \mid \text{new} \, c. P \mid \text{if } u=v \text{ then } P \text{ else } Q \mid P \parallel Q \mid \star \, P \mid [M]^{(l, n)} \mid T \\
T \in \text{Trc} & \quad ::= \quad t(c, d, n) \\
M, N \in \text{Mon} & \quad ::= \quad \text{stop} \mid u!v. M \mid u?x. M \mid \text{new} \, c. M \mid \text{if } u=v \text{ then } M \text{ else } N \mid M \parallel N \mid \star \, M \\
& \quad \mid \quad \mathbf{q}(c, \bar{x}). M \mid \text{sync}(u). M \mid \text{get}(x, y). M \mid \text{set}(u, v). M \mid \text{go} \, u. M \mid \text{ok} \mid \text{fail}
\end{align*}

Figure 1: mDPl Syntax

\textit{Systems}, $S, R$, are made up of \textit{located processes} $k \llbracket P \rrbracket$ (the tag $k$ denotes the current location hosting $P$) which can be composed in parallel and subject to scoping of channel names.

Located entities are partitioned into three syntactic categories: \textit{Processes}, Proc, comprise the standard communication constructs for output, $c!d.P$, and input, $c?\bar{x}.P$ (variables $\bar{x}$ are bound in the continuation $P$), together with the name-matching conditional, replication, parallel composition and name restriction; \textit{Traces}, are made up of individual trace entities, $t(c, d, n) \in \text{Trc}$ recording communication of values $d$ on channel $c$ at timestamp $n$ - they are meant to be ordered as a complete log recording past computation at a particular location; \textit{Monitors}, Mon, are similar in structure to processes, and are delimited at the process level by enclosing brackets $[M]^{(l, n)}$, where $(k, n)$ denotes the \textit{monitoring context} i.e., \textit{the location and log position of the trace being monitored}. In addition, they can:

- query traces for records of communication on channel $c$, $\mathbf{q}(c, \bar{x}). M$, where the index and location of the trace monitored is inferred from the enclosing monitoring context, $(k, n)$.
- get the information relating to the current monitoring context, $\text{get}(x, y). M$ ($x, y$ bound in $M$), and set the monitoring context to specific values $k$ and $n$, $\text{set}(k, n). M$, or else update to the current timestamp of a location $k$, $\text{sync}(k). M$,
- migrate to location $k$, $\text{go} \, k. M$, and
- report success, $\text{ok}$, or failure, $\text{fail}$.

\textbf{Shorthand:} We often elide trailing $\text{stop}$ processes. We thus represent asynchronous outputs such as $c!\bar{v}. \text{stop}$ as $c!\bar{v}$ and branches such as if $u=v$ then $P$ else $\text{stop}$ as if $u=v$ then $P$. We also denote $k \llbracket [M]^{(l, n)} \rrbracket$ as \textit{syntactic sugaring} for $k \llbracket [M]^{(l, n)} \rrbracket$.

Our calculus describes distributed, event-based, asynchronous monitoring. Monitoring is \textit{asynchronous} because it happens in two phases, whereby the operational mechanism for tracing is detached from the operational mechanism for querying the trace. This two-set setup closely reflects the limits imposed by a distributed setting and is more flexible with respect to the various monitoring mechanisms we
want to capture. Monitoring is _event-based_ because we chose only to focus on recording and analysing discrete events such as communication.

For simplicity, our calculus only records effected output process events in traces (which dually imply effected inputs); we can however extend traces to record other effects of computation in straightforward fashion. In order to extract realistic temporal ordering of traces across locations, the calculus provides two mechanisms for monitor re-alignment: the coarser (and real-time) _sync_ operation is used to start monitoring from a particular instant in time; the more explicit context update _setl_ enables hard-coded control of relative timing at the level of monitors. For instance, together with _getl_, it enables decomposed monitoring to hand over tracing at a specific index in a local trace. This mechanism gives more control and can improve distributed monitoring precision, but may also lead to unsound monitoring.

**Tracing:** Whenever communication occurs (possibly across two locations) on some channel _c_ with values _v_, a trace entity of the form _t(c,v,n)_ is produced at the location where the output resides. The output location, which keeps a local counter, assigns the timestamp _n_ to this trace entity and increments its counter to _n_ + 1. Local timestamps induce a partial-order amongst all trace entities _across the system_. In particular, we obtain a finite (totally ordered) chain of traces _per location_.

**Example 3.1** (Distributed Tracing). Consider the distributed system of outputs:

\[
\text{l}[](c_1!v_1) \parallel \text{l}[](c_2!v_2) \parallel \text{k}[](c_3!v_3)
\]

Assuming that locations _l_ and _k_ have the respective timestamp counters _n_ and _m_, once all outputs are consumed we can obtain either of the following possible sets of trace entities (for simplicity, we assume that _v_1 ≠ _v_2):

\[
\begin{align*}
\text{l}[](t(c_1,v_1,n)) &\parallel \text{l}[](t(c_2,v_2,n+1)) &\parallel \text{k}[](t(c_3,v_3,m)) \\
\text{l}[](t(c_1,v_1,n+1)) &\parallel \text{l}[](t(c_2,v_2,n)) &\parallel \text{k}[](t(c_3,v_3,m))
\end{align*}
\]

(1) (2)

The timestamps of trace-set (1) record the fact that the output on _c_1 was consumed before that on _c_2, whereas those of trace-set (2) record the opposite. However, in both of these trace-sets, the timestamp assigned to the trace-entity relating to _c_3, recorded at location _k_, does not indicate the order it was consumed, relative to the outputs on _c_1 and _c_2, which occurred at location _l_.

**Concurrent Monitoring:** In our model, traces may be queried by multiple concurrent entities, which allows for better separation of concerns when monitors are instrumented. Trace querying is performed exclusively by monitors, _l[](M)^(k,n)_ parameterised by the monitoring context (_k_, _n_); indicating that monitor _M_ is interested in analysing the _n_\(^{th}\) trace record at location _k_.

**Example 3.2** (Parallel Monitoring). Consider the trace-set (1) from Example 3.1. A (local) monitor determining whether, from timestamp _n_ onwards, a value _v_2 was communicated on the first output on channel _c_2 at location _l_, can be expressed as:

\[
\text{l}[](q(c_2,x).\text{if } x = v_2 \text{ then ok else fail})^{(l,n)}
\]

The counter _n_ of this monitor indicates that it starts analysing the trace-set (1) from the trace entity _l[](t(c_1,v_1,n))_ and continues moving up the chain of trace entities until the first trace entity describing outputs on channel _c_2 is encountered. Since _l[](t(c_1,v_1,n))_ states that the event occurred on another channel, namely _c_1, the monitor skips the irrelevant trace entity and its index is incremented to _n_ + 1 i.e.,
The monitor analyses the trace entity with the incremented timestamp \( n + 1 \) i.e., \( \text{l}[t(c_2, v_2, n + 1)] \), which happens to match the required event on channel \( c_2 \); the monitor thus substitutes \( v_2 \), obtained from the trace entity \( \text{l}[t(c_2, v_2, n + 1)] \), for \( x \) and proceeds with the monitor processing, which should eventually yield \( \text{ok} \). Traces can be concurrently queried by multiple monitors. For instance, consider another monitor running in parallel with the previous one of the form:

\[
\text{l}[q(c_1, x).q(c_2, y).if \ x = y \ then \ \text{ok} \ else \ \text{fail}]^{(l,n)}
\]

which checks that equal values are communicated on channels \( c_1 \) and \( c_2 \). Thus it is important that trace entities, such as \( \text{l}[t(c_2, v_2, n + 1)] \), are persistent and not consumed once analysed, as in the case of outputs in a message passing setting.

**Distributed Monitoring:** Distribution adds another dimension of complexity to monitor instrumentation, in terms of how to partition monitors across locations and how this partitioning evolves as computation progresses. In our calculus, remote querying can be syntactically expressed as \( k[q(c, x).M]^{(l,n)} \) for some \( c \) and \( n \), where \( k \neq l \). We can also describe the different classifications of distributed monitoring outlined earlier in Section 2.

**Example 3.3.** Consider a distributed system

\[
\text{Sys} \triangleq \text{l}[c_1!x.c_2!x] || \text{newd.}(k[d?x.c_2!x] || \text{l}[c_1!y.d!v.c_2?x.stop])
\]

where \( c_1!x.c_2!x \) and \( c_1!v.d!v.c_2?x.stop \) are located at \( l \) whereas \( d?x.c_2!x \) is located at \( k \). Moreover, process \( d?x.c_2!x \) shares a scoped channel \( d \) with \( c_1!v.d!v.c_2?x.stop \). For some timestamps \( n \) and \( m \), \( \text{Sys} \) non-deterministically produces either of the traces \( (3) \) or \( (4) \) below; the non-determinism is caused by the competition for the output on channel \( c_2 \) by respective inputs at \( l \) and \( k \):

\[
\text{newd.}(\text{l}[t(c_1, v, n)], \text{l}[t(d, v, n + 1)], \text{l}[t(c_2, v, n + 2)]) \quad (3)
\]

\[
\text{newd.}(\text{l}[t(c_1, v, n)], \text{l}[t(d, v, n + 1)], k[t(c_2, v, m)]) \quad (4)
\]

The preservation of the property ‘Whenever a value \( e \) is communicated on the first output on \( c_1 \) at location \( l \), then this value is not output on a subsequent output on channel \( c_2 \) at \( k \)’ in general cannot be adequately determined statically, due to the non-deterministic nature of the computation, as exhibited by the possible traces \( (3) \) and \( (4) \). However, the property can be monitored at runtime in a number of ways:

\[
M^{orch} \triangleq h[q(c, x).if \ x = e \ then \ sync(k).q(c, y).if \ x = y \ then \ fail]^{(l,n)}
\]

\[
M^{chor} \triangleq \text{newd}'.(\text{l}[q(c, x).if \ x = e \ then \ d'?x]^{(l,n)} || k[d'?x. sync(k).q(c, y).if \ x = y \ then \ fail]^{(k,m)})
\]

\[
M^{mig} \triangleq l[q(c, x).if \ x = e \ then \ go. sync(k).q(c, y).if \ x = y \ then \ fail]^{(l,n)}
\]

\( M^{orch} \) monitors this property in orchestrated fashion, querying traces at both \( l \) and \( k \) from a remote central location \( h \); this monitor is well-aligned with location \( l \) to start with, but has to explicitly re-align with location \( k \) once monitoring shifts to that location. \( M^{chor} \) is an instance of a choreographed monitor setup, instrumenting local monitors at each location where trace querying needs to be performed, namely \( l \) and \( k \). These local monitors synchronise between them using remote communication on the scoped channel \( d' \). Note that the monitor at \( k \) updates its context upon channel synchronisation on \( d' \) to ensure a temporal ordering on analysed trace records; without synchronisation, the monitor would potentially be reading past parts of the trace which may lead to unsound sequentiality conclusions. Finally, \( M^{mig} \) is a case of a migrating monitor, that starts monitoring at location \( l \) but then migrates to location \( k \) if it needs to continue monitoring there, re-aligning its index to that of the destination location.
All three distributed monitors in Example 3.3 are sound wrt. the property stated, in the sense that they never falsely flag a violation. They are nevertheless incomplete, and may miss out on detecting property violations. For instance, \( M_{\text{orch}} \) may realign with location \( k \) after the trace \( k\{t(c, v, m)\} \) is generated by \( k \), which sets the monitor timestamp index to \( (k, m+1) \). This forces the monitoring to start querying the trace at \( k \) from index \( m+1 \) and will therefore skip the relevant trace item \( k\{t(c, v, m)\} \). This aspect is however not a limitation of our encoding, but rather an inherent characteristic of distributed computing as discussed earlier in Section 2.

4 Monitoring Semantics

We define the semantics of mDP in terms of a number of related Labelled Transition Systems (LTSs), which are then used to compare systems through the standard notion of weak-bisimulation equivalence, denoted here as \( \approx \). This framework allows us to state and prove properties from a behavioural perspective about our monitored systems. For instance, we could express the fact that, ignoring monitoring location, \( M_{\text{orch}} \) and \( M_{\text{chor}} \) from Example 3.3 monitor for the same properties wrt. \( \text{Sys} \), using the statement:

\[
\text{Sys} \parallel M_{\text{orch}} \approx \text{Sys} \parallel M_{\text{chor}}
\]  

(5)

Using an LTS that does not express observable monitor actions, the property that a monitor, say \( M_{\text{orch}} \), does not affect the observable behaviour of the system \( \text{Sys} \) could be stated as:

\[
\text{Sys} \approx \text{Sys} \parallel M_{\text{orch}}
\]  

(6)

Using different LTSs, the same system could be assigned more restricted behaviour. For instance, this is useful to ensure that the monitor \( M_{\text{mig}} \) of Example 3.3 does not perform remote querying at any stage during its computation by establishing the comparison:

\[
\text{Sys} \parallel M_{\text{mig}} \approx ((\text{Sys} \parallel M_{\text{mig}}) \text{ without remote monitoring})
\]  

(7)

where the lefthand system is subject to an LTS allowing remote querying whereas the righthand monitor is subject to an LTS that prohibits it. Intuitively, if the behaviour is preserved when certain internal moves are prohibited, this means that these moves are not used (in any useful way) by the monitor.

4.1 Deriving LTSs Modularly

Closer inspection of the comparisons (5), (6) and (7) reveals that the different LTSs required are still expected to have substantial common structure; typically they would differ with respect to either the information carried by actions and/or the type of actions permitted. For instance, in (5) we would want actions that restrict information relating to the location of where monitoring is carried out, as this additional information would distinguish between the two monitors. On the other hand, for (7) we would want to prohibit actions relating to remote monitoring.

We therefore construct these related LTSs in modular fashion through the use of a \textit{preLTS}, i.e., an LTS whose transitions relate more systems, and whose action labels carry more information than actually needed. The excess transitions and label information are then pruned out as needed by a \textit{filter function} from actions in the \textit{preLTS} to actions in the LTS required.


### 4.2 A preLTS for mDP!

Our preLTS is defined over systems subject to a local logical clock at every location used by the system, which are used to generate ordered trace-entities and to re-align monitors. These clocks are modelled as monotonically increasing counters and expressed as a partial function $\delta \in \Delta :: \text{Locs} \rightarrow \mathbb{N}$, where $\delta(l)$ denotes the next timestamp to be assigned for a trace entity generated at $l$. Moreover, the counter increment is defined using standard function overriding, $\text{inc}(\delta, k) = \delta[k \mapsto (\delta(k) + 1)]$.

#### Figure 2: mDP preLTS main rules

A Configuration $C, D \in \text{Conf} :: \Delta \times \text{Sys}$ is thus a system subject to a set of localised counters. The preLTS is a ternary relation $\rightarrow :: \text{Conf} \times \text{Act} \times \text{Conf}$, denoted using the suggestive notation $C \xrightarrow{\mu} D$, where $\mu \in \text{Act}$ is a preLTS action label of the form, $\tau_y$, an internal action, $(b) c?d_y$, an output action, or $c?d_y$, an input action. In case of the output action, $\tilde{b}$ denotes the (possibly empty) set of channel names exported during an eventual interaction. These actions are standard \[\text{[9]},\] but are decorated with additional information $\gamma$ which can be of the following three formats:
\(\langle p : l, k \rangle\)– This states that it is a process \((p)\) action, involving locations \(l\) and \(k\).

\(\langle m : l, k \rangle\)– This states that it is a monitor \((m)\) action, involving locations \(l\) and \(k\).

\(\langle t : l, k : n \rangle\)– This states that it is a trace \((t)\) action at timestamp \(n\), involving locations \(l\) and \(k\).

The main rules defining the relation \(C \xrightarrow{\mu} D\) are outlined in Figure 2. The rule for process input, \(\text{InP}\), is standard, except for the additional label tag \(\langle p : k, l \rangle\) encoding the fact that the input is a process input, it resides at location \(l\), and is reading from some location \(k\) (when communication is local, then \(l = k\)). A central rule to our monitoring semantics is \(\text{OutP}\). Apart from the additional label decoration, it differs from standard output rules in two respects: first it generates a trace entity, \(k[\text{t}(c, d, \delta(k))]\), recording the channel name, \(c\), the values communicated, \(d\), timestamped by \(\delta(k)\), and second, it increments the clock at \(k\) once the trace entity is generated, necessary for generating a total order of trace-entities at \(k\). Monitor communication, defined by rules \(\text{OutM}\) and \(\text{InM}\), is similar albeit simpler since neither trace entities are generated, nor is the local counter updated\(^1\).

Rule \(\text{OutT}\) models trace actions as output labels with tags \(\langle t : k, l : n \rangle\), where the timestamp of the trace, \(n\), is recorded in the tag as well. Crucially, the trace entity is not consumed by the action (thereby acting as a broadcast), and its persistence allows for multiple monitors to query it. This action can be matched by a query action, \(\text{InT}\), expressed as an input action with a matching tag \(\langle t : k, l : n \rangle\) where the source location of the trace entity, \(k\), and time stamp \(n\) must match the current monitoring context \((k, n)\). Since the action describes the fact that a trace entity has been matched by the monitor query, the timestamp index of the monitoring context is incremented, \((k, n + 1)\) to progress to the next entity in the local trace log.

Scope extrusion of channel names may occur both directly, through process or monitor communication, or else indirectly through trace querying; these are both handled by the standard scoping rules \(\text{Open}\) and \(\text{Res}\). All three forms of communication, i.e., process, monitor and trace, are also handled uniformly, this time by the communication rule \(\text{ComI}\) (we here elide its symmetric rule). Communication yields a silent action \(\tau_\gamma\) that is decorated with the corresponding tagging information from the constituent input and output actions of the premises. This tagged information must match for both input and output actions and, in the case of the trace tags, \(\langle t : k, l : n \rangle\), this also implies a matching of the timestamp \(n\). When, for a particular timestamp, querying does not match the channel of the trace entity at that timestamp, rule \(\text{Sk}\) allows the monitor to increase its timestamp index and thus querying to move up the trace-chain at that location. Finally, \(\text{Sync}\) allows monitors to realign with a trace at a particular location, \(\text{GetI}\) and \(\text{SetI}\) allow for explicit manipulations of the monitoring context whereas \(\text{Go}\) describes monitor migration.

### 4.3 Filter Functions

Although necessary to encode extended information of system execution, the \(\text{preLTS}\) presented is too discriminating. For instance, the internal action \(\tau_\gamma\) is now compartmentalised into distinct silent actions, each identified by the tag information \(\gamma\), which complicates their use for weak actions when verifying bisimilar configurations. Similarly, external actions differentiating between a process or a monitor carrying out that action may also be deemed to discriminating. Finally, we may also want to disallow certain actions such as remote trace querying.

We obtain LTSs with the necessary level of discriminating actions using (i) the \(\text{preLTS}\) of Section 4.2 together with (i) a filter function, \(\Omega\). This function maps actions in the \(\text{preLTS}\), \(\mu \in \text{PACT}\), to actions in

\(^1\)Note that rule \(\text{OutM}\) refers to an indeterminate location \(h\), to match a reader in any such location.
Distributed System Contract Monitoring

the required LTS, \( \alpha \in \text{Act} \), through the rule:

\[
\text{Flt}_{\Omega} \quad \frac{C_1 \xrightarrow{\mu} C_2}{\frac{\alpha}{\Omega} C_1 \xrightarrow{\alpha} C_2} [\Omega(\mu) = \alpha]
\]

**Notation:** Note that filter function applications are essentially abstractions of the preLTS. LTSs obtained in this manner can effectively be indexed by their respective filter function, \( \Omega \), and for clarity we denote a configuration \( C \) subject to a behaviour obtained from the preLTS and a filter function \( \Omega \) as \( C_\Omega \). We also denote transitions obtained in this form as \( C_\frac{\alpha}{\Omega} C_2 \).

**Example 4.1 (Filter Functions).** Consider the following filter function definitions:

\[
\begin{align*}
\Omega_{\text{NTg}}(\tau_\gamma) & \triangleq \tau & \Omega_{\text{Prc}}(\tau_\gamma) & \triangleq \tau & \Omega_{\text{LTr}}(\tau_{(i,l,m)}) & \triangleq \tau \\
\Omega_{\text{NTg}}(c!d_\gamma) & \triangleq c!d & \Omega_{\text{Prc}}(c!d_{(p,l,k)}) & \triangleq c!d_{(l,k)} & \Omega_{\text{LTr}}(c!d_\gamma) & \triangleq c!d \\
\Omega_{\text{NTg}}(c?d_\gamma) & \triangleq c?d & \Omega_{\text{Prc}}(c?d_{(p,l,k)}) & \triangleq c?d_{(l,k)} & \Omega_{\text{LTr}}(c?d_\gamma) & \triangleq c?d
\end{align*}
\]

\( \Omega_{\text{NTg}} \) removes all tags from decorated actions, which in turn allows for a straightforward definition of weak actions, \( \xrightarrow{\hat{\alpha}} \) as \( \xrightarrow{\alpha} \) if \( \alpha = \tau \) and \( \xrightarrow{\alpha} \) otherwise. In addition to stripping \( \tau \) action tags, the second filter function, \( \Omega_{\text{Prc}} \), allows only process external actions, filtering out the \( p \) component in this case. The function is partial as it is undefined for all other preLTS actions. This is useful when we do not want to discriminate configurations based on the tracing and monitoring actions. The final filter function, \( \Omega_{\text{LTr}} \), removes all tags but uses them to prohibit silent tracing actions where the two locations in the tag are distinct; this in effect rules out remote trace querying, thereby enforcing localised trace monitoring.

We assume a certain well-formedness criteria on our filter functions, such as that they do not change the form of an action (e.g., an output action remains an output action), and that whenever they map to silent actions, \( \tau \), these are not decorated; the filter functions in Example 4.1 satisfy these criteria. Through this latter requirement we re-obtain the standard silent \( \tau \)-action at the LTS level.

### 4.4 Behavioural Equivalence

The technical development in sections 4.2 and 4.3 allows us to immediately apply weak bisimulation [9] as a coinductive proof technique for equivalence between LTSs obtained for our preLTS and well-formed filter functions. Two (filtered) LTSs, \( C_\Omega \), and \( D_\Omega \), are bisimilar, denoted as \( C_\Omega \approx D_\Omega \), if they match each other’s transitions; we use the weak bisimulation variant, \( \approx \), as this abstracts over internal \( \tau \)-actions which yields a more natural extensional equivalence.

