Automatic Conflict Detection on Contracts*

Stephen Fenech[1], Gordon J. Pace[1], and Gerardo Schneider[3]

[1] Dept. of Computer Science, University of Malta, Malta
[2] Dept. of Informatics, University of Oslo, Norway
{sfen002,gordon.pace}@um.edu.mt, gerardo@ifi.uio.no

Abstract. Many software applications are based on collaborating, yet competing, agents or virtual organisations exchanging services. Contracts, expressing obligations, permissions and prohibitions of the different actors, can be used to protect the interests of the organisations engaged in such service exchange. However, the potentially dynamic composition of services with different contracts, and the combination of service contracts with local contracts can give rise to unexpected conflicts, exposing the need for automatic techniques for contract analysis. In this paper we look at automatic analysis techniques for contracts written in the contract language $\mathcal{CL}$. We present a trace semantics of $\mathcal{CL}$ suitable for conflict analysis, and a decision procedure for detecting conflicts (together with its proof of soundness, completeness and termination). We also discuss its implementation and look into the applications of the contract analysis approach we present. These techniques are applied to a small case study of an airline check-in desk.

1 Introduction

Today's trend towards Service-Oriented Architectures (SOA), in which different decoupled services distributed not only on different machines within a single organisation but also outside of it, provides new challenges to reliability and trust. Since an organisation may need to execute code provided by third parties, it requires mechanisms to protect itself — one such mechanism is the use of *contracts* giving restrictions on the service behaviours. Clearly, it is important that such contracts are conflict-free — meaning that the contracts will never lead to conflicting or contradictory directives.

Services are frequently composed of different sub-services, each of which comes with its own contract. The top-level service, not only needs to ensure that each single contract is conflict-free, but also that the composition of all the contracts is itself also conflict-free. This is true not only for SOA but for any application domain with a need to specify and monitor prescriptive behaviour.

The concept of contracts has been widely interpreted in the literature, from simple pre/post-conditions, to QoS properties. In this paper, we take the deontic view of contracts — a contract specifies the normative behaviour of a system, specifying obligations, permissions and prohibitions of actions, as well as the reparations in case of not respecting an obligation or prohibition. We build upon the contract language $\mathcal{CL}$ [10], which enables formal specification of deontic electronic contracts, and we extend

* Partially supported by the Nordunet3 project COSoDIS: "Contract-Oriented Software Development for Internet Services".

the trace semantics given in [6] in order to define and discover potential conflicts in contracts.

Although useful for runtime monitoring of $\mathcal{CL}$ contracts, the semantics given in [6] is not concerned with permissions, and it loses the deontic information (obligations, etc.) of the parties involved in the contract, making it unsuitable for conflict analysis. In this paper, we present an extension of this trace semantics to support conflict analysis, which is proved correct with respect to the original trace semantics. Based on the extended semantics we define the concept of conflicting contracts, and develop and prove the correctness of a decision procedure to detect conflicts in $\mathcal{CL}$ contracts. The algorithm has also been implemented into the tool CLAN for $\mathcal{CL}$ contract analysis.

The paper is organised as follows. We start by presenting $\mathcal{CL}$ in section 2, whose deontic trace semantics is introduced in section 3. The definition and algorithm for conflict analysis is then presented in section 4, where we also present theoretical results concerning correctness of the algorithm. Section 5 presents a small case study to illustrate the use of the analysis, which is compared to related work in section 6. We finally conclude in section 7.

2 The Contract Language $\mathcal{CL}$

Deontic logic [13] enables reasoning about non-normative and normative behaviour (e.g., obligations, permissions and prohibitions), including not only the ideal behaviours but also the exceptional and actual behaviours. One of the main problems of the logic is the difficulty theoreticians have to define a consistent yet expressive formal system, free from paradoxes [8].

Instead of trying to solve the problem of having a complete paradox-free deontic logic, $\mathcal{CL}$ has been designed with the aim to be used on a restricted application domain: electronic contracts. In this way the expressivity of the logic is reduced, resulting in a language free from most classical paradoxes, but still of practical use. $\mathcal{CL}$ is based on a combination of deontic, dynamic and temporal logics, allowing the representation of obligations, permissions and prohibitions, as well as temporal aspects. Moreover, it also gives a means to specify *exceptional* behaviours arising from the violation of obligations (what is to be demanded in case an obligation is not fulfilled) and of prohibitions (what is the penalty in case a prohibition is violated). These are usually known in the deontic community as *Contrary-to-Duties* (CTDs) and *Contrary-to-Prohibitions* (CTPs) respectively. $\mathcal{CL}$ contracts are written using the following syntax:

$$\begin{aligned}
C &:= C_O \mid C_P \mid C_F \mid C \wedge C \mid [\beta]C \mid \top \mid \bot \\
C_O &:= O_C(\alpha) \mid C_O \oplus C_O \\
C_P &:= P(\alpha) \mid C_P \oplus C_P \\
C_F &:= F_C(\alpha) \\
\alpha &:= 0 \mid 1 \mid \overline{a} \mid a \mid \alpha \,\&\, \alpha \mid \alpha ; \alpha \mid \alpha + \alpha \\
\beta &:= \epsilon \mid 0 \mid 1 \mid \overline{a} \mid a \mid \beta \,\&\, \beta \mid \beta ; \beta \mid \beta + \beta \mid \beta^*
\end{aligned}$$

Being $\mathcal{CL}$ an action-based language, we assume a non-empty set of actions $\Sigma = \{a, b, \ldots\}$, together with the three special actions 0, 1 and ϵ explained below. A contract clause C can be either an obligation (C_O), a permission (C_P) or a prohibition (C_F) clause, a conjunction of two clauses, the trivially satisfied contract ($\top$), the impossible contract ($\bot$) or a clause preceded by the dynamic logic square brackets. $O_C(\alpha)$ is interpreted as the obligation to perform α in which case, if violated, then the reparation contract C must be executed (a CTD). An obligation clause may be an exclusive disjunction of two other obligation clauses. This is interpreted as being obliged to satisfy one of the obligations but not both. $F_C(\alpha)$ is interpreted as forbidden to perform α and if α is performed then the reparation C must be executed (a CTP). In what follows we will write $F(\alpha)$ (respectively $O(\alpha)$) instead of $F_\bot(\alpha)$ (respectively $O_\bot(\alpha)$) to denote that there is no CTP (respectively CTD) associated. $[\beta]C$ is interpreted as if action β[3] is performed then the contract C must be executed — if β is not performed, the contract is trivially satisfied. The conjunction of two clauses is interpreted as both clauses have to be satisfied. The trivially satisfied contract $\top$ is satisfied by any sequence of actions whereas the impossible contract $\bot$ cannot be satisfied with any sequence of actions. ϵ is an empty action, 1 is the action that matches any action, while 0 is the impossible action.

Action expressions can be constructed from basic ones using the operators $\&$, $;$, $+$ and * where $\&$ stands for the actions occurring concurrently, ; stands for the actions to occur in sequence, $+$ stands for a choice between actions and * is the Kleene star. $\bar{\cdot}$ is the complement, so $\overline{a}$ means "any action except a". In the rest of the paper $\alpha_\&$ will denote basic actions or complex actions constructed from basic actions only using the concurrent operator $\&$ (for example a, $a\&b$). It can be shown that every action expression can be transformed into an equivalent representation where $\&$ appears only at the innermost level. This representation is referred to as the *canonical form*. In the rest of this paper we assume that action expressions have