**Example 4.2.** Using the filter functions defined in Example 4.1, we can formally state and prove equivalences (5), (6) and (7) outlined earlier, for a localised clock-set \( \delta \) including locations \( l \) and \( k \):

\[
\begin{align*}
(\delta \triangleright \text{Sys} \parallel M^{\text{orch}})_{\Omega_{\text{NTg}}} & \Leftrightarrow (\delta \triangleright \text{Sys} \parallel M^{\text{orch}})_{\Omega_{\text{NTg}}} & \text{(8)} \\
(\delta \triangleright \text{Sys})_{\Omega_{\text{Prc}}} & \Leftrightarrow (\delta \triangleright \text{Sys} \parallel M^{\text{orch}})_{\Omega_{\text{Prc}}} & \text{(9)} \\
(\delta \triangleright \text{Sys} \parallel M^{\text{mig}})_{\Omega_{\text{NTg}}} & \Leftrightarrow (\delta \triangleright \text{Sys} \parallel M^{\text{mig}})_{\Omega_{\text{LTr}}} & \text{(10)}
\end{align*}
\]

Equivalence (5) formalises the behaviour expected for (5) using an LTS whose actions prohibit distinctions based on action tags; including monitoring location, i.e., \( \Omega_{\text{NTg}} \). Since in (6) we wanted to analyse
how monitors affect process computation, in its corresponding equivalence \( \Omega \) we use an LTS that tags process external actions with location information while prohibiting any actions relating to tracing or monitoring. Finally \( \Omega \) compares \( M_{\text{mig}} \) with itself, subject to a restricted semantics where remote monitoring is prohibited, i.e., \( \Omega_{\text{LT}} \).

5 Instrumentation for Distributed Monitoring

The instrumentation of contracts as distributed monitors is non-trivial and can easily lead to unsound contract monitoring. In this section we illustrate how the instrumentation of contracts, expressed using a simple regular expression-based temporal logic specifying violation traces, can be safely automated according to different monitoring approaches. The syntax of the contract language is:

\[
E ::= (c, \bar{v})@k \mid E.E \mid E^* \mid E + E
\]

Basic events have the form \((c, \bar{v})@k\) indicating that a communication on channel \(c\) with value \(\bar{v}\) occurs at location \(k\). We adopt a semantics allowing for multiple matches, rather than opt only for the shortest match\(^2\) and thus any trace terminating with a communication \(c!\bar{v}\) at location \(k\) is considered to be a violating trace. The other operators are the standard ones used in regular expressions: \(E.F\) corresponds to the traces which can be split into two, with the first matching \(E\), and the second matching \(F\); expression \(E^*\) corresponds to traces which can be split into a number (possibly zero) parts, each of which satisfies \(E\); and \(E + F\) corresponds to the set of traces which match either \(E\) or \(F\).

Notation: \(\sum_{e \in I} E\) corresponds to the generalised choice over finite \(I\), which is equal to \(E^{i_1} + E^{i_2} + \ldots + E^{i_n}\) (where \(I = \{i_1, i_2, \ldots, i_n\}\)).

Despite the apparent simplicity of this expository contract language, we can already express interesting contracts.

Example 5.1. Consider a simplification of the contract outlined in Example 2.7: “The release of a patient’s record must be approved by supervising doctors.” Stated in terms of what leads to a violation, we get: “If a patient’s medical record is released regardless of a doctor’s disapproval, the contract is violated” which can be expressed as the regular expression:

\[
\sum_{p \in \text{Patient}} (\text{req},())@p \cdot \sum_{d \in \text{Doctor}} (\text{withhold},p)@d \cdot (\text{send},p)@h
\]

where \(p, d\) and \(h\) are locations referring to the patient’s, the doctor’s and the hospital domain, channel names \(\text{req}, \text{withhold}\) and \(\text{send}\) denote actions requesting, withholding and sending medical records, and sets \(\text{Patient}\) and \(\text{Doctor}\) range over the finite patients and doctors in the system.

There are different ways in which one may transform a regular expression into an mDP\(\text{t}\) term. For instance, \((c_1, \bar{v}_1)@k_1 . (c_2, \bar{v}_2)@k_2\) may be matched by either one monitoring process, \(M_1\), or by the split monitors, \(M_2\), below:

\[
M_1 \triangleq \text{sync}(k_1).\text{q}(c_1, \bar{x}_1).\text{if } \bar{x}_1 = \bar{v}_1 \text{ then (sync}(k_2).\text{q}(c_2, \bar{x}_2).\text{if } \bar{x}_2 = \bar{v}_2 \text{ then fail)})
\]

\[
M_2 \triangleq (\text{sync}(k_1).\text{q}(c_1, \bar{x}_1).\text{if } \bar{x}_1 = \bar{v}_1 \text{ then } m?) \parallel (m? . \text{sync}(k_2).\text{q}(c_2, \bar{x}_2).\text{if } \bar{x}_2 = \bar{v}_2 \text{ then fail})
\]

\(^2\)In any case, when runtime monitoring one may choose to halt the system on the shortest match.
Distributed System Contract Monitoring

Figure 3: Compiling $E.F$, $E + F$ and $E'$ (respectively) where $C, B$ correspond to comb, bifurc.

We find that the second translation, $M_2$, lends itself better towards illustrating how monitors can be distributed in different ways across locations — for example, in an orchestrated approach, we would place all the monitoring processes in a single location, while in a choreographed approach, we would distribute the processes as required. A sequential approach such as $M_1$, may be more appropriate in an orchestrated approach (since it avoids unnecessary parallelism), but would not be possible to distribute to enable choreographed monitoring without further manipulation.

In this paper, we adopt the maximally parallelised approach, primarily to be able to observe similarities and distinctions between different compilation approaches. In particular we use this translation for a compilation strategy corresponding closely to standard approaches used in hardware compilation of regular expressions [6], producing circuits with two additional wires: an input which is signaled upon to start matching the regular expression, and an output wire which the circuit uses to signal a match with the regular expression. In our case, the wires correspond to channels: basic events $(c, \bar{v})@k$ with start channel $s$ and match channel $f$ would be translated into an expression which waits for input on channel $s$, then outputs on channel $f$ when an instance of $c$ with $\bar{v}$ occurs at location $k$. Our translations also employ two standard monitor organisations for funneling two output signals into one and forking channel communication onto two separate ones; these are expressed below as the macros comb and bifurc:

\[
\text{comb}(f_1, f_2, f) \triangleq *((f_1?\bar{x}.f!\bar{x}) \parallel *(f_2?\bar{x}.f!\bar{x})) \quad \text{bifurc}(s, s_1, s_2) \triangleq *s?\bar{x}.(s_1!\bar{x} \parallel s_2!\bar{x})
\]

We define three compilation strategies, $\psi_O$, $\psi_C$ and $\psi_M$, corresponding respectively to monitoring using orchestration, static choreography and migrating monitoring as discussed in section 2. The compilation procedures use three parameters: two control channels (used to notify when the regular expression is to start being matched, and to notify when it has matched) and the expression to be compiled. For simplicity, all three translations follow a similar pattern shown by the block diagrams in Figure 3, varying only in the location placement of the monitors and synchronisation strategy.

5.1 Orchestration-Based Monitoring Translation

Orchestration places the monitor at some predefined central location, $h$. As stated earlier, the lack of a global clock prevents it from deducing with certainty the order of events happening across different locations. Nevertheless, our translation attempts to mitigate this imprecision for sequence of events occurring at the same location using the following mechanism: when a basic event is matched, the monitoring context, $(k, n)$, at that moment is recorded using getI and passed as arguments on the match signaling channel; this allows subsequent matching to explicitly adjust the monitoring context to these values using seti in cases where the location of subsequent events does not change; in cases where the location changes, this information is redundant and alignment is carried out using the coarser sync command.

\[
\psi'_O(s, f, ((c, \bar{v})@k)) \triangleq *s(x_{loc}, x_{idx}).\text{if } k = x_{loc} \text{ then (seti}(x_{loc}, x_{idx}).\text{trg}(c, \bar{v}, f)) \text{ else (sync}(k).\text{trg}(c, \bar{v}, f))
\]

In either case, a listener triggers a signal with the updated index of the trace on the match channel for every event $c$ with data $\bar{v}$. Specifically, the macro $\text{trg}(c, \bar{v}, f)$ repeatedly reads from channel $c$, outputting
the monitoring-context information on the matching channel \( f \) every time the data traced matches \( \bar{v} \):

\[
\text{trg}(c, \bar{v}, f) \triangleq * ((q(c, \bar{x}), \text{if } \bar{x} = \bar{v} \text{ then getI}(x_{loc}, x_{idx}), f^{!}(x_{loc}, x_{idx})))
\]

The compilation of the regular expression operators matches the compilation schemata of figure 3:

\[
\begin{align*}
\psi'_O(s, f, (E.F)) & \triangleq \text{new } m. (\psi'_O(s, m, E) \parallel \psi'_O(m, f, F)) \\
\psi'_O(s, f, E') & \triangleq \text{new } c', f'. \text{comb}(s, f', c) \parallel \text{bifurc}(c, s', f) \parallel \psi'_O(s', f', E) \\
\psi'_O(s, f, (E + F)) & \triangleq \text{new } s', f_1, f_2. (\text{bifurc}(s, s_1, s_2) \parallel \psi'_O(s_1, f_1, E) \parallel \psi'_O(s_2, f_2, F) \parallel \text{comb}(f_1, f_2, f))
\end{align*}
\]

The combined monitors are located at the predefined central location \( h \), with a dummy initial monitor context continuation parameters \((h, 1)\). The monitor induced for a contract \( E \) is thus:

\[
\psi_O(E) \triangleq h[[ \text{new } s, f. (s'(h, 1) \parallel \psi'_O(s, f, E) \parallel f?'\bar{x}.\text{fail}) ]]^{(h, 1)}
\]

### 5.2 Choreography-Based Monitoring Translation

Instead of instrumenting the whole monitor at a single central location, a choreography-based approach decomposes the monitor into parts, possibly placing them at different locations. Once again, monitors are made up of two kinds of components: (i) the event listeners; and (ii) the choreography control logic made up of comb and bifurc components. The event listeners are located locally, where the event takes place, but are otherwise exactly the same as in the orchestrated approach:

\[
\psi'_C(s, f, ((c, \bar{v})@k)) \triangleq k[[\psi'_O(s, f ((c, \bar{v})@k))]^{(k, 1)}
\]

On the other hand, the choreography control logic can be placed at any location. For instance one may choose to locate them at the node where the next input will be expected, or where the last one occurred. For a particular choice of locations \( l \) and \( h \), choice \( E + F \) is compiled as follows:

\[
\text{new } s_1, s_2, f_1, f_2. (ll [\text{bifurc}(s, s_1, s_2)]^{(l, 1)} \parallel \psi'_C(s_1, f_1, E) \parallel \psi'_C((s_2, f_2), F) \parallel h[\text{comb}(f_1, f_2, f)]^{(h, 1)})
\]

Finally, we add the necessary start signal (from some start location \( k \)) to initiate the monitoring:

\[
\psi_C(E) \triangleq \text{new } s, f. k[[s!(k, 1) \parallel f?'\bar{x}.\text{fail}]]^{(k, 1)} \parallel \psi'_C(s, f, E)
\]

Note that unless all the locations enable the execution of new (monitoring) process at runtime, the contracts must be known at compile-time, which is guaranteed in the simple regular expression logic we are using.

### 5.3 Migrating Monitors Translation

For the migrating monitors technique, we use a simplified translation where the monitors generated are similar to the ones used in orchestration, except that the monitor migrates when required to the relevant location (using the \texttt{go} operator). \( \psi'_M \) is defined identical to \( \psi'_O \) except for basic events:

\[
\psi'_M(s, f, ((c, \bar{v})@k)) \triangleq * ((s?(x_{loc}, x_{idx}), \text{go } k, \text{if } k = x_{loc} \text{ then setI}(x_{loc}, x_{idx}), \text{trg}(c, \bar{v}, f)) \text{ else sync}(k).\text{trg}(c, \bar{v}, f))
\]
Note how migration (thus monitor instrumentation) is delayed and happens only once the start signal on channel \(s\) is received. Initially, the monitor can be chosen to reside anywhere. For a particular location choice \(h\), the migrating monitor approach for a contract \(E\) would be the following:

\[
\psi_M(E) \triangleq h[\text{new } s, f. (s!((h, 1) \parallel \psi_M'(s, f, E) \parallel f?\overline{x}.\text{fail}))]^{(h, 1)}
\]

Despite the resemblances resulting from our simplistic translations, migration improves on an orchestrated approach by avoiding remote tracing. As in the choreographed approach, one can also choose to explicitly run the combining and bifurcation processes at particular locations by adding explicit migration instructions. A better approach would be to nest all the monitors within each other to avoid monitors migrating or installed before they are actually required. For example, monitoring for an expression of the form: \((c_1, v_1)@l. (c_2, v_2)@k. (c_3, v_3)@h\) would be transformed into a monitor of the form:

\[
\text{go } l. (q(c_1, x_1), \text{if } x_1 = v_1 \text{ then go } k. (q(c_2, x_2), \text{if } x_2 = v_2 \text{ then go } h. (q(c_3, x_3), \text{if } x_3 = v_3 \text{ then fail})))
\]

Note that using this approach entails minimal local monitor instrumentation since this happens on a by-need basis: the translation avoids installing any monitor at location \(k\) unless \(c_1!v_1\) happens at \(l\).

Even within this simplistic formal setting, migrating monitors can be seen to be more versatile than a choreographed approach. For instance, if our contract language is extended with variables and a binding construct, \(\exists x. E\), we could express a more dynamic form of contract such as \(\exists x. (c_1, x)@k. (c_2, v)@x\); in such a contract the location of the second event depends on the location communicated in the first event and, more importantly, this location is not known at compile time. Because of this last point, this contract cannot be handled adequately by traditional choreographed approaches which would need to preemptively instrument monitors at every location. However, in a migrating monitor approach, this naturally translates to a single runtime migration.

5.4 The Approaches and Limitations

We have shown how one can formulate different monitoring strategies of the same contract using mDP. The contract language and its compilation procedure have intentionally been kept simple to avoid their complexity from obscuring the underlying monitoring choices. The different approaches mostly differ only in the location of the monitors. The migrating monitor approach also allows for straightforward setting up of new contracts at runtime, including references to locations not known at compile-time. Furthermore, the migrating approach procrastinates from setting up monitors in remote locations until necessary. In contrast, on a choreographed approach, monitors are set up at all locations, even though some of them may never be triggered.

Formalising the compilation of regular expression contracts into mDP also gives us opportunities to formally verifying certain properties. For instance, as a generalisation of (8) we can state and prove that, for arbitrary expression \(E\), different compilation approaches give the same monitoring result. We can state this as:

\[
\delta \triangleright Sys \parallel \psi_O(E)_{\Omega_{NTg}} \approx (\delta \triangleright Sys \parallel \psi_C(E))_{\Omega_{NTg}} \approx (\delta \triangleright Sys \parallel \psi_M(E))_{\Omega_{NTg}}
\]

and prove it by giving witness bisimulations defined by induction on the structure of \(E\). One can prove similar results on the lines of the equivalences given in section 4.4.
6 Conclusions

We have presented a novel process calculus framework in which distributed contract monitoring can be formalised and analysed. We have shown it to be expressive enough to encode various distributed monitoring strategies. To the best of our knowledge, it is unique in that it traces are first class entities rather than meta-constructs. We modularly developed various semantics for this calculus, using transition abstraction techniques that enable selective reasoning about aspects such as locality of communication and distinctions between monitor and process actions.

We are currently working on an implementation in Erlang [2], guided by the design decisions made for our calculus. This should give us insight into practical issues, such as that of addressing trust issues when installing monitors and the avoidance of indirect data exposure due to monitoring. We are also studying mDPf further, addressing issues such as clock boundaries and real-time operators. As the calculus stands, the monitoring component is non-intrusive, in that it reads system events but does not otherwise interact with it. To handle reparations and enforcements upon contract violation, and to be able to express monitor-oriented programming [5] we require potentially intrusive monitoring. We believe that our bisimulation approach can also handle reasoning about monitor intrusiveness.

References

Handling Conflicts in Depth-First Search for LTL Tableau to Debug Compliance Based Languages *

Francois Hantry Mohand-Said Hacid
Université Claude Bernard Lyon 1
LIRIS CNRS UMR 5205
Lyon, France
francois.hantry@liris.cnrs.fr mohand-said.hacid@liris.cnrs.fr

Providing adequate tools to tackle the problem of inconsistent compliance rules is a critical research topic. However, few tools fully analyze conflicts over underpinning logics of a natural language (e.g., temporal logic, deontic logic...). Such early and declarative specifications can be critical for specifying policies and requirements in agile and distributed environments. Thus, formal languages for compliance requirements and their analysis have become critical in many computer science domains (e.g., business process management, service-oriented computing, e-commerce). An ongoing research topic is the analysis of a conflicting set of temporal logic compliance rules. For instance, Table 1.a gives a toy set of compliance rules. It will be used as a running example in the paper. All those rules except the last one originate from an ongoing supply contract. Let us assume that the last one (r3.c) originates from another internal requirement from a supplier. It comes out that this new requirement entails a conflict with rules (r3.a) and rules (r3.b) shown on Table 1.b. This example shows the importance of automatic detection of conflicting subsets of compliance rules. This problem is critical for debugging declarative specifications [16,21], handling conflicting contracts [10], or tackling unrealizable service compositions [20].

There exist several formalisms to deal with time such as LTL, MSO [9], TLTL, MTL [1]. These logics underpin many of modern compliance languages and their associated theories and tools are used to address problems related to verification [15,26,7], service composition [20], graphical design of property patterns [21,16].

We investigate the problem of efficiently extracting temporal logic unsatisfiable cores for debugging compliance rules. Intuitively, an unsatisfiable core is a conflicting subset of rules. We restrict ourselves to LTL for which many results and efficient model checking methods exist. However, the problem of efficiently detecting a small LTL unsatisfiable core is still open [23,6]. Conflict driven methods exist for SAT-solver algorithms. They provide quite efficient extraction of conflicting rules written in propositional logic [28]. SAT-solvers have been extended (e.g., Unbounded Model Checking (UMC) SAT-solvers

---

1 Introduction

Providing adequate tools to tackle the problem of inconsistent compliance rules is a critical research topic. However, few tools fully analyze conflicts over underpinning logics of a natural language (e.g., temporal logic, deontic logic...). Such early and declarative specifications can be critical for specifying policies and requirements in agile and distributed environments. Thus, formal languages for compliance requirements and their analysis have become critical in many computer science domains (e.g., business process management, service-oriented computing, e-commerce). An ongoing research topic is the analysis of a conflicting set of temporal logic compliance rules. For instance, Table 1.a gives a toy set of compliance rules. It will be used as a running example in the paper. All those rules except the last one originate from an ongoing supply contract. Let us assume that the last one (r3.c) originates from another internal requirement from a supplier. It comes out that this new requirement entails a conflict with rules (r3.a) and rules (r3.b) shown on Table 1.b. This example shows the importance of automatic detection of conflicting subsets of compliance rules. This problem is critical for debugging declarative specifications [16,21], handling conflicting contracts [10], or tackling unrealizable service compositions [20].

There exist several formalisms to deal with time such as LTL, MSO [9], TLTL, MTL [1]. These logics underpin many of modern compliance languages and their associated theories and tools are used to address problems related to verification [15,26,7], service composition [20], graphical design of property patterns [21,16].

We investigate the problem of efficiently extracting temporal logic unsatisfiable cores for debugging compliance rules. Intuitively, an unsatisfiable core is a conflicting subset of rules. We restrict ourselves to LTL for which many results and efficient model checking methods exist. However, the problem of efficiently detecting a small LTL unsatisfiable core is still open [23,6]. Conflict driven methods exist for SAT-solver algorithms. They provide quite efficient extraction of conflicting rules written in propositional logic [28]. SAT-solvers have been extended (e.g., Unbounded Model Checking (UMC) SAT-solvers

---

*The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).
to deal with the more expressive LTL. For the case of satisfiability\(^1\) it consists in searching a lasso-shaped model of length \(k \leq 2^{O(|f|)}\) and in reducing to boolean SAT problems for increasing \(k \in [0, 2^{O(|f|)}]\).

One of critical (and basic) points of current boolean SAT-solvers is their ability of pruning ‘bad’ search space. It is based on a smart use of boolean propagation. Analyzing the propagation enables to handle conflict while backtracking and enables to avoid revisiting immediately the same conflict. Learning conflict using conflict clause also avoids revisiting the conflict later. This conflict-driven approach leads easily to the extraction of a core. \(^6\) proposes to extract unsatisfiable cores from the UMC method of \(^{17}\). The authors propose also a ‘Sat Modulo Theory’ like framework applied with symbolic global model checking \(^5\), but the conflict handling is not introduced inside the symbolic global model checking. \(^{23}\) analyzes a very expanded\(^2\) tableau of \(^{14}\) to define unsatisfiable core but again no analysis of conflict is performed. Thus, on the contrary to boolean SAT-solver and extended UMC, neither global model checking, nor On-The-Fly techniques handle conflict. Moreover, in the nineties, resolution \(^{12}\) for temporal logic has been proposed to tackle unfair SCC as minimal ‘temporal conflict’ but to the best of our knowledge current boolean SAT-Solvers(\(^{17},^{13},^{24},^{18}\) ) have not investigated this idea yet, mainly because they are Breadth-First-Search. But, a drawback of resolution is that any conflict is recorded using resolvent, this entails a too large use of memory space in contrast to On-The-Fly tableau, symbolic model checking and UMC. In this paper, we propose a new conflict-driven depth-first-search solver inspired by SAT-based ones, DFS for tableau and resolution for temporal logic. Furthermore, we show how it is possible to extract a small unsatisfiable core.

**Overview of the paper** Section 2 introduces Background. Section 3 describes sound technical details of section 4. Section 4 shows the Solver. Section 5 is devoted to the correctness, completeness, extraction of unsatisfiable cores. We conclude in Section 6.

## 2 Background

**Definition 1** (Syntax of LTL)

Let \(P\) be a non empty finite set of propositional variables, and \(p \in P\). \(A\) and \(B\) two LTL formulas. A temporal logic formula is inductively built by means of the following rules:

\[
\begin{align*}
\text{TRUE} &\mid \text{FALSE} & | \ p & \ | A \land B & \mid A \lor B & \mid \neg A & \mid X(A) \\
AUB & \mid AWB
\end{align*}
\]

Furthermore, \(G(A) = (A)W(\text{FALSE})\) and \(F(A) = (\text{TRUE})U(A)\).

**Definition 2** (Semantic \(^9\)) A linear time structure is an element \(\mathcal{M}\) in \((2^P)^\mathbb{N}\). \(\forall i \in \mathbb{N}, \forall \mathcal{M} \in (2^P)^\mathbb{N}\):

\[
\begin{align*}
(\mathcal{M}, i) \models p & \text{ with } p \in P \text{ iff } p \in \mathcal{M}(i) \\
(\mathcal{M}, i) \models X(A) & \text{ iff } (\mathcal{M}, i + 1) \models A \\
(\mathcal{M}, i) \models AUB & \text{ iff } \exists j \geq i, (\mathcal{M}, j) \models B \text{ and } \forall k, i \leq k < j, (\mathcal{M}, k) \models A \\
(\mathcal{M}, i) \models AWB & \text{ iff } \forall j \geq i, (\mathcal{M}, j) \models A \text{ or } (\exists j \geq i, (\mathcal{M}, j) \models B \text{ and } \forall k, i \leq k < j, (\mathcal{M}, k) \models A)
\end{align*}
\]

\(^1\)No model to check against a LTL formula

\(^2\)The expansion disregards boolean conflict
Intuitively, the formula $X(A)$ stands for ‘at the next time $A$ will hold’, $AUB$ stands for ‘$B$ will hold in the future and from current time until $B$ holds, $A$ must hold’, $AWB$ stands for ‘if $B$ holds in the future then from current time until $B$ holds, $A$ must hold, and if $B$ will never hold, then $A$ must hold forever(weak until)’. $G(A)$ stands for ‘at any time $A$ holds’ and $F(A)$ stands for ‘$A$ will hold in the future’. For instance $\neg iWp$ means that $i$ cannot occur as long as $p$ has not occurred.

In the rest of the paper we will assume w.l.g that any LTL formula solely may contain $\neg$ symbol applied to propositional variable(s). We call such formula Negative Normal Form (NNF).

**Definition 3** (LTL SAT problem) A LTL formula $\phi$ is satisfiable iff there exists a linear model $M$ such that $(M, 0) \models \phi$. Conversely, a LTL formula $\phi$ is unsatisfiable iff there is no linear model $M$ such that $(M, 0) \not\models \phi$.

**Definition 4** (unsatisfiable core) An unsatisfiable core of an unsatisfiable formula $\phi$ is a formula $\phi'$ such that (1) $\phi'$ is the result of some substitution(s) in $\phi$ of some positive subformula(s) by TRUE, (2) $\phi'$ has no subformula of the form $A∪(TRUE)$, $(TRUE)WB$, $A ∨ (TRUE)$, $\land_i True$ or $X(\text{TRUE})$ and (3) $\phi'$ still remains unsatisfiable. Table 1.b shows a small unsatisfiable core of our toy example formula. It is critical to find a small (or ideally a minimal\(^3\)) unsatisfiable core in order to detect the cause of a conflict.

**Theorem 1** (LTL minimal unsatisfiable core decision problem)

Deciding if a LTL formula is a minimal unsatisfiable core is in P-SPACE

**sketch of the proof**: For each positive subformula of $f$, substitute by TRUE and check unsatisfiability. $f$ is a MU iff any substitution leads to a satisfiable formula. There is a linear number of subformulas, and each checking is in P−SPACE.

We furthermore conjecture that the above problem is P-SPACE complete.

---

\(^3\)An unsatisfiable core $\phi$ is minimal iff $\phi$ is its only one unsatisfiable core
on finding small coarse unsatisfiable core. 

**Definition 5** (Closure) Let \( f \) a LTL formula. We note the set of closure variables of \( f \)-\( Cl(f) \)- as the smallest set \( Set \) such that:

- \( f \in Set \)
- If \( \psi = \psi_1 \land \ldots \land \psi_s \in Set \) and \( \psi_j \) is not a conjunction, \( \forall j \ \psi_j \in Set \)
- If \( \psi = \bigvee_1 \ldots \bigvee \psi_r \in Set \) and \( \psi_j \) is not a disjunction, \( \forall j \ \psi_j \in Set \)
- If \( \psi = F/G(\psi') \in Set \), \( \psi' \in Set \) and \( XF/G(\psi') \in Set \)
- If \( \psi = \psi'U/W\psi'' \in Set \), \( \psi'' \) and \( \psi' \land X(\psi) \) are in \( Set \)
- If \( \psi = X(\psi') \in Set \) then \( \psi' \in Set \)

Furthermore the number of closure variables of \( Cl(f) \) is linear in the size of \( f \). 

A traditional mathematical tool to analyze satisfiability is tableau. It is a particular automata of states, whose any state is a subset of \( Cl(f) \). Intuitively, a state is built from a prestate. A prestate is either the starting state containing only the starting formula \( f \) either a state containing only closure formulas derived from a precedent state. The derivation of a formula \( Xh \) at a state is \( h \) at the next prestate. The prestates are intermediary results to build the tableau and do not occur in the tableau except the first one. On Figure 1, the rounded rectangle is a prestate, the others are states. A state is computed by unwinding prestates are intermediary results to build the tableau and do not occur in the tableau except the first one. A state is computed by unwinding the formulas and making a choice for the disjunctive ones. For instance, the occurrence of \( G(\neg i) \) implies the occurrence of \( \neg i \) and \( XG(\neg i) \). In Figure 1, at the goal state of transition 1, the \( p \) is chosen from the disjunction \( \psi \lor (\neg i \land X(\neg iWp)) \) unwound from \( \neg iWp \).

**Definition 6** (state, prestate, \( f \)-tableau) The \( f \)-tableau is a special finite state automata \((St,s_0,R)\) with \( St \) the set of states, \( s_0 \) the initial state and \( R \subset ST^2 \) the set of transitions. The \( f \)-tableau is the ‘minimal’ automata \( A \) such that:

- Any state of \( A \) is a subset of \( Cl(f) \).
- \( s_0 \) is a prestate with \( s_0 = \cup_i \{ f_i \} \) where \( f = \land_i f_i \).
- Let a set \( S \) derived from a prestate \( PS \) st. \( PS \subset S \subset Cl(f) \) and \( \exists p \) a total choice function from \( S \cap (\text{disjunction} \cup \text{Future} \cup \text{Until} \cup W\text{Until}) \) to \( S \). Furthermore, if \( S \) is the smallest set \( Set \) such that

  - \( PS \subset Set \)
  - If \( \psi = \psi_1 \land \ldots \land \psi_s \) and \( \psi_j \) is not a conjunction, \( \forall j \ \psi_j \in Set \)
  - If \( \psi = \psi_1 \lor \ldots \lor \psi_r \) and \( \psi_j \) is not a disjunction, \( \rho(\psi) = \psi_j \in Set \) for some \( j \)
  - If \( \psi = F(\psi'), \rho(\psi) \in \{ \psi'; XF(\psi') \} \cap Set \)
  - If \( \psi = G(\psi'), \psi' \in Set \) and \( XG(\psi') \in Set \)
  - If \( \psi = \psi'U/W\psi'', \rho(\psi) \in \{ \psi''; \psi' \land X(\psi) \} \cap Set \)

then \( S \) is a state of \( A \).

- Let a set \( PS \) containing only all formulas derived from a precedent state \( S \) such that \( PS = \{ \phi, st.X\phi \in S \} \). Then, \( PS \) is a prestate of \( A \).

- Transitions \( R \) of a \( f \)-tableau stand for the collapsing of \( S_1 \rightarrow PS \rightarrow S_2 \) derivation sequences, i.e., collapsed transitions of the form \( S_1 \rightarrow S_2 \).
Theorem 2 ([19], [14]) A LTL formula $f$ is satisfied iff there exists a path of states in the $f$-tableau (finite with no successor at the last state or infinite) starting from the starting prestate and such that any occurrence of Future and Until modal operator in a state of the path fulfills its corresponding promise operand later (in the future) in the path. We call the path: fair path.

In Figure 1, $f$ is a simpler version of our toy example, and there is only unsatisfiable paths (infinite in this case) since each possible path contains a Future $F(i)$ but does not realize the promise operand $i$. An argument is that any infinite path will reach in the future a Strongly Connected Component (SCC) where the path will remain in forever. Then $f$ is unsatisfiable. On-the-fly techniques for satisfiability of temporal logic (eg. [14], [19]) use nested deep-first-search of fair loop or simple deep first search of fair SCC.

Theorem 3 ([25], [19]) There exists a depth-first-search algorithm for computing SCCs of a $f$-tableau, and for deciding their fairness.

In Figure 1, the exploration steps of simple depth-first-search follow the numbered labels on the transitions. An example of a SCC is the set of states as a support for the set of transitions $\{3; 4; 5; 6\}$.  

\[^4\text{disjunctive unwinding are not shown in the tableau since this is an intermediary result}\]
We say that a three-values formula 

\[ \text{fair-valid if for any } x \text{ then } \neg x \in \text{Set}, \neg \neg x, XG(\neg x), G(p \Rightarrow G(\neg x)) \text{ represents a set of literals). Let } S \text{ be a set of LTL formulas. We will use the word ‘literal’ for } x \text{ we note } S' \text{ of the closure algorithm with the initial condition on } \text{Set} = S \text{ instead of } \text{Set} = \{ f \} \text{ and without the last rules } (Xg \text{ derives } g). \text{For any element } g \text{ in } S \text{ we note } x_g \text{ a fresh boolean variable, that we call closure variable. This means presence of } g \text{ in the state. We will use the word ‘l literal’ for } x_g \text{ or } \neg x_g. \text{Finally, we call a clause a disjunction of such literals (also represented by a set of literals). Let } S' \subseteq Cl(f). \text{We say that } S' \text{ is conflicting if there exists } h \text{ and } \neg h \text{ in } S'. \text{Let } V \text{ be a set of closure variables, } L \text{ the literals of } V. \text{Then if } g \text{ and } h \text{ are ‘three-values’ logic formulas then } x_g \in V, g \land h, \text{and } \neg g \text{ are ‘three-values’ formulas. Furthermore assuming } S' \text{ is non-conflicting:}

\begin{align*}
- S' &\models x_g \iff h' \in S' \\
- S' &\models g \land h \iff S' \models g \text{ and } S' \models h \\
- S' &\models \neg g \iff S' \not\models g
\end{align*}

We say that a three-values formula } g \text{ is valid iff for any non-conflicting set of } S', S' \models g. \text{We say that } g \text{ is fair-valid if for any } S' \text{ which is a state from any fair path } S' \models g.

**Definition 8** (Unwinding clauses from a prestate) Let } PS \text{ a prestate and } Presence(PS) = \{ x_h | h \in PS \}. \text{The corresponding Unwound Clause Set } UCS(PS) \text{ is a set containing the unwound clauses and } AUX(PS).
the three values conditions. Set, $\text{AUX}(PS)$ and $\text{UCS}(PS)$ are the smallest sets following the rules:

- $\text{Presence}(PS) \subseteq \text{Set} \cap \text{UCS}(PS)$
- If $x_w = x_{w_1\wedge\ldots\wedge w_j} \in \text{Set}$ and any $x_{w_j}$ is not a conjunction,
  \[ \forall j \text{ the formulas } x_w \Rightarrow x_{w_j} \in \text{UCS} \text{ and } \forall j x_{w_j} \in \text{Set} \]
- If $x_w = x_{w_1\lor\ldots\lor w_j} \in \text{Set}$ and any $x_{w_j}$ is not a disjunction,
  \[ x_w \Rightarrow (x_{w_1} \lor \ldots \lor x_{w_j}) \in \text{UCS} \text{ and } \forall j x_{w_j} \in \text{Set} \]
- If $x_w = x_{w'}U/W x_{w''}$,
  \[ x_w \Rightarrow (x_{w''} \lor (x_{w'} \land X(\psi))) \in \text{UCS} \text{ and } x_{w''} \land x_{w'} \in \text{Set} \]
- $x_h, x_{\neg h} \in \text{Set}$ then $\neg x_h \lor \neg x_{\neg h} \in \text{AUX}$

Furthermore, $\text{AUX}(f)$ (resp. $\text{UCS}(f)$, $\text{Presence}(f)$) is the union of $\text{AUX}(PS)$ for any $PS$ in the $f$-tableau (resp $\text{UCS}(PS)$, $\text{Presence}(PS)$). The unwound formulas $\text{UCS}(PS) \setminus \text{Presence}(PS)$ are fair valid formulas (see proof section 5) and of the form $x_h \Rightarrow \text{dis} j x_h$ where $\text{dis} j x_h$ is the classical disjunctive unwinding of closure formulas $^{[14, 19]}$.

The formula $f$ of Figure 3 provides the clause $\text{UCS}(f)$:

\[
\begin{align*}
    x_{F(i)} & \Rightarrow x_i \lor x_{XF(i)} & \text{XG}(f) & \Rightarrow \text{XG}(f) \\
    x_{G-c} & \Rightarrow x_{-c} & x_{(-i)Wp} & \Rightarrow x_p \lor x_{-i} \lor X((\neg j)Wp) \\
    x_{-i} \lor X((-i)Wp) & \Rightarrow x_{-i} & x_{-i} \lor X((-i)Wp) & \Rightarrow \text{XG}(((-i)Wp) \\
    x_{G(p \Rightarrow G(-i))} & \Rightarrow \text{XG}(p \Rightarrow G(-i)) & x_G(p \Rightarrow G(-i)) & \Rightarrow \text{XG}(p \Rightarrow G(-i)) \\
    x_{G(-i)} & \Rightarrow x_{-p} \lor XG(-i) & x_{G(-i)} & \Rightarrow \text{XG}(G(-i)) \\
    x_{G(-i)} & \Rightarrow x_{-i} & \forall v \in \text{CLST} f \Rightarrow \neg x_v \lor \neg x_{\neg v}
\end{align*}
\]

**Proposition 1** An instance $\text{IS}$ of the boolean SAT problem $\text{UCS}(PS) \cup \text{AUX}(PS)$ provides a state $S$ from $PS$ and reciprocally.

Since many instances correspond to a state in the tableau, and since several states may be redundant regarding LTL satisfiability problem, we introduce Fair Prime Implicant.

**Definition 9** (Fair Prime Implicant) Let $\text{IS}$ as above, a Fair Prime Implicant $\text{IS}.FPI$ of $\text{IS}$ is a maximal $^5$ switching from some assigned $x_h$ at $\text{IS}$ to $\neg x_h$ such that $h$ is not a promise operand and $\text{IS}.FPI \models \text{UCS}(PS) \cup \text{AUX}(PS)$. At a given $\text{IS}.FPI$ it corresponds only one state $\text{FPI}$ in the $f$-tableau.

**Theorem 4** (Fair prime implicant version of Depth-First-Search) A formula $f$ in LTL is satisfiable iff there exists a fair path solely with FPIs as states.$^{(proof is omitted)}$.

For instance, the FPI technique enables in our depth first search to ignore the goal state of the transition number 4 at Figure 1.

To solve the boolean SAT-problem current solvers use unit rule propagation $^{[8]}$.

**Definition 10** (Unit rule propagation)

- Each instantiated literal must be propagated$^6$ over any non yet satisfied clause containing the opposite one. This opposite literal is then temporally erased from the clause.
- If a clause becomes unit literal $l$ because of unit rule propagation(s), then $l$ is assigned

This propagation is critical for conflict analysis. In the following we show how to handle unit rule propagations to support conflict analyses.

$^5$Intuitively the switching simulates the removal of closure element in corresponding state

$^6$A Weakest version and optimized one of current solvers requires only propagation along watched literals $^{[22]}$
Handling Conflicts in DFS for LTL tableau

3.2 Implication Graph to support Conflict analyses

The Implication Graph is an extension of propositional SAT-solvers’ one to LTL-tableau. The intuition is to record the occurrences of elements of the closure at a given state that entail another one. An Implication Graph is a bicolor graph \((\text{Nodes}, \text{T}_{\text{red}}, \text{T}_{\text{black}})\) where \(\text{T}_{\text{red}}\) and \(\text{T}_{\text{black}}\) are subsets of \(\text{Nodes}\). Figure 3 shows a part of the Implication Graph adapted from the \(f\)-tableau of Figure 1. Intuitively, the Implication Graph is a concatenation of several \(IS\)'s implication graphs denoted \(ISIG\). The red part \(\text{T}_{\text{red}}\) is used for conflict analysis of the depth-first-search stack \(S\) and it is a DAG, and the black part \(\text{T}_{\text{black}}\) records some past red edges and corresponds to the conflict analysis of the SCC-search using stack \(S'\) and allows loop for inductive reasoning.

**Nodes’ feature** Intuitively, a Node \(N\) stands for an assigned literal at a given state. On Figure 3, the rounded corners rectangles are Nodes. Each node is inside a big rectangle standing for state. More precisely, a Node corresponds to the ongoing prestate, to an ongoing \(IS\) while it is found and to a chosen extracted \(IS.FPI\) in this case. On Figure 3, the three states are the one which support the transitions \(\{2;3;4\}\) on Figure 1. Furthermore a Node can be either chosen or required. On Figure 3, a chosen node is doubly surrounded. The level of a chosen Node \(N\) is its chronological order of choice in the whole \(f\)-tableau. On figure 3 numbers are levels of chosen nodes. The level of any node \(N\) is the maximum level of the chosen nodes which involve \(N\) i.e which are ancestors of \(N\) in \(\text{T}_{\text{red}}\). The level of a set of nodes is the maximum level its nodes. A required node is either without antecedent but with level 0 either gets an antecedent in \(\text{T}_{\text{red}}\).

**Transitions’ feature** If a Node \(N_i\) which corresponding literal \(l\) comes from a clause \(C = \lor j_j \lor l\) which has become unit, then the red edges \((\text{N}_i, \text{N}_j)\) are in \(\text{T}_{\text{red}}\) just after this unit propagation. Let’s focus on the above state. For instance, \((x_p, x_{G(-i)})\) and \((x_p = G(-i)), x_{G(-i)}\) are red edges because of the unit rule from the clause \(x_p => G(-i) \Rightarrow (x_p \lor x_{G(-i)})\). Furthermore, the derivation from a state to a next state is also recorded using red edges such that the occurrence of \(x_{\text{Fl}} \in IS.FPI\) entails the occurrence of \(x_{\text{F}}\) at the next prestate. For instance on Figure 3, the above FPI derives to the middle one, thus there exists a red edge in the graph from \(x_{XF(i)}\) at above state to \(x_{Fi}\) at the next one.

\(^7\)for understanding but w.l.g, the below state is not a FPI
\(^8\)0 if no ancestor nodes are chosen
Furthermore, while a FPI is revisited, then the current IS implication graph IS.IG has to be connected to the first one IG_{old} which visited the same FPI. The algorithm creates black transitions from any nodes N(¬x_{opm}) (resp. x_{opm} ∈ IS.FPI) at IS.IG to the same literal one of IG_{old}. This connection is called ‘bind’ function. For instance, for corresponding derivation on Figure 1 for IS.IG at the goal state of transitions \{3;5;6\} the IS.IG and IG_{old} are the same. For simplicity and w.l.g.\footnote{The particular computation of fixpoint remains the same while superimposing in this simple case} they have been superimposed at Figure 3. In this case, the transitions of ‘bind’ have been omitted w.l.g, and solely the bottom-up edges from source state to the goal state of transitions \{3;5;6\} are shown (e.g., x_{XF(i)} at the below state to x_{XF(i)} at the same state for transition 3). Finally, given a T_{red} and a chosen Node N still red, \text{flip}(Nodes,T_{red},T_{black},N) = (Nodes\cup\{¬N(red)\}),T_{red}\setminus\{(N_1,N_2)\in T_{red}\mid level(N_2) ≥ level(N)\},T_{black} is the flipped Implication Graph regarding N with ¬N(red) a fresh node.

4 Solver

Our depth-first search temporal conflict driven solver is a combination of depth first search of fair SCC in tableau\footnote{The main components of the algorithm are shown in a recursive form for convenience} and of boolean SAT-solver. Thus, our solver uses unit rule propagation method, boolean conflict handling\footnote{a boolean conflict detection occurs while a clause is falsified by current partial assignment}. It also uses a new temporal conflict driven method inspired by resolution for temporal logic\cite{12}.

**Basic Solver**\footnote{10} Algorithm 1 shows the main method of the algorithm called Solver. At each new prestate, the solver populates by clauses by unwinding the prestate according to Definition 8. Otherwise, unit rules and boolean conflict detection\footnote{11} are launched. A Backtrack (Algorithm 2) is triggered in case of a conflict, otherwise if it is possible, a choice of literal following a heuristic is done. Once an IS is found and a FPI extracted, then a SCC-search-forward (Algorithm 3) function is called. Otherwise the Solver is recursively called.

**Algorithm 1: Solver**

\[
\text{if not unwound then} \\
\quad \text{Unwind;} \\
\quad \text{Unit-rule ; bool-conflict-detection ;} \\
\text{if conflict then} \\
\quad \text{Backtrack;}
\]

\[
\text{if IS found then} \\
\quad \text{SCC-search-forward;} \\
\text{else} \\
\quad \text{make a choice of literal;} \\
\quad \text{Solver ;}
\]

**Propositional Conflict Handling while backtracking** A Propositional Conflict Handling is triggered when a clause is falsified (or equivalently when a literal and its opposite occurs). Similarly to SAT-solvers’ one, the Propositional Conflict Handling starts from a set of conflicting nodes Nodes\_SC and corresponding literals \(C\) which falsifies the clause \(¬C\) and analyzes which nodes have involved those conflicting literals using (Nodes(red),T_{red}). Let \(α(C)\) be the subDAG of (Nodes(red),T_{red}) which stands for ancestors of Nodes\_SC. Let \(\alpha(C)(\text{conflict – level})\) be the subDAG of \(α(C)\) with nodes of ‘conflicting’ level of Nodes\_SC i.e. conflict – level and \(N(\text{conflict – level})\) the chosen node of level conflict – level. Let Limit(C) = \(\{N(\text{conflict – level})\} \cup (\text{Parent}_{T_{red}}[α(C)(\text{conflict – level})] \setminus \{N(\text{conflict – level})\})\) ∩ Level(conflict – level – 1, α(C)) \] where Level(m, α(C)) means the subDAG of α(C) with node level
Handling Conflicts in DFS for LTL tableau

at most $m$. We call limit conflict clause $\neg\text{Limit}(C)$. The last conflicting chosen node $N(\text{conflict}−\text{level})$ is then switched if the corresponding flipped partial assignment has not been visited yet (node.flip=1). In this first case, similarly to boolean SAT-solving, the function ‘Conflict-require’ adds red edges to $(\text{Nodes}(\text{red}), T_{\text{red}})$: the red transitions with a source node in $\text{Limit}(C) \setminus \{N(\text{conflict}−\text{level})\}$ to the goal node $\sim N(\text{conflict}−\text{choice})$. However, differently from boolean SAT-solver, since the algorithm records informations in black part $(\text{Nodes}(\text{black}), T_{\text{black}})$ in the second case (flip=2), the same transitions but in color black are added. Furthermore, $\sim N(\text{conflict}−\text{level})$ is now required and not chosen. Those red or black edges are to ensure we can compute the reason of the requirement of $\sim N(\text{conflict}−\text{level})$. Finally, if the conflict level is 0 then the algorithm terminates by unsatisfiable.

**Algorithm 2: Backtrack**

- Compute Conflict-level;
- if $\text{Conflict-level}=0$ then print (‘unsat’), break;
- State-Conflict-Clause-learning;
- Tableau.IG.erase(Conflict-level);
- stack-s.erase(C-level);
- stack-s’.erase(C-level);
- Conflict-require;
- SCC-search-backward;

On Figure 3, the backtrack is done from the conflicting (see. TC-Analysis) nodes $x_{G}(-i)$ and $x_{F}(i)$ at the middle state. Following the red part, the last involved and chosen node is $x_{p}$ at above state. While backtracking bad states and corresponding nodes are erased (above state at Figure 3). On the contrary to propositional SAT-solver, the algorithm has to record the cause of these states to be bad (to avoid revisiting them) using a conflict clause per state.\(^{12}\) These learned clauses must not be forgotten. On figure 3, the yellow literals are conflicting literals at middle state but the clause $\neg x_{G}(-i) \lor \neg x_{F}(i)$ has already been learned. At above state, the pink literals provide the learned clause $\neg x_{p} \lor \neg x_{p \rightarrow G}(-i) \lor \neg x_{f}(i)$. Finally, a SCC-search backward is launched. Algorithm 2 summarizes the above ideas. We refer to \([27]\) for more details about backtracking in boolean SAT-solvers.

**Algorithm 3: SCC-search-forward**

if $\text{FPI}$ is new then
- $\text{Nb}(\text{FPI}):=i:=i+1;
- \text{Lp}(\text{FPI}):=\text{Lv}(\text{FPI}):=\text{Nb}(\text{FPI});
- \text{stack-s.push}(\text{FPI});
- \text{stack-s’.push}(\text{FPI});
- \text{parent}=\text{FPI}; \text{prestate}=\text{FPI}.next();
- \text{Solver};$

else
- case $\text{state} \in \text{stack}−\text{S}$
  - $\text{Lp}(\text{Parent}):=\min(\text{Lp}(\text{Parent}), \text{Nb}(\text{FPI}))$;
- case $\text{FPI} \notin \text{stack}−\text{S}$
  - $\text{Nb}(\text{parent}) > \text{Nb}(\text{FPI})$
    - $\text{Lp}(\text{Parent}) := \min(\text{Lp}(\text{parent}), \text{Nb}(\text{FPI}))$
    - $\text{parent.unr-prom} = \text{parent.unr-prom} \cap \text{FPI}_{\text{old}}$.unr-prom$
    - \text{bind}($IS, IG, IG_{\text{old}}$);
    - SCC-search-backward;

**SCC-Search-Forward** The SCC-search-forward shown Algorithm\(^{3}\) is similar to the ‘forward’ part of the computation of strongly connected components and uses depth first search numbers (Lp,Nb,Lv). If the FPI is new, then new numbers are computed and if it is possible, the next prestate (and corresponding

\(^{12}\)We ask that the conflicting clause forbids corresponding red FPI of state
prestate Nodes and transitions from derivations) are created from the (red) nodes from literals $x_{Xh} \in IS.FPI$, otherwise the problem is satisfiable. Moreover, if the already visited $FPI_{old}$ is still in $Stack - \Delta'$ or in $Stack - S$, a computation on Tarjan’s numbers is also launched. The unrealizable promises are also computed. Furthermore, in any revisiting case, a rollback is launched while calling SCC-search-backward (see Algorithm 4).

**SCC-Search-Backward** First the algorithm adds black copies of red edges in $IS.IG$. Then, starting with the current choosen node $N$ of current level, the Algorithm 4 simply finds the last non-flipped chosen node. If it is in $IS$ then, it calls $flip(IG,N)$ and Solver. Otherwise change color red to black at the ‘next’ edges from $parent.IG$ to $IG$. Then a SCC test over Tarjan numbers is launched from the parent state, and if a SCC is found a SCC-handling is called, otherwise, update of unrealizable promise is done. If a promise is unrealizable then SCC-handling calls a Temporal Conflict Analysis (TC-Analysis), otherwise the problem is satisfiable.

**Algorithm 4**: SCC-search-backward

```plaintext
N=node(level)
IS.IG.edges.black-copies
if N.flip=2 ∧ N ∈ IS then
  level=level-1; SCC-Search-Backward
if N.flip = 1 ∧ N ∈ IS then
  flip(IG,N)
  Solver;
if N /∈ IS then
  red-to-black-parent.IG-IS.IG-derivation
  FPI=parent; pop stack-s
  parent= head stack-s
  if Lp(FPI)=Nb(FPI)=Lv(FPI) then
    SCC-handling∗
  else
    Lp(parent)=min(Lp(parent),Lp(FPI))
    Lv(parent)=min(Lv(parent),Lv(FPI))
    parent.unr-prom= parent.unr-prom ∩ FPI.unr-prom
    SCC-search-backward

SCC-handling∗ ::
if unrealizable promise = ∅ then
  print ‘satisfiable’; break;
else
  TC-Analysis;
```

**TC-Analysis of unfair SCC** In the SCC, the algorithm 5 chooses an unfair promise and computes a backward fixpoint from some nodes $N(\neg x_{op}(Promise))$ for any SCC states along the recorded black implication graph. Precisely, except the root state of the SCC, any state of the SCC gets a corresponding black ‘IG’ from $stack - S$ which is the $IS.IG.edges.black-copies$ one while SCC-backward-search. For the root state SCC, only the nodes $N(x_{Xh})$ and $N(\neg x_{op}(Promise))$ get some black transitions.

The fixpoint computation starts from those nodes at $IS.IG.edges.black-copies$ or particular nodes at the root. Once the inflationary backward fixpoint using $T_{black}$ is terminated, then at each state in SCC, the algorithm picks up a corresponding $IG$. For any state, the ‘prestate(s)’ Nodes $Nodes_{prestate}$ in the $IG$ which are also in the fixpoint are declared conflicting with the unfair promise and the algorithm learns and must not forget the conflict clause. Then, the method erases all the states of this SCC. It finally triggers a classical Backtracking at the nodes of the Root from the conflicting prestate(s). Nodes of the

---

13 Please see for more details about Tarjan’s numbers
14 Since the root has been revisited, it gets at least one black $IG$
root. At Figure 3, the unfair promise is $F(i)$, and the fixpoint computation is shown by double arrow. In this SCC, the yellow and green Nodes are involved in the temporal conflict, and the yellow are the causes of this conflict, i.e., $x_{F(i)}$ and $x_{G(-i)}$ are conflicting. Thus, $\neg x_{G(-i)} \lor \neg x_{F(i)}$ is learned forever.

## 5 Correctness, Completeness, and Extraction of a small unsatisfiable core

**Lemma 1** Any clause from $AUX(f)$ or $UCS(f) \setminus Presence(f)$ are fair valid.

**proof:** Any fair state is non conflicting then AUX is fair valid. By construction, any fair state satisfies any clause from $UCS(PS) \setminus Presence(PS)$.

**Lemma 2** Let $f$ be a LTL formula. Assume the Algorithm has computed a conflict analysis from the conflicting literals $C$. Let $ICl(f, C) = AUX(f) \cup UCS(f) \cup Learn(f, C)$ with $Learn(f, C)$ containing any learned clause occurring in the algorithm strictly before $C$ and any limit conflict clause occurring at any conflict handling strictly before $C$. Assume that $Learn(f, C)$ are fair valid clauses. Let $Cf$ be the conjunction of conflicting literals used to learn a resulting clause of the conflict analysis $\neg Cf$. Then $\neg Cf$ is fair valid.

**sketch of the proof:** Thanks to lemma

**Algorithm 5:** Temporal Conflict Analysis

```
INI: Vector= ¬ops(Promise) ∩ SCC;
while ∃e ∈ Vector ∧ e not marked do
  mark e; v = e.blake - parents:
  for l ∈ v ∧ l not marked do Vector.push(l)
end
∀state ∈ SCC pick up a State.IG;
do learn( Vector ∩ state.IG.prestate, promise);
erase SCC;
Backtrack;
```

Let $ICl(f, C) = AUX(f) \cup UCS(f) \cup Learn(f, C)$ containing any learned clause occurring in the algorithm strictly before $C$ and any limit conflict clause occurring at any conflict handling strictly before $C$. Assume that $Learn(f, C)$ are fair valid clauses. Let $Cf$ be the conjunction of conflicting literals used to learn a resulting clause of the conflict analysis $\neg Cf$. Then $\neg Cf$ is fair valid.

**sketch of the proof:** Thanks to lemma

1. Either the conflict $C$ is boolean. Let $\mathcal{A}'(C) = Level(\mathcal{A}(C), conflict - level)$ and $Limit(C)$ as above. Then each node in $\mathcal{A}'(C) \setminus \{n(conflict − level)\}$ is required and it originates either from state to prestate derivation, either from a clause $Cl \in ICl(f, C) \setminus Presence(f)$ which has become unit at a given state. Since $ICl(f, C) \setminus Presence(f)$ are assumed fair valid then $S' \models Cl$ for such a clause $Cl$ and for any state $S'$ in $pS$. Then the proof from $\mathcal{A}'(C)$ by unit rule of the conflict $C$ of our algorithm implies that there exists a state $S'_{conflict}$ in $pS$ such that $S'_{conflict}$ contains $\square$. This implies a contradiction since $p$ is assumed fair and then no state of $p$ should be conflicting.

2. Either the conflict $C$ is temporal. Assume $S'$ any state of the unfair SCC. For any state of the SCC, let $Pre(S')$ be the set of ‘black’ prestates from a chosen IS.IG. from $S'$. Let $k \in \mathbb{N}$. Imagine virtually the exploration of any non conflicting prefix path $p'$ of length $k$ in the induced tableau $T(Pre)$ by also considering $ICl(f, C) \setminus Presence(f)$. It consists of the building of a boolean SAT-problem based on the following observations:

   - Since any bad old SCC is not reachable by not forgetting any conflict clause of bad state/bad SCC, then there exists a k-depth-first navigation over the Prime Implicants from $T(Pre)$ but

---

15 the limit conflict clause is $\neg Limit(C)$; we consider it even if the limit conflict clause is not learned by the solver

16 suffixes of $p$ from $S$
remaining in the unfair SCC and following the Prime implicant depth-first-search of the f-tableau.

- if \((s_i = Pre, s_{i_1}, ..., s_{i_k})\) is a Prime Implicant path in \(T(Pre)\) from the precedent k-depth-first navigation, then from the algorithm, at any transition \((s_j, s_{j+1})\), it corresponds (several) state(s) Implication graphs \(IG_{i_j}\) for \(s_{i_j}\), and \(IG_{i_{j+1}}\) for \(s_{i_{j+1}}\) corresponding at any (re)visit of the states.

- There exists a k-depth-first navigation of full \(T(Pre)\) following the Prime implicant depth-first-search of the f-tableau, such that if \((s'_i = Pre, s'_{i_1}, ..., s'_{i_k})\) is a path of states in \(T(Pre)\), then a corresponding Prime implicant path \((s_i = Pre, s_{i_1}, ..., s_{i_k})\) is one from k-depth-first navigation of the Prime implicant f-tableau.

- Let \(CL_0(Pre), UC_k(f) \backslash Presence_k(f) \cup AUX_k(f), Learn_k(f, C)\) be the timestamped variables and corresponding clauses. Let \(Next_k = \{x_j(X(f)) \Rightarrow x_{j+1}(f) | 0 \leq j < k\}\) be the clauses encoding the state to next prestate derivations. Then there exists a DLL-exploration \(E\) of the propositional problem \(CL_0(Pre) \cup UC_k(f) \backslash Presence_k(f) \cup AUX_k(f) \cup Learn_k(f, C) \cup Next_k\) following the k-depth-first navigation of the full \(T(Pre)\) but disregarding conflicts which do not occur in the DFS of the SCC in the f-tableau.

- Let \(E'\) be the modified exploration of \(E\) but by pruning any part of the exploration which contradict any timestamped limit conflict clause.

- Let \(E_{Promise}\) be the modified exploration of \(E'\) for the boolean SAT problem \(CL_0(Pre), UC_k(f) \backslash Presence_k(f) \cup AUX_k(f), Learn_k(f, C), Next_k, x_k(Promise)\) without learning. Furthermore it non chronologically backtracks. It also considers only conflicts of the form \(\{x_k(op(Promise)) ; \neg x_k(op(Promise))\}\). Then clearly \(E_{Promise}\) does not find any solution because the promise is not fulfilled and particularly at step \(k\), i.e. the boolean problem is unsatisfiable.

It is now feasible to show that:

(a) The last conflict \(C_{last}\) of \(E_{Promise}\) is at level 0. This means that ancestor literals in \(s'_k(C_{last})\) with no parent gets a level 0, i.e. they correspond to clauses \(Core_k\) of length one in \(CL_0(Pre), UC_k(f) \backslash Presence_k(f) \cup AUX_k(f), Learn_k(f, C), Next_k, x_k(Promise)\) since there is no learning in \(E_{Promise}\). Furthermore \(x_k((op(Promise)) \in Core_k\). Finally, \(Core_k \cup UC_k(f) \backslash Presence_k(f) \cup AUX_k(f), Learn_k(f, C), Next_k, x_k(Promise)\) is an unsatisfiable core.

(b) Let \(C_{f'_k} = Core_k \backslash \{x_k(op(Promise))\} \cup learn_k(f, C)\), then \(C_{f'_k} \subset CL_0(Pre)\). Let \(C_{f_k}\) be the non timestamped literals. Then if \(S = C_{f_k}\) and since \(S\) is a state of a fair path, then if \(p_{S,k}\) is the suffix path from \(S\) but truncated of length \(k\), \(p_{S,k} = Core_k \backslash \{x_k(op(Promise))\} \cup UC_k(f) \backslash Presence_k(f) \cup AUX_k(f), Learn_k(f, C), Next_k, \neg x_k(op(Promise))\).

(c) \(C_{f_k} = \{e \in Pre\} \{N(e_0 = e) \Rightarrow N(e_1) \Rightarrow ... \Rightarrow N(e_k = \neg x_k(op(Promise))\}, \text{with } N(e_i) \Rightarrow N(e_{i+1}) \in T_{black} \text{ and } N(\neg x_{op(Promise)}) \in SCC\}.

It is then straightforward that if \(S = x_{Promise} \wedge k \in N C_{f_k}\) then there is a contradiction since \(p_S\) will never realize the operand promise \(x_{op(Promise)}\). Furthermore, \(\wedge k \in N C_{f_k}\) is computed as the set of \(Pre\) contained in the backward fixpoint over \(T_{black}\) computing ancestors of any \(\neg x_{op(Promise)}\) for all states of the SCC.

**Theorem 5** The learned clauses and Limit conflict clauses[^17] are fair valid.

**Sketch of the proof:** By chronological induction on the learned clause and limit conflict clause per conflict. First, assume that conflict \(C\) is the first, thus the \(Learn(f, C) = \emptyset\) at lemma\[^2\]. Thus \(-Cf\) and

[^17]: in case of propositional conflict
\(-\text{Limit}_C\) is fair valid. Assume now that \(\text{Learn}(f, C)\) are valid. Thanks to lemma \(^2\), it follows that \(\neg C f\) and \(\neg \text{Limit}(C)\) are fair valid.

**Theorem 6** The algorithm terminates, is correct and complete

(\textit{sketch of the proof}: As long as a state is not known to be bad or in a Bad SCC, then it is recorded\(^{18}\) to avoid infinite loop. As soon as it is sure that it is a bad state or in a bad SCC, then a clause which will never be forgotten and standing for the bad state is learned. Thus, our algorithm is similar to a depth-first-search of SCC in a LTL tableau \(^{19}\). However, as soon as there is a conflict, the algorithm prunes part of the tableau which is sure to lead to a failing state/SCC by, sound learning and backtracking using implication dependencies of conflict.

**Theorem 7** (Extraction of coarse small unsatisfiable core)

If \(f = \land_i f_i\) then \(\land_i\{f_i|x_{f_i} \in \mathcal{A}(C_{\text{last}})\}\) is a coarse small unsatisfiable core.

(\textit{sketch of the proof}: If the algorithm terminates with ‘unsat’, the last conflict \(C_{\text{last}}\) is at level 0. This means that ancestor nodes in \(\mathcal{A}(C_{\text{last}})\) with no parent gets a level 0, i.e. they correspond to some clause in \(\text{presence}(f) : x_{f_i}\) where \(f = \land_i f_i\) or eventually to some learned clause of the form \(\neg x_h\). But since \(IC_l(f, C_{\text{last}}) \setminus \text{Presence}(f)\) are fair valid, then \(\land_i\{f_i|x_{f_i} \in \mathcal{A}(C_{\text{last}})\}\) is a coarse small unsatisfiable core.

### 6 Conclusion

In order to detect which compliance rules are conflicting, we have provided a conflict-driven Tableau depth-first-search for LTL. We have shown how it can be used to extract a small unsatisfiable core. Our method is theoretically \(\text{EXPTIME}\) and \(\text{EXPSPACE}\), but although deciding a MU is in \(P – \text{SPACE}\) no \(P – \text{SPACE}\) method have been proposed to extract cores yet. Our method does not suffer from cumbersome timestamped variables, handling of incrementation, searching upper bound for UMC. Implementation is ongoing work. Three enhancements of the method would be to study a QBF-encoding of our method and analyzes if the learning we propose is easy for QBF solvers to learn. Other ways could be to use symbolic DFS \(^3\) or alternating Büchi automata. Detecting conflicts in rules is critical for human interactive contract management systems. Moreover, our method pinpoints temporal issues in any automatic tool which is sensitive to the consistency of many evolving heterogeneous policies such as regulatory laws, internal business rules, security or privacy. The extension of our method to deontic modality \(^4, 10\) used in contracts appears straightforward, and we are also focusing on this issue.

### References


\(^{18}\) by a hash table for instance


From Contracts in Structured English to $\mathcal{CL}$ Specifications

Seyed Morteza Montazeri, Nivir Kanti Singha Roy
Department of Computer Science and Engineering
University of Gothenburg, Sweden
gusmonse@student.gu.se, guskanthni@student.gu.se

Gerardo Schneider
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
Department of Informatics
University of Oslo, Norway
gersch@chalmers.se

In this paper we present a framework to analyze conflicts of contracts written in structured English. A contract that has manually been rewritten in a structured English is automatically translated into a formal language using the Grammatical Framework (GF). In particular we use the contract language $\mathcal{CL}$ as a target formal language for this translation. In our framework $\mathcal{CL}$ specifications could then be input into the tool CLAN to detect the presence of conflicts (whether there are contradictory obligations, permissions, and prohibitions. We also use GF to get a version in (restricted) English of $\mathcal{CL}$ formulae. We discuss the implementation of such a framework.

1 Introduction

Natural language (e.g., English) descriptions and prescriptions abound on documents used in different phases of the software development process, including informal specifications, requirements, and contracts at different levels (methods/functions, objects, components, services, etc.). There is no doubt of the usefulness of having such descriptions and prescriptions in natural language (NL), as most of the intended users of the corresponding documents would not have problems to understand them. However, it is well known that NL is ambiguous and imprecise in many cases due to context sensitivity, underspecified terminology, or simply bad use of the language.

At the other extreme, we have formal methods with a myriad of different formal languages (logics) with complex syntax and semantics. Those languages are indeed extremely useful as they are precise, unambiguous, and in many cases tools are available as to provide the possibility of (semi-) automatic analysis. However, in many cases they require high expertise not only at the syntactic level in order to use the language to specify system properties or requirements, but also to interpret the results of the tools. For instance, though the use of model checkers has been advertised as a “push-button technology” not requiring user expertise, the reality is that one still needs to write the properties on a logic and interpret the counter-examples which are also usually given as a big formula representing the trace leading to the problematic case.

On an ideal world, software engineers (and the mortal non-technical users) would only need to deal with natural language descriptions, push a button and get a result telling them whether for instance the given contract (specifications, set of requirements)\(^1\) is consistent and conflict-free. The current state-of-practice however is far from that ideal world. Though the state of the art on NL processing has advanced

\(^1\)In the rest of the paper we will use the term “contract” to refer to contracts at different levels (including legal contracts), software specifications, requirements, etc.
quite a lot in recent years, we still have to depend on the use of formal languages and techniques to analyze such contracts. A relatively new trend is then to restrict the use of NL in order to get something that “looks like a NL” but it has a better structure, and if possible avoid ambiguity. We call such constrained languages restricted or controlled NL.

Our aim in this paper is to advance towards finding a suitable solution to the problems and challenges just mentioned, in particular the possibility of writing contracts in NL, but being able to analyze them with tools, automating the process as much of possible. In particular, we show that it is possible to relate the formal language for contracts $\mathcal{CL}$ and a restricted NL by using the Grammatical Framework (GF). In this way we are able to take contracts written in NL, manually obtain a restricted NL version of such contract, use GF to automatically obtain a $\mathcal{CL}$ formula that could then be analyzed using CLAN. Note that the above process should be in both directions, as we might need the translation from $\mathcal{CL}$ into NL since the result of CLAN in case a conflict is detected is given as a (eventually huge) $\mathcal{CL}$ formula representing the counter-example.

The paper is organized as follows. In next section we recall the necessary technical background the rest of the paper is based on, including $\mathcal{CL}$ and GF. In section 3 we present our framework in general terms, and we provide some details on the implementation. Before concluding in the last section we present a case study in section 4.

2 Background

2.1 The Contract Language $\mathcal{CL}$

$\mathcal{CL}$ is a logic based on combination of deontic, dynamic and temporal logics, designed to specify and reason about legal and electronic (software) contracts. With the help of $\mathcal{CL}$ it is possible to represent the deontic notions of obligation, permission and prohibition, as well as the penalties applied in case of not respecting the obligations and prohibitions. In what follows we recall the syntax of $\mathcal{CL}$, and we give a brief intuitive explanation of its notations and terminology, following [10]. A contract in $\mathcal{CL}$ may be obtained by using the syntax shown in Fig. 1.

A contract clause in $\mathcal{CL}$ is usually defined by a formula $C$, which can be either an obligation ($C_O$), a permission ($C_P$) or a prohibition ($C_F$) clause, a conjunction of two clauses or a clause preceded by the dynamic logic square brackets. $O$, $P$ and $F$ are deontic modalities, the obligation to perform an action $\alpha$ is written as $O_C(\alpha)$, illustrating the primary obligation to perform $\alpha$, and if $\alpha$ is not performed then the reparation contract $C$ is enacted. The above represents in fact a CTD (Contrary-to-Duty) as it specifies what is to be done if the primary obligation is not fulfilled. The prohibition to perform $\alpha$ is represented by the formula $F_C(\alpha)$, which not only specifies what is forbidden but also what is to be done in case the prohibition is violated (the contract $C$); this is called CTP (Contrary-to-Prohibition). Both CTDs and CTPs are then useful to represent normal (expected) behavior as well as the alternative (exceptional) behavior. $P(\alpha)$ represents the permission of performing a given action $\alpha$. In the description of the
syntax, we have also represented what are the allowed actions (α and β in Fig. 1). It should be noticed that the usage of the Kleene star (the * operator) which is used to model repetition of actions, is not allowed inside the above described deontic modalities, though they can be used in dynamic logic style-conditions. Indeed, actions β may be used inside the dynamic logic modality (the bracket [·]) representing a condition in which the contract C must be executed if action β is performed. The binary constructors &., ., and + represent concurrency, sequence and choice in basic actions (e.g. “buy”, “sell”) respectively. Compound actions are formed from basic actions by using the above operators. Conjunction of clauses can be expressed using the ∧ operator; the exclusive choice operator (⊕) can only be used in a restricted manner. ⊤ and ⊥ are the trivially satisfied and violating contract respectively. 0 and 1 are two special actions that represent the impossible action and the skip action (matching any action) respectively.

The following example is an excerpt from part of a contract between an Internet provider and a client, where the provider gives access to the Internet to the client: “The Client shall not supply false information to the Clients Relations Department of the provider”, and “if the client does provide false information, the provider may suspend the service” [8]. If we consider the action fi (representing that “client supplies false information to Client Relations Department”), and s (representing that “provider suspends service”), in CL the above would be written as F_{P(s)}(fi).

One of the usefulness of CL is that there are tools available to detect whether a given CL formula contains conflicts. There are four main kinds of conflicts in normative systems, and contracts in particular. The first one is when there is an obligation and the prohibition of performing the same action, in which case, independently of what the performed action is, will lead to a violation of the contract. The second conflict type happens when there is a permission and a prohibition to do the same action, which might lead to a contradicting situation. The other two cases are when there is an obligation to perform mutually exclusive actions, and the permission and the obligation to perform mutually exclusive actions. CLAN is a tool that automatically determines whether there are conflicts on CL formulae, giving a counter-example in case a conflict is detected [3].

2.2 Grammatical Framework

The Grammatical Framework (GF) is a framework to define and manipulate grammars [11]. Historically, GF was first implemented in the project “Multilingual Document Authoring” at Xerox Research Center Europe in Grenoble in 1998 [11]. As explained in [4] the main idea of this project was to build an editor which helps a user to write a document in a language which is unfamiliar, as Italian, while at the same time seeing how it develops in a familiar language such as English. The development of GF continued over time and evolved into a functional programming language, with multiple application domains [11]. GF has a central data structure called abstract syntax, based on type theory where it is possible to define special purpose grammars. It also has a concrete syntax part where it is possible to specify how the formulas defined in the abstract syntax can be translated into NL or a formal notation. One notable use of GF useful for our purposes, is the possibility to relate NL and formal languages in both directions: it is possible to go from a sentence written in NL to a term in the formal language, and vice-versa. GF has a module system consisting basically of two main modules, one to define the abstract syntax and the other the concrete syntax.

Abstract syntax. Abstract syntax is a type-theoretical part of GF where logical calculi as well as mathematical theories may be defined simply by using type signatures [4]. Let us consider the simple “Hello

2 Some of the reasons behind the CL syntax have been motivated by a desire to avoid deontic paradoxes [6]. See [9] for a more detailed explanation of the design decisions behind CL.
world” example. We start by defining the types of meaning (categories) Greeting and Recipient:
cat Greeting ; Recipient.
We then define Hello as a function for building syntactic trees. The type of Hello has Greeting as its
value type and Recipient as its argument type: fun Hello: Recipient -> Greeting;.

Concrete syntax. Once an abstract syntax is constructed, we can build a concrete syntax by using linearization
rules, which basically allow us to generate the language. For instance, the English linearization
rules for the function Hello is: lin Hello recip = {s = ‘hello’ ++ recip.s};
The above defines the linearization of Hello in terms of the linearization of its argument, represented
by the variable recip. The terminal ‘‘hello’’ is concatenated with recip.s using the concatenation
operator ++, which combines terminals. The most important thing to consider in a linearization rule is
to define a string as a function of the variable it depends on.

The example above shows that the fun rules should be defined in the abstract syntax modules and lin
rules in concrete syntax modules. In the abstract syntax is where we need to define powerful type theory
to express dependencies among parts of texts, and in the concrete syntax we need to define language-
dependent parameter systems and the grammatical structures.

From the practical point of view, the concrete syntax description above could be saved on a file (let
say example.gf), which could be uploaded by making import example.gf. By executing the command line Parse "hello world" we generate the abstract syntax Hello World. It should be noticed
that Hello is the function defined in the abstract syntax above (prefixed with fun), and Recipient is the
argument type of the function (i.e., World is of type Recipient). In the concrete syntax, Recipient is
a record recip containing a string s which in this case will take the value "world".

To summarize, the two main functionalities in GF useful for our purpose are linearization (translation
from abstract to concrete syntax) and parsing (translation from concrete to abstract syntax).

3 Our framework

In this section we present our framework, AnaCon, in general terms, and we present a summary of the
linearization and parsing process of CL into GF.

3.1 The framework in general terms

AnaCon takes as input a text file containing the description of a contract consisting of 3 parts: 1) A
Dictionary listing the actions being used in the contract together with a textual description; 2) The contract
itself written in restricted English; 3) A list of contradictory actions. (Fig. 2 shows the input file
containing a simple contract.)

Our framework is summarized in Fig 3 where arrows represent the flow of information highlighting
how it works. Essentially, it consists of a parser, the Grammatical Framework, the conflict analysis tool
CLAN, and some scripts used to connect these different parts. Overall, the typical workflow of AnaCon
is as follows:

1. The user writes a contract (specification, set of requirements, etc) in NL (“plain” English), which
   is then manually translated into restricted English. This is a modeling task and it is done manually.
   It does not require any technical skill from the user, only to get access to the list of “allowed”
   English words to be used in the restricted version of the language. For instance, a sentence origi-
   nally written as “The ground crew is obliged to open the check-in desk” would be translated into
It is mandatory to open the check-in desk”, where “open the check-in desk” is an action name representing the real action.

2. The version of the contract written in restricted English is then passed to AnaCon script as an argument so the analysis starts (in what follows we call the input file containing the given contract, Contract.txt).

3. Cont_ParserScriptGen (a Java program) generates a script file based on the content of Contract.txt. The script file Cont.Parser then projects the content of the file to testGrammarCl parser.

4. At this stage testGrammarCl conducts syntax analysis based on the structure defined for the system. This parser is based on Labelled BNF grammar and generated from BNF Converter [5].

5. The Java program Comparison connects with testGrammarCl in order to obtain actions defined in the contract and then compare these actions against the ones defined in the Dictionary part of Contract.txt. Other analysis such as comparison between actions defined in Contradiction and the ones in Dictionary, duplication of actions in Dictionary and empty string assertion, are conducted at this level.

6. After successful parsing, the Cont_GF_Cl script file is generated with the contract and necessary information to start the translation process in GF from Restricted English to CL.

7. The version of the contract written in restricted English is then represented in the abstract syntax part of GF.

8. The abstract syntax obtained above is translated into concrete syntax (CL) which is then stored in the text file Result_Cl.txt.

9. The concrete syntax (CL formula) in textual form is transformed into XML by using Cl2XML (implemented but not integrated).

10. The XML version of the CL output of GF is fed into CLAN for analyzing whether the contract has a conflict.

11. If the output of CLAN is ‘NO’, then the answer is immediately given to the user. If the answer is ‘YES’ then the counter-example will be given by CLAN (a big CL formula containing the conflicting subclauses as well as the trace leading to such a state).

12. The formula obtained from CLAN is then linearized into a restricted English using GF. The Cl_GF
script will take the content of Result_Cl.txt and pass it to Cl_GF_ContScript (a Java program), which generates the Cl_GF_Cont script to start the translation. The result in restricted English is given in a file named Result_Eng.txt.

13. The user must then find in the original contract where the counter-example arises. This last step is currently done manually, by simply searching in the text the keywords given in the counter-example in restricted English. (We discuss in the last section our future work on how to automate part of this process.)

Except for the steps above where we explicitly mention that it is manual, the rest of the process is completely automatic. So far, we have only fully implemented steps 2-8 and the translation back from $\mathcal{CL}$ to restricted English (as used in step 12).

Besides the above, we provide the possibility of generating a restricted English version of a $\mathcal{CL}$ formula, by executing AnaCon with a special flag (AnaCon -cl <input_file.txt>).

In what follows we will present some details of the implementation concerning the use of GF only.

### 3.2 Linearization and Parsing

In what follows we present the abstract and concrete syntax of $\mathcal{CL}$ and NL, in GF. At first, we present all the categories and functions for handling different $\mathcal{CL}$ clauses and actions in the abstract syntax part,
and then we concentrate on the concrete syntax part showing their representation in natural language. Due to lack of space we only show here some parts of the process.

The abstract syntax module as a central structure contains the basis for representation and formalization of $CL$ syntax. The linearization of the different used functions and structures into $CL$ symbolic syntax and natural language is done by using two concrete syntax modules. In the first one we write the exact $CL$ syntax, and in the latter we express the corresponding restricted natural language.

We define the following categories, based on the BNF of $CL$.

-- Abstract module (Cl.gf module)

```plaintext
cat Act;KleeneStarAct;KleeneCompAct;ClSO;ClSF;ClSP;Clause;Clauses;ClauseP;
ClauseO; ClauseP;And;Or;Dot;Cross;CompAct;Star;Not;[Clause]{2};[CompAct]{2};
[ClauseO]{2}; [ClauseP]{2};
```

We define similar categories in the concrete modules ClEng.gf and ClSym.gf, using lincat instead of cat prefixing them. The main difference is that in the linearization type definitions we need to state that for instance Clause and Act are records containing a field of type string $s$.

**Obligations, Permissions and Prohibitions.** Obligations, permissions and prohibitions have the same structure in the Cl.gf module. In the rest of this section we will only show the abstract and concrete syntax for obligations, those for permissions and prohibitions being similar.

-- Abstract module (Cl.gf module)

```plaintext
fun Obl : ClSO -> CompAct -> Clauses;
OClause : ClauseO -> Clause;;
Clo : ClSO -> CompAct -> ClauseO;
```

There are some differences among these clauses (or rather group of clauses, as there are similar ones for permissions and prohibitions) which is worth mentioning. As it is clearly shown CompAct is the argument type used among the four groups which represent both basic and compound actions respectively. This will provide the possibility to be able to express Obligation over both basic and compound actions (actions will be explained later on in this section). ClauseO follows the BNF of the syntactic definition of $CL$ obligations and allows to express the obligation that together with the action (verb) form the actual clause. The other difference is that generally a Clause itself can consist of different structures and thus it can be either constructed from ClauseO, ClauseP and ClauseF from basic or compound actions. They are defined in this way to avoid mixing certain operators only permitted for some of the deontic notions. It facilitates to make the conjunction and choice of certain kind of clauses, but not the direct linearization and parsing of each structure. For that we specify Clauses as a start category for parsing and linearization so that each structure can be linearized and parsed directly. The linearization of the above in concrete syntax is as follows:

-- Concrete module (ClEng.gf module)

```plaintext
lin Clo clo compact = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
OClause clo = {s = clo.s};
Obl clo compact = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
```

Basically, Clo and Obl clauses are expressed in NL using restricted words as “It is mandatory to” (clo.s) as terminals (quoted words in GF are terminals) so that together with actions they formulate the clauses in NL.

As explained before, we provide another concrete syntax module called ClSym.gf to provide the user with the possibility of writing with specific $CL$ syntax, such as operators, parentheses, brackets,
etc. The converse is also true, when writing any specific clause in NL, the framework (with the help of this module) would be able to provide \( \mathcal{CL} \) formulas in the intended format.

\[\text{-- Concrete module (ClSym.gf module)}\]
\[
\begin{align*}
\text{lin Clo clo compact} & = \{s = \text{"(" ++ clo.s ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{"")}\}; \\
\text{OClause clo} & = \{s = \text{clo.s}\}; \\
\text{Obl clo compact} & = \{s = \text{"(" ++ clo.s ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{"")}\};
\end{align*}
\]

The structure of the above module is very similar to the ClEng.gf module, the only difference being that \( \text{clo.s} \) represent specific characters such as "O" instead of words or sentences.

\[\text{-- Concrete module (ClSym.gf module)}\]
\[
\text{lin O = \{s = \"O\"\};}
\]

**Contrary-to-Duties (CTDs) and Contrary-to-Prohibitions (CTPs).** CTDs and CTPs are both related to obligation and prohibition clauses respectively, and are expressed as the following functions:

\[\text{-- Abstract module (Cl.gf module)}\]
\[
\begin{align*}
\text{fun CTDc : CompAct} & \rightarrow \text{Clause} \rightarrow \text{Clauses}; \\
\text{CTDcc} & : \text{CompAct} \rightarrow \text{Clause} \rightarrow \text{Clause};
\end{align*}
\]

CTD and CTP clauses are functions taking an action (which is to be obliged, or prohibited) and a clause representing what is to be done in case the obligation or the prohibition is not fulfilled. We specify again these operators over simple and compound actions. We only present now the concrete modules for CTDs, the ones for CTPs being similar.

\[\text{-- Concrete module (ClEng.gf module)}\]
\[
\begin{align*}
\text{lin CTDc compact clause} & = \{s = \text{"(" ++ \"It is mandatory to" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" if not" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" then" ++ clause.s ++ "}\text{"")}\}; \\
\text{CTDcc compact clause} & = \{s = \text{"(" ++ \"It is mandatory to" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" if not" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" then" ++ clause.s ++ "}\text{"")}\};
\end{align*}
\]

Now, we show how to express the logical syntax in the other concrete module:

\[\text{-- Concrete module (ClSym.gf module)}\]
\[
\begin{align*}
\text{lin CTDc compact clause} & = \{s = \text{"(" ++ \"0" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" -" ++ clause.s ++ "}\text{"")}\}; \\
\text{CTDcc compact clause} & = \{s = \text{"(" ++ \"0" ++ "}\text{"(" ++ compact.s ++ "}\text{"") ++ "}\text{" -" ++ clause.s ++ "}\text{"")}\};
\end{align*}
\]

The only important thing to notice here is the ‘‘\text{" -”}’’ character to express the reparation, meaning that the clause after this symbol is the reparation clause which has to be considered in case of a violation of the primary obligation.

**Conjunction of clauses** (\( C \land C \)) Other operators, like conjunction and exclusive or, need also to be represented in the abstract and concrete syntax. We only show here how to represent the conjunction.

\[\text{-- Abstract module (Cl.gf module)}\]
\[
\begin{align*}
\text{fun Conj_np : [Clause]} & \rightarrow \text{Clauses}; \\
\text{Conj_np2} & : \text{[Clause]} \rightarrow \text{Clause};
\end{align*}
\]
As it is clearly defined in the above representation the structure used for conjunction of clauses consists of list of clauses which may be any kind of clause such as obligation, prohibition, etc. In this it is possible to define conjunction of many clauses. The two concrete modules are as follows:

```
-- Concrete module(ClEng.gf module)
lin Conj_np xs = {s = "(" ++ xs.s ++ "! Conjunction_np ++ ")"};
Conj_np2 xs = {s = "(" ++ xs.s ++ "! Conjunction_np ++ ")"};
-- Concrete module(ClSym.gf module)
lin Conj_np xs = {s = "(" ++ xs.s ++ "! Conjunction_np ++ ")"};
Conj_np2 xs = {s = "(" ++ xs.s ++ "! Conjunction_np ++ ")"};
```

The structure used in above to show iteration conjunction of clauses, corresponds to the way it was defined in CL.

**Test Operator.** The test operator where in CL is used to express conditional obligations, permissions and prohibitions. The test operator may be applied to simple or compound actions including the Kleene star. We should thus add a function to each different application of the test operator; we will only present the abstract and concrete syntax of the application of the Kleene star to simple and compound actions.

```
-- abstract module(Cl.gf module)
fun TestOpc,TestOpcStar : KleeneCompAct -> Clause -> Clauses;
      TestOpcc,TestOpccStar : KleeneCompAct -> Clause -> Clause;

The concrete modules are as follows:

-- concrete module(ClEng.gf module)
fun TestOpc kleenecompact clause = {s = "(" ++ "If" ++ "(" ++ kleenecompact.s ++ ")" ++ "then" ++ clause.s ++ ")"};
fun TestOpcStar kleenecompact clause = {s = "(" ++ "(" ++ "Always" ++ ")" | "(" ++ "After" ++ ")" | "(" ++ "When" ++ ")" | "(" ++ "Before" ++ ")" | "(" ++ kleenecompact.s ++ ")" ++ "then" ++ clause.s ++ ")" ++ ")"};
fun TestOpcc kleenecompact clause = {s = "(" ++ "If" ++ "(" ++ kleenecompact.s ++ ")" ++ "then" ++ clause.s ++ ")"};
fun TestOpccStar kleenecompact clause = {s = "(" ++ "(" ++ "Always" ++ ")" | "(" ++ "After" ++ ")" | "(" ++ "When" ++ ")" | "(" ++ "Before" ++ ")" | "(" ++ kleenecompact.s ++ ")" ++ "then" ++ clause.s ++ ")" ++ ")"};
-- concrete module(ClSym.gf module)
fun TestOpc kleenecompact clause = {s = "(" ++ "[" ++ kleenecompact.s ++ "]" ++ clause.s ++ ")"};
fun TestOpcStar kleenecompact clause = {s = "(" ++ "[" ++ "Always" ++ "]" | "(" ++ "After" ++ ")" | "(" ++ "When" ++ ")" | "(" ++ "Before" ++ ")" | "(" ++ kleenecompact.s ++ "]" ++ "then" ++ clause.s ++ "]" ++ ")"};
fun TestOpcc kleenecompact clause = {s = "(" ++ "[" ++ kleenecompact.s ++ "]" ++ clause.s ++ "]"};
fun TestOpccStar kleenecompact clause = {s = "(" ++ "[" ++ kleenecompact.s ++ "]" ++ clause.s ++ "]" ++ ")"};
```


**Actions.** Defining basic, compound and Kleene star actions in GF is not difficult, however it should be noted that since actions in our case are generally verbs that could be considered as a specific vocabulary part (domain lexicon), it is more efficient to use module extension [11]. This in effect separates the grammar part (Cl.gf module) from a more specific vocabulary part (Action module). In other words, the developer will be provided with a modular system giving more flexibility to modify the modules, and thus increasing maintainability. The Action module extends the Cl module. In such a module we will define all the involved simple actions (e.g., “Pay”), other actions only affected by the Kleene star (e.g., “CloseCheckIn”), as well as those operators over actions. In what follows we only show the conjunction (sequence of actions, choice, etc are defined similarly).

---

**Action module (abstract syntax)**

fun Pay,Buy : Act;

**Cl module (abstract syntax)**

fun CompActSI : Act -> CompAct;

CompActa : CompAct -> And -> CompAct -> CompAct;

The linearization of the functions specified in the Action module above described is easy to specify:

---

**ActionEng module (concrete syntax)**

lin Pay = {s = "pay a fine"};

Buy = {s = "buy a car"};

CloseCheckIn = {s = "closeTheCheckIn"};

CorrectDetail = {s = "checkThatThePassportDetailMatch"};

The structure of compound actions shows how the operators’ name has been used as an argument types to build the functions. However, what we need to focus on in the translation of compound actions into NL is to know how each operator should be interpreted. As a consequence we end up with the following concrete syntax where it is possible to express all the operators:

---

**Cl module (abstract syntax)**

fun CompActSI : Act -> CompAct;

CompActa : CompAct -> And -> CompAct -> CompAct;

**ClEng module (concrete syntax)**

lin CompActSI acti = {s = acti.s};

CompActa compact and compact1 = {s = compact.s ++ and.s ++ compact1.s};

User defined operations such as the above fall under specific logical symbols which are defined below:

---

**ClSym module (concrete syntax)**

lin CompActa CompAct and CompAct1 = {s = compact.s ++ and.s ++ compact1.s};

CompActSI acti = {s = acti.s};

In this manner, the representation of and.s (similarly for or.s, dot.s, and not.s) are reduced to "&", "+", "." and "!" which are logical operators as used in CLAN to manipulate CL formulae.

The Kleene star is actually a compound action as shown below with the difference that it can only be used between test operator:

---

**Cl module (abstract syntax)**

fun KleeneActSI : Act -> KleeneCompAct;

KleeneActa : KleeneCompAct -> And -> KleeneCompAct -> KleeneCompAct;
1. The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves.

2. The airline is obliged to reply to the passenger manifest request made by the ground crew when opening the desk with the passenger manifest.

3. After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass.

4. If the luggage weighs more than the limit, the crew is obliged to collect payment for the extra weight and issue the boarding pass.

5. The ground crew is prohibited from issuing any boarding cards without inspecting that the details are correct beforehand.

6. The ground crew is prohibited from issuing any boarding cards before opening the check-in desk.

7. The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before.

8. After closing check-in, the crew must send the luggage information to the airline.

9. Once the check-in desk is closed, the ground crew is prohibited from issuing any boarding pass or from reopening the check-in desk.

10. If any of the above obligations and prohibitions are violated a fine is to be paid.

Figure 4: Case study

In what follows we show the corresponding concrete syntax enabling the translation to natural and symbolic languages:

-- C1Eng module (concrete syntax)
lin KleeneActSI acti = {s = acti.s};
    KleeneActa kleenecompact and kleenecompact1 = {s = kleenecompact.s ++
        and.s ++ kleenecompact1.s};
-- C1Sym module (concrete syntax)
lin KleeneActSI acti = {s = acti.s};
    KleeneActa kleenecompact and kleenecompact1 = {s = kleenecompact.s
        ++ and.s ++ kleenecompact1.s};

4 Case Study

In this section we apply our framework to a case study taken from [2] of a contract concerning the check-in process of an airline company. The full description is given in Fig. 4. We provide the detailed translation into restricted English, and the corresponding C.L formula for all the clauses. Note that clause 10 is “distributed” among the others as it represent a penalty in case the other clauses are not satisfied.

1. The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves (Fig. 4 first clause).
2. The airline is obliged to reply to the passenger manifest request made by the ground crew when opening the desk with the passenger manifest

[Restricted English]: ( (When) ( If (opening_the_desk_with_the_passenger_manifest) then (It is mandatory to (reply_to_the_passenger_manifest_request) if not (reply_to_the_passenger_manifest_request) then (It is mandatory to (pay_a_fine)))) )

[Program output]: ( [ 1 * ] ( [ opening_the_desk_with_the_passenger_manifest ] ( O ( reply_to_the_passenger_manifest_request ) - ( O ( pay_a_fine ) ) ) )

3. After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass (Fig. 4 third clause).

[Restricted English]: (((After) (If (open_the_check_in_desk) then (It is mandatory to (check_that_the_passport_details_match_what_is_written_on_the_ticket_and_check_the_luggage_is_within_the_weight_limits) if not (check_that_the_passport_details_match_what_is_written_on_the_ticket_and_check_the_luggage_is_within_the_weight_limits) then (It is mandatory to (pay)))))) and (If (check_that_the_passport_details_match_what_is_written_on_the_ticket_and_check_the_luggage_is_within_the_weight_limits) then (It is mandatory to (issue_the_boarding_pass) if not (issue_the_boarding_pass) then (It is mandatory to (pay_a_fine)))) )

[Program output]: (( [ 1 * ] ( (open_the_check_in_desk) ] ( O (check_that_the_passport_details_match_what_is_written_on_the_ticket & check_the_luggage_is_within_the_weight_limits) _ ( O (pay_a_fine ))) ) ∧ ( (check_that_the_passport_details_match_what_is_written_on_the_ticket & check_the_luggage_is_within_the_weight_limits) ] ( O (issue_the_boarding_pass) _ ( O (pay_a_fine )) ) )

4. If the luggage weighs more than the limit, the crew is obliged to collect payment for the extra weight and issue the boarding pass (Fig. 4 seventh clause).

[Restricted English]: ( If (the_luggage_weighs_more_than_the_limit) then (It is mandatory to (collect_payment_for_the_extra_weight_and_issue_the_boarding_pass) if not (collect_payment_for_the_extra_weight_and_issue_the_boarding_pass) then (It is mandatory to (pay_a_fine))))

[Program output]:( [ (the_luggage_weighs_more_than_the_limit) ] ( O (collect_payment_for_the_extra_weight & issue_the_boarding_pass) _ ( O (pay_a_fine ) ) ) )
5. The ground crew is prohibited from issuing any boarding cards without inspecting that the details are correct beforehand. (Fig. 4 ninth clause).

[Restricted English]: (It is mandatory to (inspect that the details are correct beforehand) if not (inspect that the details are correct beforehand) then (It is prohibited to (issue any boarding cards) if (issue any boarding cards) then (It is mandatory to (pay a fine))))

[Program output]: (O (inspect that the details are correct beforehand) _ (F (issue the boarding pass) _ (O (pay a fine) )))

6. The ground crew is prohibited from issuing any boarding cards before opening the check-in desk (Fig. 4 tenth clause).

[Restricted English]: ( (Before) (If (open the check in desk) then (It is prohibited to (issue any boarding cards) if (issue any boarding cards) then (It is mandatory to (pay a fine))) ) )

[Program output]: ( [ 1 * ] ( [ (open the check in desk) ] ( F (issue the boarding pass) _ ( O (pay a fine) ) ) ) )

7. The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before

[Restricted English]: ( (Before) (If (20 minutes the flight is due to leave and not before) then (It is mandatory to (close the check in desk) if not (close the check in desk) then (It is mandatory to (pay a fine))))

[Program output]: ( [ 1 * ] ( [ (20 minutes the flight is due to leave and not before) ] ( O (close the check in desk) _ (O (pay a fine) )) ) )

8. After closing check-in, the crew must send the luggage information to the airline

[Restricted English]: ( (After) (If (close the check in desk) then (It is mandatory to (send the luggage information to airline) if not (send the luggage information to airline) then (It is mandatory to (pay a fine))))

[Program output]: ( [ 1 * ] ( [ (close the check in desk) ] ( O (send the luggage information to airline) _ ( O (pay a fine) ) ) ) )

9. Once the check-in desk is closed, the ground crew is prohibited from issuing any boarding pass or from reopening the check-in desk

[Restricted English]: ( (Always) (If (close the check in desk) then (It is prohibited to (issue any boarding pass or open the check in desk) if (issue any boarding pass or open the check in desk) then (It is mandatory to (pay a fine))))

[Program output]: ( [ 1 * ] ( [ (close the check in desk) ] ( O (issue any boarding pass or open the check in desk) _ ( O (pay a fine) ) ) ) )
From Contracts in Structured English to $\mathcal{CL}$ Specifications

Program output:

\[
[\text{1 * } ( (\text{close\_the\_check\_in\_desk} ) (\text{issue\_the\_boarding\_pass} + \text{open\_the\_check\_in\_desk} ))) (\text{ O ( pay\_a\_fine ) ) })]
\]

The case above has already been analyzed using CLAN before and a conflict has been detected as reported in [3]. So, in this sense we do not report any new result here. We have used the same example as our intention is to validate our approach on a familiar case study when all the steps in our framework be implemented (we are currently working on a full implementation of the framework).

5 Related Work

GF has been used on a variety of application domains. We will only focus here on the one reported in [4], since it is closely related to our research. In such paper Hähnle et al. describes how to get a NL version of a specifications written in OCL (Object Constraint Language). The paper focused on helping to solve problems related to authoring well-formed formal specifications, maintaining them, mapping different levels of formality and synchronizing them. The solution outlined in the paper illustrates the feasibility of connecting specification languages at different levels, in particular OCL and NL. The authors have implemented different concepts of OCL such as classes, objects, attributes, operations and queries in GF. Our work is similar to [4] with the difference that $\mathcal{CL}$ is a more abstract and general logic allowing to specify contracts in a general sense (as mentioned in the introduction $\mathcal{CL}$ may formalize legal contracts, software specifications, contracts in SOA, or even be used to represent requirements). Besides we are not interested only on “language translation” but rather in the use of the formal language to further perform verification (in our case conflict analysis) which is then integrated within our framework by connecting GF’s output into CLAN. In what concerns the technical difficulties related to the implementation in GF we do not have enough knowledge of the work done in [4] as to make a more careful comparison.

From the perspective of relating a contract language and natural language, it is worth mentioning the work by Pace and Rosner [7], where it is presented an end-user system which is specifically designed to process the domain of computer oriented contracts. The translation is not based on GF but on a completely different technology. They use controlled natural language (CNL) to specify contracts, and define a similar logic to $\mathcal{CL}$, which is embedded into Haskell in order to manipulate the contracts. So, the comparison is not straightforward as the aim of their work and ours diverges.

6 Conclusions

We have presented in this paper an encoding of the contract language $\mathcal{CL}$ into GF, and back. We have integrated the above into a framework that allows to analyze $\mathcal{CL}$ formulae for conflicts, and eventually give a counter-example in restricted English helping the user to find it in the original specification in natural language. As a proof-of-concept we have applied it to a case study which has already been used for conflict detection. The framework as presented here does not automate the whole process, though we are working on those parts as described below.

We would like to emphasize what was said in the introduction concerning the scope of our approach. $\mathcal{CL}$ is a formal language to specify contracts in a broad sense, and as such one should not think that our work limits to the analysis of contracts in that language. As an abstract logic, $\mathcal{CL}$ can be used to describe and prescribe “contracts” (including specifications) in SOA, component-based development systems, e-business, requirement engineering, etc. We believe the approach is useful in practice, the
only potential bottleneck being CLAN since the current version is not optimized as to obtain small non-redundant automata (the tool is very much a specialized explicit model checker, where high number of transitions are generated due to the occurrence of concurrent actions).

One practical way to reduce the size of the automaton created by CLAN is to try to define as many mutually exclusive actions as possible. Note that some of the actions in our contract example are obviously mutually exclusive (e.g., ‘open the check in desk’ and ‘close the check in desk’), while others are mutually exclusive in the “formal” sense, that is we know that they cannot occur at the same time (for instance, ‘issue a fine’ and ‘issue the boarding pass’). We are currently working on more fundamental ways to improve the performance of CLAN by reducing the size of the automaton while building it.

A challenging future work concerns the use of Passage Retrieval tools (as for instance the one presented in [1]) to help to find the counter-example in the original English contract by using the information in restricted English (obtained from CLAN and translated into English by our framework). This will avoid to manually go through big part of the English text, increasing efficiency and precision. Another interesting line of research is to study how to combine our approach to the one presented in [2]. We believe we could then improve our analysis capability, by using specialized tools for some purposes (as we have done here, using CLAN), and the embedded language technology for others.

References


A Software Tool for Legal Drafting*

Daniel Gorín  
Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Argentina  
dgorin@dc.uba.ar

Sergio Mera  
Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Argentina  
smera@dc.uba.ar

Fernando Schapachnik  
Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Argentina  
fschapachnik@dc.uba.ar

Although many attempts at automated aids for legal drafting have been made, they were based on the construction of a new tool, completely from scratch. This is at least curious, considering that a strong parallelism can be established between a normative document and a software specification: both describe what an entity should or should not do, can or cannot do. In this article we compare normative documents and software specifications to find out their similarities and differences. The comparison shows that there are distinctive particularities, but they are restricted to a very specific subclass of normative propositions. The rest, we postulate, can be dealt with software tools. For such an enterprise the FORMALEX tool set was devised: an LTL-based language and companion tools that utilize model checking to find out normative incoherences in regulations, contracts and other legal documents. A feature-rich case study is analyzed with the presented tools.

1 Introduction

Although many attempts at automated aids for legal drafting have been made (e.g., [17, 24, 15, 25, 18, 10]), they were based on the construction of a new tool, completely from scratch. This is at least curious, considering that a strong parallelism can be established between a normative document and a software specification: both describe what an entity should or should not do, can or cannot do. In the case of normative documents, it is a legal entity. In the case of software specifications, it is a piece of software. Is a software specification so different from a normative document? If it is not, why do not reuse the already existing machinery that can successfully analyze specifications?

In this article we compare normative documents and software specifications to find out their similarities and differences. The comparison shows that there are distinctive particularities, but they are restricted to a very specific subclass of normative propositions. The rest, we postulate, can be dealt with temporal model checkers. For such an enterprise the FORMALEX tool set was devised: an LTL-based language, FL, and companion tools that utilize model checking to find out normative incoherences in legal documents.

FL is based on the following key-concepts:

- It provides a background theory to state matters about the real world, such as event precedence (e.g., sunrise before dawn), uniqueness (each person is born only once), etc., that would otherwise had to be accommodated into unnatural deontic rules. Said background

---

*Partially supported by PICT 2007 532, UBACyT 20020090200116, 2002009020084 and 20020100200103.
theory is translated into an automaton that determines the class of models over which the rest of the language predicates. Section 4.1 provides the details.

- Deontic rules are translated into LTL, but the input language, that is, the way the original deontic rules are written, is preserved, so this information can be used to perform a set of analysis, at a meta-logical level. See Section 4.3 for details.

- The combination of an automata-based formalism plus a logic that can refer to it is very powerful, and the software community knows it. FL takes advantage of that to easily express properties that are generally difficult to pose in other formalisms, or lead to computational complexity problems. These features can be seen in Sections 3 and 5.

A comparison between specifications and legal documents is presented in Section 2. Section 3 highlights FL, our LTL-based language, by describing its use to express otherwise hard-to-write properties, and Section 4 gives details of the tool’s inner workings and formal semantics. A case study, where FL is used to expose problems in a feature-rich university’s regulation, is discussed in Section 5. Section 6 compares related work and Section 7 concludes the article.

2 Specifiable Regulations?

It is often understood that regulations can be abstractly represented using the three well-known deontic operators for obligation (O), prohibition (F) and permission (P). Specifications can be also thought of as using the same operators. “The system must activate the brakes in no more than three seconds after the emergency stop button is pressed” is clearly an example of obligation. Prohibitions are also found: “it is forbidden that the server sends unencrypted passwords through the wire”. Permissions are not that frequent, but also possible: “if out of resources the system can drop requests until the processor is freed”.

Nevertheless, some types of statements found in legal documents are not common in software specifications. We divide them in two groups. The first one is composed of statements that have equivalents in specifications, but under a somehow different structure:

Contrary-To-Duty Obligations. Contrary-To-Duty (CTD) obligations are a way to model phrases like “The agent is required to do action X. If she does not, then action Y should be performed”. The first sentence is called the obligation and the last one the reparation, and we denote that as \( O_Y(X) \). Software specifications equivalents are conceivable, but the key difference is how to treat \( O(X) \land O_Y(X) \). A Software Engineering perspective could read the formula as “obligation to perform X and obligation to perform X, plus reparation Y in case of failure of X, equals obligation to perform X plus reparation Y”, considering that \( O_Y(X) \) somehow supersedes plain \( O(X) \). However, a legal point of view is that \( O_Y(X) \) means that a behavior that does not satisfy X but performs Y is still legal, while the same behavior is illegal for the formula \( O(X) \), that does not contemplate a reparation.

Amendments to Deal with Contradictions. A legal corpus might contain the formula F(kill), and then be modified to allow self-defense by the addition of P(kill in self-defense). It is hard to consider that such a corpus is contradictory, yet a software engineer will rather use the specification F(kill unless in self-defense) that does not entitle a logical contradiction.

---

1 Another FL case study was presented in [13], yet it was much smaller both in size and concepts.
Permissions. Besides their use as exceptions to prohibitions, what are permissions exactly? The question has been raised and analyzed many times before (e.g., [2, 5, 21, 19]), so let us here only say that in software a permission is little more than no-determinism, while in a normative system a permission is a much complex individual. It is worth noticing that the ability of a user of some application to use or not some functionality is not a permission, it is an obligation: the specification would probably say that the software is obliged to behave in a way or some other depending on the user’s choices. Some phrases lend themselves to confusion: “The user may print the displayed listing”, in the context of a software specification is just a simpler wording for stating that in order to comply with the specification the software is obliged to present the printing option, and, if chosen, print the listing.

Hierarchies. Legal corpora have hierarchies. For example a regional law might set a tax to $10 while the national law sets the same tax to $20. If national laws override regional ones in said normative system, the outcome is clearly $20. A software engineer might be tempted to say that such system is equivalent to one that sets the tax to $20. However, the real normative system permits that if the national tax-setting law is canceled, the tax is still set at $10 by the regional one, while the software engineer model’s does not.

Ontologies. In legal corpora ontologies are of common use. For instance, a law might set standards for animals such as pets, while there might be another, more specific for dogs, that might set different conditions. Although software engineers are familiar with inheritance and subclassing, it is not common to specify the requirements for a general class of actors and separately others, possibly contradicting the former, for a subclass. In a software specification they will be treated in an ad-hoc manner, for example, with a requirement for dogs and another for pets that are not dogs, thus avoiding the contradiction.

There are other types of statements that we believe have no equivalent in software specifications:

Nesting of Deontic Operators. Nesting of deontic operators, as in obligation to obligate (or to forbid, or to permit). In normative propositions such as “The judge is obliged to obligate the citizen to do X” there are two obligations (and two responsible parties): if the judge does not comply to obligate the citizen, she is to blame. If the judge complies but the citizen does not, then the citizen is at fault. The specification of a security system might use a similar phrase: “The system is obliged to obligate other users to do Y”, with Y being something like “not access each others private files”. In this case, if the system does not enforce Y, then the system is at fault. Also, if users fail to do Y, then the same system is also responsible. This type of predicates seem to be just a complex wording for only one obligation.

Care should be exercised when analyzing propositions where there is an apparent nesting of obligations, but can be rewritten without nesting. E.g., “The voting system should obligate users to deposit the ballot in the case in less than one minute or else face prosecution”. Such a phrase has a very different meaning if found in the legal norm that regulates electronic vote, or in the software’s specification. In the first case, it binds both developers and voters, while in the second only developers, as a software specification has no power over voters. In such a case, it can be
rewritten as “The voting system has to give one minute for the ballot to be deposited into the case. In case of timeout, prosecution actions should be initiated [i.e., by notifying officials].”.

Self-Referencing Modifications. Self-referencing modifications, as in “Let Article X of Bill Y be modified to mandate that from now on such and such”. Self here means that they modify the same normative systems that contains them. This should not be confused with any type of software compile-time or runtime configuration. In such cases the specification is still fixed and contemplates the different possible behaviors.

Deontic-Conditional Validity. Deontic operators whose activating condition is the validity of another deontic operator. A typical example is a conditional over an obligation as in “if at the time of the execution the agent were obligated to ... then she ...”. Software specifications might impose obligations based on the runtime operating conditions of the software, but they do not specify behaviour that is conditional to the runtime requirements, if that term makes any sense at all.

We found that the common denominator of the last group is considering the deontic operators as first-class operators, allowing for operators that take operators as parameters, check if they are active, and so on. However, we believe that if we consider only the legal documents that do not use such classes of predicates we can a) cover an important and varied amount of regulations that are common in the real world, and b) resort to the tools and technologies that can be used to analyze software specifications. The first group of predicates, we postulate, can be accommodated in such setting if treated properly.

We believe that this is good news for the deontic community, as it means that decades of effort in software-analysis tool building, optimizations and expressivity improvements can be leveraged, and there is no need to start from scratch and climb again the road from handling toy examples to real-size ones.

3 The FL Language

Our starting point is that many contracts and regulations can be formally analyzed with tools originally aimed for software specifications. This allows for making the most out of existing tools.

FL, introduced in [13], is built on the following premises:

- It aims at finding coherence problems, defined in a very pragmatic way: behaviours can not be permitted and forbidden, or obligated and forbidden, can not be plain mandatory or mandatory with CTDs, CTD reparations cannot be forbidden, etc. The complete list of covered topics is presented in Section 4.3.

- The input language is divided into a background theory and a set of rules. While the rules are LTL formulae with additional deontic operators aimed at capturing normative propositions, the background theory provides some simple constructs to describe the class of models over which the rules predicate.

---

2 Although there are some prototype dynamic specification languages with self-referencing capabilities, they are still far from being used in the current state of the practice.
• Models are linear and each one describes a possible legal behaviour. That is, behaviours that do not comply with the normative rules are discarded.

• If something is obligatory, then it must hold in every legal model at every possible state, and thus $O(\varphi)$ is interpreted as $\Box \varphi$. Prohibition of something is obligation to the contrary ($F(\varphi) \equiv O(\neg \varphi)$). The diamond operator works in the usual way, and $\Diamond \varphi$ looks ahead in the model for some state where $\varphi$ holds.

• Contrary-To-Duty obligations are supported as $O_{\rho}(\varphi)$, translated as $\Box (\neg \varphi \rightarrow \rho)$. $F_{\rho}(\varphi)$ is interpreted as $O_{\rho}(\neg \varphi)$. It is worth noticing that our encoding skips out most of the deontic logic paradoxes (see [14, Sect. 6] for details).

• Although based on translating to LTL, the input syntax is preserved to perform analysis at the meta-logical level.

Permission is thought of as absence of prohibition, but treated not as an operator that modifies the set of legal behaviours, but rather as a predicate that the legal models must satisfy. Otherwise, it is considered that the normative system under analysis (NSUA) has a coherence problem: it states that something is permitted when it actually is not. If the user flags the permission as an exception to a prohibition, as in $F(\text{kill})$ and $P(\text{self-defense killing})$, the internal representation of the affected prohibition is changed to reflect that. In the example, to $F(\text{kill unless self-defense})$.

The main component of the background theory is the action. An action can be happening or not at any moment. In FL an action is interpreted as a digital signal that can be on or off for an arbitrary number of consecutive states. Actions can represent proper actions by the implicit agents (e.g., action DriveCar) or non-controllable, external events (e.g., action CarCrash). There is no explicit notion of a role performing an action, so if they are needed the subject must be encoded in the action. We plan to add this feature in the future.

Some requirements seem to be easy to express, like being licensed in order to drive a car. It would seem that it suffices to forbid the DriveCar action if there is no prior GetLicense action. But the easiness is only apparent, as individuals can not only get their license, but also lose it. If the LoseLicense action is also considered, establishing whether an individual is licensed or not by means of a pure formula amounts to “parenthesis counting”, and that can be very challenging to write or plain impossible depending on the particular logic used. FL incorporates the notion of intervals, similar to the fluents of [12]. An interval is delimited by beginning and ending actions, in a such a manner that there is no nesting nor closing of an already closed interval. In the automata, a propositional variable is set to true or false, indicating whether the interval is open (see Section 4.1 for details). With intervals, the driving requirement can be posed as

\text{interval licensed delimited by actions GetLicense-LoseLicense}

and then simply stating $F(\neg \text{is licensed} \land \text{DriveCar})$.

Intervals can also be used to bound the occurrence of other actions:

\text{interval school_time delimited by actions CourseBegin-CourseEnds action TakeExam occurs only in scope school_time}

There is also the view of obligation not as something that must always hold, but rather as something that must be done, usually within some bound of time, sometimes called non-persistent obligation. We can deal with such expressions in two ways. Either with the $O^E(\varphi)$

\footnote{It does not mean the branching alternatives are not present, each possible alternative is present in one of the considered models.}

\footnote{This can be done automatically for simple cases and requires manual intervention in others.}
A Software Tool for Legal Drafting

operator that expresses an eventual obligation, one that ceases to exists as soon as it is fulfilled, or, if time bounds are provided (e.g., “You ought to return the borrowed books within school time”), by using intervals inside obligations: \( O(\Diamond_{\text{school\_time}} \text{ReturnBook}) \). Other interactions between obligations and deadlines are shown in [14].

As the background theory is translated into an automata, we can accommodate there (bounded) counting, allowing for expressions like \( O(\text{bbc} > 0 \rightarrow \Diamond \text{bbc} = 0) \), where the integer counter \( \text{bbc} \), borrowed books count, is handled by the automata incrementing it and decrementing it with every \( \text{BorrowBook} \) or \( \text{ReturnBook} \), respectively. Such formula properly states that every borrowed book must be returned, whereas the simple \( O(\text{BorrowBook} \rightarrow \Diamond \text{ReturnBook}) \) is satisfied by borrowing many and returning just one.

Also interesting are persistent obligations with one-time-each reparations, where violating the obligation once, or even performing the reparation should not free the subject from the obligation. An example of that is obligation to not cross red traffic lights, subject to a fee for each violation. Such CTD obligations are usually hard to express. For instance, \( F_{\text{PayFine}}(\text{RedCrossing}) \), says that red crossing is forbidden, except that a fine is paid immediately. If a diamond is added, as in \( F^3_{\text{PayFine}}(\text{RedCrossing}) \), then many violations can be canceled with one payment.

In FL the formula can be easily stated with the aid of a fines counter that increments with \( \text{BeFined} \) and decrements with \( \text{PayFine} \), and the following formulae: \( F_{\text{BeFined}}(\text{RedCrossing}) \) (red crossing is forbidden, under the penalty of being fined), \( O(\text{fines} > 0 \rightarrow \Diamond \text{PayFine}) \) (it is mandatory to eventually pay the fines).

Section 4 shows formal semantics, how the background theory is translated into automata, the handling of formulae and how the coherence checks are defined and performed.

4 Semantics & Inner Workings

4.1 Background Theory

FL’s background theory is translated into a Büchi automata network with additional code that controls state transitions and handles state variables. Each run of the automata defines a standard LTL model over which the rules formalized in the next section predicate. The tool can use SPIN [20], DiVinE [3] or NuSMV [9] as backends for model checking, but the encoding presented here will use an agnostic dialect.

Time modeling is discrete, considered as succession of states, some of which have a proper name to refer to timestamps of interest for the NSUA. In FL an action is thought of as a digital signal that can be on or off for an arbitrary number of consecutive states. Thus, each action is represented by an enumerated variable that covers the HAPPENING and NOT_HAPPENING states. This is an easy way to model time density, as the net effect is that any event can happen while others are taking place.

One single automaton is responsible for controlling all the variables. It has a single state called running and non-deterministically guarded self-transitions. Let’s exemplify with the action \( \text{BorrowBook} \) (e.g., from the library).

declare enum borrow_book = NOT_HAPPENING

running -> running

\begin{align*}
guard \text{borrow_book} = \text{NOT\_HAPPENING} & \rightarrow \text{set borrow_book} = \text{HAPPENING}; \\
guard \text{borrow_book} = \text{HAPPENING} & \rightarrow \text{set borrow_book} = \text{NOT\_HAPPENING};
\end{align*}
The automaton is therefore defined as follows:

\[
\begin{align*}
&\text{borrow\_book} \\
&\text{NOT\_HAPPENING} \rightarrow \text{HAPPENING} \\
&\text{HAPPENING} \rightarrow \text{NOT\_HAPPENING}
\end{align*}
\]

The automaton can switch the value of \text{borrow\_book} or leave it as it is, changing other variables. At the automaton level only one variable changes at a time, resembling the one input assumption of SCR [4]. As said before, at the normative level many actions can be taking place at the same time.

The encoding shown so far allows to easily refer to whenever actions are happening or not, but sometimes it is required to express that an action has happened completely (i.e., it has finished taking place) or just happened (i.e., has finished taking place in the current state). For instance “account the loan after borrowing”, needs to refer to a moment when the BorrowBook action is not happening after having happened. To facilitate this, another state called \text{just\_happened}, of a type sometimes referred to as \text{urgent} or \text{committed}, is included in the automaton. The semantics of such type is that whenever an execution reaches one of these states, of all the available options, the automaton must execute a transition that leaves a committed state:

\[
\begin{align*}
&\text{running} \rightarrow \text{running} \\
&\text{guard} \text{ borrow\_book} = \text{NOT\_HAPPENING} \rightarrow \text{set} \text{ borrow\_book} = \text{HAPPENING}; \\
&\text{running} \rightarrow \text{just\_happened} \\
&\text{guard} \text{ borrow\_book} = \text{HAPPENING} \rightarrow \text{set} \text{ borrow\_book} = \text{JUST\_HAPPENED}; \\
&\text{just\_happened} \rightarrow \text{running} \\
&\text{guard} \text{ borrow\_book} = \text{JUST\_HAPPENED} \rightarrow \text{set} \text{ borrow\_book} = \text{HAPPENING};
\end{align*}
\]

With this new possible value for the state variable actions can be not happening for an arbitrary number of states, switch to HAPPENING, also for an arbitrary number of states, but before switching to NOT\_HAPPPENING again they must spend one state as JUST\_HAPPENED. Finished actions are easy to pose with this new state: in \(O(\text{BorrowBook} \rightarrow \Diamond \text{AccountLoan})\). From the formula perspective, the terms BorrowBook and AccountLoan are translated to propositional variables whose value is \text{borrow\_book} = \text{JUST\_HAPPENED} and \text{account\_loan} = \text{JUST\_HAPPENED} respectively.

If actions have output values, as in:

\text{action} \text{BorrowBook output values} \{ \text{available, in\_house\_reading\_only, not\_available} \}

another enumerated variable \text{borrow\_book\_output} is added to the automaton and the encoding turns into:

\[
\begin{align*}
&\text{running} \rightarrow \text{just\_happened} \\
&\text{guard} \text{ borrow\_book} = \text{HAPPENING} \rightarrow \text{set} \text{ borrow\_book} = \text{JUST\_HAPPENED}, \\
&\text{guard} \text{ borrow\_book\_output} = \text{AVAILABLE}; \\
&\text{guard} \text{ borrow\_book} = \text{HAPPENING} \rightarrow \text{set} \text{ borrow\_book} = \text{JUST\_HAPPENED}, \\
&\text{guard} \text{ borrow\_book\_output} = \text{IN\_HOUSE\_READING\_ONLY}; \\
&\text{guard} \text{ borrow\_book} = \text{HAPPENING} \rightarrow \text{set} \text{ borrow\_book} = \text{JUST\_HAPPENED}, \\
&\text{guard} \text{ borrow\_book\_output} = \text{NOT\_AVAILABLE};
\end{align*}
\]
So the output is set non-deterministically to any of the possible values and it is retained until the next setting. Similarly, if the action has extra guards, they are added to the guard of the transition.

As we mentioned before, intervals are another important feature of the language. Let’s suppose books can only be borrowed during the academic year:

```
interval academic_year delimited by actions BeginYear-EndYear
action BorrowBook occurs only inside academic_year
```

A boolean variable, `academic_year_opened` is added to the automaton, and is set by the transitions that represent the respective actions. Also, it is added as a guard, so no `BeginYear` happens inside an academic year and no `EndYear` happens outside one. Similarly, it is added as a guard for `BorrowBook`.

Expressivity-wise, counters are a very powerful feature: they are basically a non-negative integer variable with actions that either increment, decrement or reset their value. The implementation is straightforward: an integer variable is added to the automaton, and it is manipulated in the respective transitions.

Temporal actions are a way to implement timers. The temporal actions `ta_1,...,ta_n` clause declares a sequence of time points that follow the specified order and let an arbitrary number of actions happen between them. The implementation uses another synchronizing automaton with one state representing each `ta_i` and others representing the time intervals after `ta_i` and before `ta_i+1`.

### 4.2 FL’s Syntax and Semantics

We present here a formal definition for the rule-stating part of FL^E. Although allowed in the input language, general terms like `bcc > 0`, `action.value`, etc. are abstracted away as propositional symbols in the presented syntax. There is no need to model them explicitly since they can be thought of as encoded in propositional values that later the model handles in the proper way. The same happens with actions `a`, which are abstracted as the proposition `a=JUST_HAPPENED`.

**Syntax.** Let `PROPS` be a countable infinite set of symbols, `INTERVALS ⊆ (PROPS × PROPS)` a set of intervals, and `FORMS` the set of FL formulae in the signature `<PROPS, INTERVALS>` defined as

```
INNER_FORMS ::= ⊤ | ⊥ | p | ¬φ | φ_1 ∧ φ_2 | ◇φ | ◇_iφ
FORMS ::= O(φ) | F(φ) | O_p(φ) | P_p(φ) | O_E(φ) | P(φ),
```

where `p ∈ PROPS`, `φ, ρ, φ_1, φ_2 ∈ INNER_FORMS` and `i ∈ INTERVALS`. We usually work with (finite) sets of `FORMS` when specifying a NSUA, so conjunction between formulae in `FORMS` does not need to be formally defined. We will usually write one formula below another, implicitly defining a conjunction between them. Some operators could have been defined in terms of others, but we intentionally define all of them at this level since our tool considers them differently for coherence analysis (see Section 4.3 for more details).

---

^5This reduced presentation does not allow for nesting of deontic operators, a restriction only introduced to save space in this article. For the same reason we omit the repared version of the `O_E` operator.
Semantics. We give FL semantics by providing a translation \( Tr \) from FL into classic LTL\(^6\) as both work over the same class of models. Let \( \mathcal{F} \) be a set of FL formulae whose intended meaning is defining the set of legal models for the NSUA. Recall that LTL models are linear infinite structures that represent possible runs on the automata defined by the background theory. We first split permissions from the rest and define the \( Tr \) domain as \( \mathcal{F}_\rho = \{ \varphi \mid \varphi \in \mathcal{F} \} \) such that \( \varphi \) is not of the form \( P(\psi) \). \( Tr \) acts as the identity for the inner-forms constructions not explicitly specified and it is assumed that the target LTL signature has the implicitly defined propositions involved in the translation (like \( i_{\text{opened}} \) for each interval \( i \)).

\[
\begin{align*}
Tr(\circ_1 \varphi) &= i_{\text{opened}} \to (i_{\text{opened}} U Tr(\varphi)) \\
Tr(F(\varphi)) &= \Box \neg Tr(\varphi) \\
Tr(\neg \varphi) &= \neg Tr(\varphi)
\end{align*}
\]

\( Tr \) can be lifted to take a set of FL formulae and return the set of their translations.

Let’s now consider an automaton \( A \) defined by the background theory and the class of models \( \mathcal{C}_A \) that represents all possible runs on \( A \). Let \( \mathcal{F} \) be the set of FL formulae that encode the NSUA. The class of legal models defined by \( \mathcal{F} \) over \( \mathcal{C}_A \) is defined as

\[\mathcal{C}^\mathcal{F}_A = \{ \mathcal{M} \in \mathcal{C}_A \mid \mathcal{M} \models Tr(\mathcal{F}_\rho) \} .\]

That is, every legal model must satisfy the obligations and prohibitions specified by \( \mathcal{F} \).

But what about permissions? Permissions are actually a check that must be performed on \( \mathcal{C}^\mathcal{F}_A \) to ensure coherence. The condition that \( \mathcal{C}^\mathcal{F}_A \) must fulfill is the following: for every \( \varphi \) of the form \( P(\psi) \) in \( \mathcal{F} \) there must be a model \( \mathcal{M} \) in \( \mathcal{C}^\mathcal{F}_A \) such that \( \mathcal{M} \models Tr(\psi) \). I.e., if something is permitted then the rest of the NSUA does not prevent it from happening.

We are going to expand the concept of coherence in the next section by analyzing other cases of interest.

4.3 Analyzing Coherence

A difficult topic in deontic logic is the concept of coherence of a normative system: whenever the set of rules is “contradictory” in any sense. As stated in the literature (e.g., \([19]\)), the problem cannot be simply reduced to logical consistency. We take a pragmatic approach where a normative system is not coherent if it has any of a list of problems.

While some of them are straightforward to check, others require more sophistication. Let’s see an example of each class. To fix notation, \( \varphi \# \psi \) means that \( \varphi \) is incompatible with \( \psi \) and the following equivalences hold: \( O(\varphi) = O_\bot(\varphi) \), \( F_\rho(\varphi) = O_\rho(\neg \varphi) \).

To check that there are no forbidden obligations (i.e., that there is no pair of rules \( O(\varphi) \) and \( O(\psi) \) such that \( \varphi \# \psi \)), we need to check that there is at least one possible legal behaviour that satisfies both the background theory and the complete set of formulae. To do that, all the rules \( r_i \) are conjuncted into \( \Phi = \bigwedge Tr(r_i) \), and both the automata and \( \neg \Phi \) are fed to the model checker. If \( \neg \Phi \) is not satisfiable the model checker will output a counter example trace, \( \tau \). \( \tau \) satisfies the negation of \( \neg \Phi \) so it is the legal behaviour we were looking for. Should \( \neg \Phi \) be satisfiable, that means that \( \Phi \) is not, so a backtracking-type of procedure should be started to find the “guilty” rules. How to improve this procedure is an active area of research.

\(^6\)That is, the basic version of LTL with the standard boolean connectives plus the until operator (from which the diamond is defined).
We are also interested in checking that there are no “contradicting obligations”: rules $O_\rho(\phi)$ and $O_{\rho'}(\psi)$ even if it is not the case that $\rho \# \rho'$. That is, incompatible obligations, but with compatible reparations. If that were the case, there would be a legal behaviour: doing the reparations $\rho$ and $\rho'$, yet it makes no sense that the primary obligations $\phi$ and $\psi$ are impossible to comply with.

To check for that, we build $\Phi'$ as $\land \text{Tr}(O(\phi_i))$ for all the rules $r_i = O_{\rho_i}(\phi_i)$. Then the automata and $\neg \Phi'$ are fed to the model checker as before. If $\neg \Phi'$ is satisfiable there is no way to comply with all the obligations, leaving aside the possible reparations. If $\neg \Phi'$ is not, then, as before, the $\tau'$ counter example trace is a possible way of complying with all the primary obligations.

It should be noted that this last check is one of the analyses where preservation of input syntax is important and the translation of repaired obligations is not done directly.

The following conditions also violate coherence:

- **Forbidden reparations**, such as $O_\rho(\phi)$ and $F(\rho)$. In that case the reparation is only nominal, as it is indeed forbidden. $O_\rho(\psi)$ and $O(\psi)$ with $\rho \# \psi$ is another case of the same problem.

- **Obligations with conflicting reparations**. If $\rho \# \rho'$ and $O_\rho(\phi)$ and $O_{\rho'}(\phi)$ is found then there is a contradiction in how the obligation $\phi$ could be repaired. Note that this does not mean that there is no legal behaviour, as respecting $\phi$ is always allowed.

- **Impossible permissions**. Whatever was said to be permitted should be possible as was explained in Sections 3 and 4.2.

- **Unrealizable background theory**. The background theory should not generate an empty set of traces, which would mean it is, by itself, logically inconsistent.

## 5 Case Study

To show FL at work we analyze some excerpts from a university regulation. This case study focuses on conflicts that can arise from students being able to be also teachers. Although fictional, the inspiration is real. The case study features the use of actions, intervals, counters, obligations, prohibitions, permissions and many forms of coherence analysis.

The analyzed excerpt is the following:

1. **Chapter 1, Students.**
   
   (a) Every individual that has enrolled for a career and has not yet graduated from it is considered to be a student.
   
   (b) Students should respect each other. Major disciplinary faults are punished by forbidding the entering to university premises for one year after the fault.
   
   (c) Students have the following rights: ..., participate in research activities, ...

2. **Chapter 2, Teachers.**
   
   (a) There are three teaching categories: c1) Undergraduate Teaching Assistant (aka UTA), c2) Teaching Assistant and c3) Professor.
   
   (b) To become a teacher, aspirants must apply when the selection opens. The selection will be made based on the following criteria for each category: [omitted, not relevant to the case study]
   
   (c) To apply for the UTA category, aspirants must be students at the time of the selection.

---

7The UTA category position lasts one year in the real case. This particular spelling of the norm was chosen because it is desired that only students fill this position, yet allow them to keep the job if they graduate after the selection.
(d) Teachers must perform their duties, starting 30 days after the selection ends.
(e) Working from home is allowed, but teachers must spend at least one day a week in the premises of the university.

3. Chapter 3, Research.
(a) Research activities can only be pursued by members of approved research groups.
(b) Research groups are conformed by accredited professors or teaching assistants.

4. Chapter 4, University Library.
(a) Every borrowed book should be returned by the end of the month.\footnote{The more realistic requirement of returning within days is also possible but more involved, thus the simpler version is preferred for space reasons.}
(b) Students and teachers are subject to a fine for not returning books in time.
(c) As students are generally on a budget, their fine should be low.
(d) Teachers should be an example of conduct, thus their fine should be strictly higher that the students’ one.

Let’s model the student’s chapter first. To avoid clutter some abbreviations will be used, such as not declaring actions that are used to bound intervals if they do not take any extra parameter, as it is the case for declaring what a student is.

\textbf{interval student delimited by actions Enroll-Graduate}

Regarding discipline, they should not commit disciplinary faults or be banned from entering the premises for one year. To model that we will define two intervals: one that spans from the fault to one year after, and another that accounts for being inside the building.

\textbf{interval ban delimited by actions CommittFault-OneYearPassed}

\textbf{interval inside_building delimited by actions Enter-Exit}

And stipulate the prohibition:

\[ F_{\text{ban} \neg \text{inside_building}}(\text{CommitFault}) \]

meaning that the fault should not occur but if it does, during the ban period the student cannot be inside the building (\text{is}\_\text{inside}\_\text{building} is a boolean variable made true between the bounding actions of the interval).

Article 1c permits students to do research, in what can be thought of as a case of antithetical permission \cite{26}: a permission set to invalidate future prohibitions.

\[ P(\text{is}\_\text{student} \land \text{DoesResearch}) \]

Now let’s focus on the teachers’ selection process. There is the selection interval and its possible outcomes: being elected in any of the categories, or not being elected at all.

\textbf{action ElectWinners output values \{ teacher\_c1, teacher\_c2, teacher\_c3, no\_teacher \}}

\textbf{interval selection delimited by actions OpenSelection-ElectWinners}

\textbf{action Apply only occurs in scope selection}

For simplicity let’s only model the requirements for the c1 category (UTAs) of still being a student:

\[ O(\Diamond_{\text{selection}}(\text{Apply} \rightarrow \text{is}\_\text{student})) \]
interval grace_period delimited by actions ElectWinners-30DaysAfter
interval on_duty delimited by actions 30DaysAfter+inf
interval week delimited by actions StartWeek-EndWeek occurs only in scope on_duty repeatedly

It is mandatory to have a weekly visit:

\[ O(is\_teacher \rightarrow \Diamond_{\text{week}} \text{Enter}) \] (4)

with \( is\_teacher \) defined as:

\[
\text{macro } is\_teacher = \text{Apply} \land
(ElectWinners.teacher.c1 \lor ElectWinners.teacher.c2 \lor ElectWinners.teacher.c3)
\]

Regarding Chapter 3, the restriction of research activities to members of research groups is:

\[ F(\neg \text{JoinResearchGroup} \land \text{DoesResearch}) \] (5)

while the requirements of being TA or professor to belong to a research group is:

\[ F(\neg (\text{ElectWinners.teacher.c2} \lor \text{ElectWinners.teacher.c3}) \land \text{JoinResearchGroup}) \] (6)

Then there is the borrowing of books, similar to both students and teachers, that requires the \( \text{bbc} \) (borrowed books count) counter and signaling months (it should be noted that the exact duration of a month is not important and it is thus abstracted away):

\[
\text{counter } \text{bbc} \text{ increases with action BorrowBook decreases with action ReturnBook}
\]

interval month delimited by actions MonthBegin-MonthEnd repeatedly

Although articles 4c and 4d are mainly motivational, there is one prescriptive consequence, even at the level of abstraction we are using – fines should be different: \textbf{StudentFine # TeacherFine}.

Article 4b is a CTD to 4a so we encode them as:

\[
O_{\text{StudentFine}}(is\_student \rightarrow \Diamond_{\text{month}}(\text{bbc} > 0 \rightarrow \Diamond_{\text{month}} \text{bbc} = 0)) \] (7)

\[
O_{\text{TeacherFine}}(is\_teacher \rightarrow \Diamond_{\text{month}}(\text{bbc} > 0 \rightarrow \Diamond_{\text{month}} \text{bbc} = 0)) \] (8)

When analyzed by \textsc{FormaLex} three coherence problems are pointed out. First, the reparation for not returning borrowed books is troublesome for teachers that are also students, and the tool signals a case of conflicting reparations, as there are traces where the implicit agent is indeed a teacher and a student. Looking at the NSUA nothing impedes students to become teachers, quite the contrary, as there is a UTA category.

What type of fine should a UTA be subject to? Whatever conclusion can be obtained by looking at the motivational articles 4c and 4d is disputable, as it is both true that UTAs should be an example of conduct because they are teachers, and that they are also on a budget, because UTAs’ salaries are symbolic. As the fining would probably be decided by a library’s clerk, there is high risk of different solutions applied to identical cases. It would be much better if this case could be decided by the norm-givers, and that is the sense of the warning.

The second problem is more involved and related to the weekly visit rule. The tool detects that the reparation of the rule that forbids discipline faults \textbf{[1] contradicts the obligation to the weekly visit \textbf{[4]}. Indeed, there is the possibility of a student committing a fault, thus being banned to enter the premises, applying for a teaching position, then being elected as UTA and not being able to comply with his weekly visit duty.

A possible solution is to forbid the application of punished students as in:

\[ \text{Note that although the name of the } \text{Apply} \text{ action is in the present tense, because of the translation, it is easier read if it thought of as written in the past tense. This reflects the fact that the intended semantics is to check if the action has occurred at some point in the past. The same happens with formulae 5, 6 and 9.} \]
action Apply only occurs in scope selection requires that ¬CommitFault

However, the problem persists, as the student can now apply (not yet being faulty), commit the fault, then being selected, to the same effect. A better solution would be to restrict the ElectWinners action so only non-faulty students can become teachers:

\[ F(\text{CommitFault} \land \text{ElectWinners.teacher}_c) \] (9)

This formula states that it is forbidden to commit the fault and to be elected UTA. Observe that if the fault is committed before becoming a teacher, then this formula forbids the election, which is the primary intention of it. On the other hand, if the fault is committed after becoming UTA, then again the reparation of the rule that forbids discipline faults (1) contradicts the obligation to the weekly visit (4). This situation is also noticed by the tool and notified to the user as a warning. If the user considers that it should be possible to repair the fault even under these conditions, then she must introduce appropriate modifications in the NSUA to envisage this situation. If, on the other hand, she thinks that it is correct to tighten the rules for teachers so they should not have a way to repair their faults, then the warning can be ignored.

The third problem is the collision of allowing research only to TAs and professors (rules 5 and 6) with the permission for students by rule 2. The fix is either the removal of the permission or the inclusion of at least UTAs into research groups.

There is another interesting aspect of this NSUA. Let’s assume somebody proposes a different writing for article 2c, that is supposedly more faithful to the spirit of not letting graduates fill the UTA category. She proposes defining what a graduate is:

interval graduate delimited by actions Graduate+inf

and plain forbidding the application of graduates. When queried about the validity of this new writing:

\[ ? F(\Diamond_{\text{selection}}(\text{Apply} \land \text{is_graduate})) \]

the tool responds that the prohibition does not hold as there is a trace where a student graduates and then enrolls again (say, for a different career) before applying. That perfectly satisfies the requirement that during the selection process applications must be done by students (3), as the implicit agent is also a student. It means that the phrasing of the rule (3) does not comply with the intended normative effects: restrict the UTA category to non-graduates; so the proposed alternate writing is actually the correct one. This “bug” is extracted from a real university’s regulation.

An interesting remark is that although the lack of roles in FL can be seen as a drawback, it turned useful in this case. Should roles had been available, probably many of the above mentioned bugs would have been concealed, including the real one.

6 Related Work

The idea of using a temporal logic for deontic purposes is not new. It can be traced to [1, 22, 16, 8, 11, 7, 23], among others who use some type of temporal-deontic logic. However, as far as we know they provide neither tool support nor a translation into a standard tool, and thus are not directly comparable to our work which is heavily tool-biased.

[18] deals with automated conflict detection in norms by using a tool that supports ontologies and translates normative propositions into a Prolog program, but the analysis is restricted to logical contradiction. More similar to our approach of analyzing contracts and regulations for
coherence problems are BCL \cite{15} and $\mathcal{L}$ \cite{24}. BCL is a contract specification language that is meant for monitoring, allows to build executable versions, can detect conflicts among rules off-line and provides features like clause normalization. However, it lacks support for temporal reasoning and background theories.

$\mathcal{L}$ is another logical language based on dynamic logic that treats deontic operators as first-class citizens. It is based on an ad-hoc tool and it neither uses background theories, nor discusses how to deal with some limitations of expressivity: for instance, in dynamic logic approaches it is easy to say that if a book is borrowed ($b$), a book should be returned ($r$) as $[b]O(r)$, but such rule matches the borrowing of multiple books and the returning of just one; the correct version is pretty involved or plain impossible depending on the particular logic.

7 Conclusions

Starting from the premise that normative systems are very similar to software specifications we propose a language and related tool set to analyze the former with tools designed for the latter. Besides the similarities, the decision is based on avoiding to build tools from scratch: model checkers have decades of effort in tool building, optimizations and expressivity improvements.

In this article we analyzed a feature-rich, real-life inspired, case study. Although fictional, we believe it shows the power of both the tool and its underlying definition of coherence to spot defects that are not self-evident. Our next step is dealing with a 100% real case study to investigate the payoff of having to logically encode the NSUA vs. the severity of the defects found.

References


Contract-Based Discovery in Sensor Web *

Javier Cubo
Dept. Computer Science
Univ. of Málaga, Spain
cubo@lcc.uma.es

Nadia Gámez
Dept. Computer Science
Univ. of Málaga, Spain
nadiga01cc.uma.es

José Antonio Martín
Dept. Computer Science
Univ. of Málaga, Spain
jamartin@lcc.uma.es

Lidia Fuentes
Dept. Computer Science
Univ. of Málaga, Spain
lff@lcc.uma.es

The evolution from traditional wireless sensor networks towards the Internet of Things has originated the need of interconnecting heterogeneous devices. Sensor Web represents a web of sensor networks accessible by means of service interfaces which enable an interoperable usage of sensor resources. Hence, a shift in the sensor-as-a-service paradigm is required. Dynamic, scalable and adaptive service discovery is crucial in this new scenario of diverse and scarce resources. Contracts present a versatile solution to reason, negotiate and discover services based on their characteristics and capabilities, as well as on their current states. Therefore, we propose a novel approach to address discovery of heterogeneous sensor nodes based on contracts. We use ontologies to capture and support sensor descriptions with semantics concepts and their relationships, increasing the interoperability and allowing complex reasoning in the discovery. We have implemented the discovery process as a service provided by FamiWare, a middleware family for Ambient Intelligence.

1 Introduction

The Future Internet has emerged as a new initiative to pave a novel infrastructure linked to objects and things of the real world, ranging from mobile devices and sensors to cars and electrical appliances, in order to meet the changing global needs of business and society. Future Internet applications will have to support the interoperability between many diverse stakeholders or objects by governing the convergence and lifecycle of Internet of Contents (IoC), Services (IoS), Things (IoT), and Networks (IoN). These applications should handle dynamic and continuous changes, for instance, in the provisioning of services, availability of things and contents, connectivity of networks, mobility of wireless connected objects and so on. These objects will sometimes have their own Internet Protocol addresses, be embedded in complex networks and use sensors or tags to obtain information from their environment (e.g., recording temperature or humidity), and/or use actuators to interact with it (e.g., air conditioning valves that react to the presence of people) [3].

Specifically, sensor networks are increasingly being used for monitoring different domains, such as environmental, agriculture, industrial or traffic. The Open Geospatial Consortium (OGC) proposes the Sensor Web Enablement (SWE) framework in order to integrate heterogeneous sensor. SWE promotes Service-Oriented Architectures (SOAs) to specify Web service interfaces used for accessing sensor data, controlling sensors and alerting based on measures. In OGC, a Sensor Web represents a web of sensor networks accessible by means of service interfaces which enable an interoperable usage of sensor resources [1]. Hence, a shift in the sensor-as-a-service paradigm is required. Dynamic, scalable and adaptive service discovery is crucial in this new scenario of diverse and scarce resources. Contracts

---

*This work is partially supported by the projects TIN2008-05932 (ReSCUE) and TIN2008-01942 funded by the Spanish Ministry of Science and Innovation (MICINN) and FEDER; and the projects P07-TIC-03131 and P09-TIC-05231 (FamWare) funded by the Andalusian local Government.

1In the sequel, we use the terms devices and sensors indistinctly to refer to whatever node in a network.

2http://www.opengeospatial.org/projects/groups/sensorweb

Accessed on 20 April 2011.

To appear in EPTCS.
Contract-Based Discovery in Sensor Web

present a versatile solution to reason, negotiate and discover services based on their characteristics and capabilities, as well as on their current states.

In this paper, we propose a novel approach to address discovery of heterogeneous sensor nodes based on contracts. Specifically, we use contracts in order to specify a query and find out the most suitable sensor nodes for that query. In our approach, the contracts could be negotiated in case there not exist any node fulfilling the initial query. Moreover, the system could be reconfigured dynamically in order to satisfy the contract. In addition, we take advantage of ontologies to capture and support sensor descriptions with semantics concepts and their relationships, increasing the interoperability and allowing complex reasoning in the discovery.

An overview of our approach is presented in Figure 1, which will be detailed throughout the paper. This figure represents all the possible interactions supported by our approach. Our proposal supports a wide range of devices with different capabilities and requirements and hence it comprises a range of different messages.

The basic message is query delegation, which is a discovery request that must be replied either synchronously or in an eventual service result message. Query delegation can entail a subsequent chain of delegations to broaden the scope of the discovery process. Additionally we have dynamic state requests which demand the current dynamic state of the receiving device. This request is also replied with the service result message. Unlike query delegation messages, dynamic state requests are never delegated.

Nodes with larger capabilities in memory and energy may (optionally) serve as a registry of services or, more precisely, a cache of known services. This cache, known as knowledge base of the node and represents the scope of services known to it. For example, in Figure 1 the laptop is aware of the services in scope a, the desktop computer knows the services in both the scopes a and b whereas the router is contains in its knowledge base the single mote of scope c.

The knowledge base is populated in two ways: i) via the chains of query delegation that the node participate in, and hence intermediates in the service result messages; and ii) via knowledge base update messages that can be sent to other nodes with knowledge base. The latter kind of messages is replied with a knowledge base data which contains the requested knowledge base.

We build our approach on two assumptions: to try to minimise the number of communications required for a distributed discovery process and large messages consume more resources (mainly battery) than several small control messages. In this sense, those communications which entail service informa-
tion (i.e., service result and knowledge base data) are the only large messages in our approach, and they should be used sparingly.

Our discovery process, which has implemented as a service provided by FamiWare, a middleware family for Ambient Intelligence, is conceived to address several challenges occurring in Wireless Sensor and Actuators Networks (WSAN).

Hierarchical. Certain nodes have a hierarchy (e.g., based on resources or spanning tree) which will be used to delegate queries. For instance, in Figure 1, the laptop might be the head or sink of the motes in scope a whereas the mote in scope c depends on the router nearby.

Indirect. It should be able to discover services not directly known by the head of the hierarchy. For instance, the initiator must be able to discover the mote in scope c.

Dynamic scope. New services which fulfill the query can be discovered even after the query has been initially answered.

Dynamic characteristics. The query can be composed of static features (such as functionality, ID and hardware capabilities) and dynamic features (such as QoS, remaining energy and location, for instance). The former can be cached and stored in the knowledge base to quicken future discoveries but the later have to be consulted when the services are needed.

Aggregated results. We aim at supporting queries that must be collaboratively answered by several services working together. Therefore, the possible results of the discovery are not just a set of candidate services but a set of sets of services that fulfill the query altogether.

Robustness. A service can always be discovered providing that it is possible to (indirectly) communicate with it. Nonetheless, we assume that it is provided a transport layer in charge of message communication and routing. Therefore, the transport layer is considered out of scope.

The rest of this paper is structured as follows. Section 2 presents our model to specify contracts. In Section 3, the discovery process is described. Section 4 details the discovery process incorporated into FamiWare middleware, in the scope of this work. Finally, in Section 5, some conclusions and comparison with related works are drawn, and plans for future work are outlined.

2 Contract-Based Model

In this section, we present our model based on contracts used to reason, negotiate and discover heterogeneous sensor nodes. Contracts specify queries on sensor characteristics and capabilities, as well as on their current states.

2.1 Sensor Characteristics

Resources in IoT are limited. Communication in devices with limited capabilities causes a major consumption of their resources, such as the battery which is very scarce in these devices. Hence, one of the main goals in a discovery process in Sensor Web is to reduce as much as possible the communication between devices. In order to achieve this goal, when a device needs to discover other devices or sensors satisfying certain characteristics, the discovery process should not rely on expensive message flooding. Instead, we propose to perform queries in a two-step process by using a knowledge base or repository that contains information about the sensor networks related to certain sensor characteristics. This repository is updated when a new sensor is incorporated to the network, by including its characteristics to such
knowledge base. The first step of the process consists in a negotiation by means of the knowledge base, and in the second step the queries are directly executed on the devices obtained in the first step.

On the one hand, in the first step, the knowledge base contains the static characteristics known at design-time corresponding to the devices registered in such a base. These static characteristics are the following: (i) Sensor Identification, the identification of a node within a network; (ii) Sensor Description, the description of a node; (iii) Sensor Type, the type of a sensor node (e.g., static, mobile, mote-like, body sensor, etc.); (iv) Sensor Location, the position of a static node in a network; (v) Sensor Capabilities, the sensing units of the sensor node (e.g., temperature, humidity, etc.); (vi) Sensor Data, the data that can be sent by a node to other nodes of the network (e.g., units of measurement); and (vii) Sensor Lifetime, the initial lifetime of the sensor.

On the other hand, in the second step, we assume the dynamic characteristics can change at run-time. Then, these characteristics are not represented in the knowledge base, thereby it is required to ask the devices for them. Dynamics characteristics may be the following: (i) Sensor Battery, the level of a sensor node battery; (ii) Sensor Node Traffic, the amount of traffic sent and received by the node; (iii) Sensor Location, the position of a mobile node in a network (by using GPS or relative positioning algorithms); (iv) Sensor Active Capabilities, the sensing units which are active at run-time; and (v) Sensor Node State, the state of a node (e.g., alive, dead, idle, slept, etc.).

Next, we describe the syntax of our contract-based model, as well as how a query is specified in our contract model. In addition, we briefly mention the semantic annotation we use to relate concepts of the query with those corresponding to sensor nodes.

### 2.2 Contract in a Nutshell

In our proposal, a contract is specified as a query described in a XML-based language for advertising the capabilities of sensor nodes providers according to certain preferences. The knowledge base can be specified as a XML document. Technologies such as XQuery, XPath or XPointer can be used to manipulate well-formed XML documents consisting of information items and structural metadata which define the relationships between the items. In order to represent queries, we use XQuery [12], which is a query and functional language recommended by W3C for providing flexible query facilities to extract and manipulate data expressed in XML by means of expression [3]. Then, in our proposal, XQuery allows the selection of sensor nodes by means of the different criteria or preferences established in the query, considering that all sensor nodes are registered in the knowledge base whose information is represented in XML. In XQuery, the queries can be made up of up to five different types of clauses, following the SQL-like FLWOR expression for performing joins: For, Let, Where, Order, and Return.

**Example 1:** We present a query specified in XQuery on the knowledge base encoded in an XML document called “network.xml”. This query lists all the sensors with capabilities temperature and humidity, and being mobile mote-like nodes as static characteristics; and with battery more than 50% as dynamic characteristic.

```xml
for $node in doc("network.xml")/Middleware/Device
where every $device in $node//Device satisfies ($device//SensingUnits/Temp and $device//SensingUnits/Humidity and $device//Type="MoteLike"
and and $device//Type/Mobility="Mobile" $device//Battery>50%)
order by $/device//Battery
return $node/Device
```

As we will explain in Section 4, the knowledge base is represented by using Feature Models [8]. We also use a feature modelling tool, called Hydra [5], in order to solve the queries.

Since a network is made up of heterogeneous sensor nodes, the same sensor characteristics may be described by using different concepts (e.g., Temp or temperature). Therefore, we use ontologies to represent both static and dynamic characteristics with semantic concepts and their relationships in order to lead to the development of Semantic Sensor Web (SSW), increasing the scalability and interoperability and allowing complex reasoning in the discovery process [11].

Among the different languages that focus on Semantic Web technologies, W3C recommends OWL to capture semantic descriptions. OWL proposes a formal representation of a set of concepts within a domain by capturing the relationships between those concepts. This is called an ontology, and OWL is currently the de facto standard for constructing ontologies. Using OWL ontologies a sensor node could describe its characteristics. Therefore, using a sufficiently expressive semantic model, the queries can be specified as both qualitative and quantitative preferences.

A contract creates a negotiation, explained in the next section, that comprises both static and dynamic characteristics of the sensor nodes, and may change dynamically depending on the nodes found in our discovery process (see Figure 1) described in Section 3.

### 2.3 Static and Dynamic Negotiation

As aforementioned, the negotiation of our contracts is divided in two phases: firstly we solve the static characteristics, and secondly the dynamic ones. In the following steps we present the negotiation process.

1. First, the knowledge base, which contains the information of all sensor nodes registered in this base, is queried for the sensor static characteristics. This corresponds to the message number 1 in Figure 1. All those nodes fulfilling the required static characteristics are obtained and ranked according to their suitability to the contract. In our example, mobile mote-like nodes that are able sensing temperature and humidity are selected.

2. If no sensor node is found, then a negotiation to reduce the contract could be performed in two ways (under the user’s decision). Following with our example, let us suppose there not exist mobile mote-like nodes with the capabilities to sense temperature and humidity at the same time.
   a) Devices not satisfying the less prioritised characteristics could be obtained. User must provide the information about the prioritisation, otherwise a default order will be used. In our scenario, mobile characteristic could be considered the less prioritised one.
   b) Devices satisfying the major number of characteristics could be selected, considering in this case that the characteristics no fulfilled are in between the less prioritised ones. For instance, here nodes which can sense temperature and humidity, even if they do not satisfy the other static characteristics, could be selected.

3. Once the devices fulfilling the static characteristics have been discovered, our contract establishes an order with respect to the preferences of the static characteristics of the selected devices. This ordering is taken into account for the search of the devices that could satisfy the dynamic characteristics. This search (message number 4 in Figure 1) returns as many devices as have been requested in the query, ending the search at the moment in which those devices are found. In such a way, we avoid to ask more devices than needed, by reducing once again the communication. For our example, we assume that in the previous steps corresponding to the static phase, three nodes has been returned, and we need only one of them that fulfills the dynamic characteristic, i.e., its battery must be more than
50%. Therefore, this dynamic phase asks one by one until find out the first one with the battery level more than 50%.

4. If after this search, no device has been discovered, then three process could be performed (under the user’s decision).
   a) Initialise a negotiation to reduce the contract of the dynamic characteristics. For instance, the required battery level could be reduced to 40%.
   b) Reinitialise the initial negotiation by reducing the contract with respect to the static characteristics. We proceed in a similar way to step 2.
   c) Reconfigure the system when it is possible, in order to obtain the enough number of nodes satisfying the contract. For instance, if our query will request for the battery lifetime instead of the battery level, then the negotiation could reconfigure a node with a similar battery lifetime than the required one by reducing some sensing tasks or the frequency of sensing.

In the next section, we present our discovery process based on contracts and the previous negotiation process, as well as current data diffusion mechanisms in WSAN.

3 Discovery Process

Our discovery approach has been inspired by data diffusion mechanisms in sensor networks [6, 7]. Depending on their capabilities, nodes might have a knowledge base (see Section 2) and a store to keep track of recent discovery requests and a cache of their results. The steps of the discovery process are enumerated as follows:

1. If the query was already known by the receiving node, return the results which were associated with it, if any. Otherwise store the query, its source node and the current time. When the device is aware of new services, these are evaluated against stored queries (i.e., matched) and, if they match any of these queries, a result update is emitted. This result update is a message analogous to the one containing the discovery results but it is sent when the query is considered already solved. The matching is performed according to the contract evaluation explained in Section 2.3.

2. Check against the local knowledge base to find possible matches. Keep track of partially matching services.

3. If it exists, delegate the query on the direct parent in the hierarchy only if the query was not received/delegated from it. Then, the parent starts the process from step 1.

4. If no enough services are found then evaluate the next closest match.
   a) If the current device has enough resources, request a knowledge base update.
   b) Otherwise delegate the query. In this case, the intermediary remembers the request and dynamically provides new results if they appear later.

5. Repeat step 4 until the request is fulfilled or no more candidates are found.

6. Once the query is solved, store the results under the query, source and time stored in step 1.

7. If it was a delegated query, rely the answer to the source node (i.e., the one stored in step 1).

8. If the node receives a control message notifying about a query expiration (i.e., a forget-query message), or after a certain timeout, remove the query from the store.
The search in the local knowledge base (step 2) is done following the procedure described in Section 2. Let us highlight that, if the discovery query depends on some dynamic characteristics, these are requested on-demand to the services which partially match the query. For instance, the message number 4 in Figure 1 is a request of the current dynamic characteristics.

For devices with limited storage or computation capabilities which lack of knowledge base, or in those cases where no match is found locally but the current node belongs to a hierarchy, the query is delegated on the parent (step 3). If a partial solution has been found already, send it to the parent along with the query. All subsequent computation and communications are performed by the delegate, which will only reply either with the most suitable results or a negative answer. This corresponds to message 1 in Figure 1.

At this point (step 4), no ancestor in the hierarchy is aware of any complete result for the query. Therefore, it is necessary to ask to other known devices which may be closer to the desired services. We assume that services with similar characteristics are deployed in devices which usually know each other. For this reason, we calculate a tree distance (see [?]) between the query and each of the services known to the current node. The device where it is deployed the closest matching service is designated as the next device to consult (i.e., the next closest match). In this way, we achieve an indirect discovery mechanism. This indirect querying is exhaustive in the sense that, if no result is found from the designated device, the next one in the ranking is consulted until the current node finds results or runs out of alternatives, therefore obtaining robustness in the discovery.

Once another device is designated to carry on with the discovery, the current node can decide to delegate the discovery (step 4b) or to increase its local knowledge base (step 4a) depending on its current resources. This delegation is analogous to the hierarchy delegation explained in step 3. Alternatively, if the node decides to increase its knowledge base, it asks the designated device for the information of its own knowledge base. Then it can proceed with this new information and continue with step 4. In Figure 1, the discovery is delegated to the server in between scopes a, b and c by means of message 2. Additionally, message 3 and its reply correspond to a knowledge base update so that the server is now aware of the services in scope c.

Some queries demand more than one service as candidate result or aggregated results. For this reason, delegation (both to the parent node or to the next designated device) includes the partially matching services found so far. In addition, step 4 is repeated until the query is completed.

Every node with enough resources keeps track of the processed queries and a cache of its results in its store. This serves to support queries with dynamic scope where previously solved discovery queries are reanswered or notified of new coming services that match them (step 1). In this way, results are kept fresh and updated with new nodes and services.

Once the device which originated the query is satisfied and it is no longer interested in matching services, then it emits a forget-query message to the chain of delegates so that the query can be removed from their stores.

4 Discovery in a Middleware for IoT

This section presents the implementation of the service discovery as part of the services provided by FamiWare [4], which is a family of middlewares for IoT, ambient intelligence and pervasive systems.
4.1 FamiWare Feature Model

Considering the high variety of the hardware and software that constitute the IoT and pervasive systems, instead of developing a single middleware, FamiWare is defined as a family of middleware for this kind of systems. FamiWare uses the Software Product Line (SPL) [10] approach in order to characterise the inherent variability of the IoT domain by a Feature Model (FM).

An FM specifies which elements or features of the family of products are common and which are variable. Thereby, the features are classified as mandatory, optional, alternative OR and alternative XOR. In addition, it is possible to specify formal cross-tree constraints or dependencies between the features. A valid FM configuration corresponds to a set of features of the FM that satisfies the tree constraints and the cross-tree constraints.

The FamiWare feature model represents the following information: (i) the characteristics of the devices supported by the middleware, (ii) the services (with their corresponding versions) provided by the middleware, and (iii) the different routing protocols implemented in the middleware that can be used for the communication between devices. Figure 2 depicts a partial feature model with the most representative features considered in this paper, including the sensor static characteristics (e.g., Temp, Humidity, Mobile, MoteLike, etc.) and the discovery service.

![FamiWare Partial Feature Model](image)

This feature model is the base of a model-driven process that derives a minimum configuration of the middleware adapted to the requirements of each network node. Then, FamiWare uses the feature model to customise the family and get middleware configurations at design-time. In the scope of this work, we propose to use this model also at run-time to discover the nodes by fulfilling the static characteristics of the contract. FamiWare makes use of Hydra tool for designing the FM and obtaining the configuration for every device automatically. In order to achieve these configurations, certain input parameters are given as initial constraints and Hydra generates the minimum valid configuration that satisfies those initial constraints.

4.2 Discovery in FamiWare

Once we have the FM configuration corresponding to all the devices of the network within the knowledge base, we can discover the nodes satisfying the query by providing to Hydra as input the query as initial constraints. In such a way, Hydra automatically returns a configuration containing those nodes that fulfill
the query. For this, we have to transform the query of our contract specified in XQuery as initial constraints, i.e., regular expressions. This process corresponds to the checking of the static characteristics.

**Example 2:** Going back to our example, the static characteristics of the query represented in XQuery are transformed in an initial constraint related to the discovery of mobile mote-like sensors with capabilities temperature and humidity. This initial constraint is an regular expression as follows:

\[ \text{Temp and Humidity and MoteLike and Mobile;} \]

Taking as input this initial constraint, Hydra returns all the nodes that contain these four features.

For the verification of dynamic characteristics, with the list of nodes provided by Hydra by fulfilling the static characteristics, we directly ask every node for its dynamic characteristics. To get this goal, monitoring services in FamiWare are executed in every node to give information about those characteristics. For instance, services monitoring the battery, the location, or the node traffic (see Figure 2).

**Example 3:** Specifically, in our scenario, the service monitoring the battery of every node given in the static phase should be queried in order to find out the first node fulfilling the required battery level, i.e., more than 50%, or the level closest to 50% if there is no sensor with the desired level.

As it was mentioned in Section 2.3 sometimes it will be require the reconfiguration of the sensor nodes in order to satisfy the contracts. FamiWare allows this reconfiguration process.

5 **Concluding Remarks**

We have presented a novel approach to tackle the discovery of devices or sensor nodes in the IoT domain by using contracts based on queries on a knowledge base. We take advantage of the feature model used in the FamiWare middleware as static information to represent the knowledge base. Our discovery process allows a negotiation at run-time by either relaxing the contracts or reconfiguring the devices of the system. In addition, discovery requests are remembered throughout the nodes involved in the process so that when new services appear, dynamic updates are sent to the interested node.

Our discovery approach has been inspired by current data diffusion and dynamic routing mechanisms in WSAN (e.g., [6, 7]) by extending and specialising them to serve for service discovery in Sensor Web. These protocols assume that the network overhead caused by control packages is neglectable in comparison to data packages. For this reason, data requests or advertisement messages are key in these approaches and actual data is only transferred when there is a real interest negotiated between a particular sink and source. Heidemann et al. [6] propose that, when a node requires certain information, it sends a query using message-flooding. Every node which receives that request, evaluates it against its provided information and, if there is a match, it floods that information as result for the request. The TinyDiffusion protocol [9] improves the idea in the following way. When the initial request is flooded, every node stores the first source where it receive the request from. In this way, whenever a node replies with the data, these heavy data messages are forwarded following a direct son-to-parent path, avoiding the data flooding used in the previous approach. In addition, requests are stored (with an expiration date) in every node so that new information that matches the request can be sent even after the request was issued. Compared to these works, we obtain a dynamic approach which is robust with regard to the ever-changing structure of the network; message-wise efficient, as it avoids any kind of broadcast or flooding; and hierarchy-aware, as it exploits both the hierarchy and the static self-information stored in the nodes.

As regards future work, we plan to further develop the ideas presented in this ongoing paper. In addition, we will evaluate the complexity of our approach, to formally demonstrate desirable properties (such as robustness) and illustrate the approach with real-world examples.
References


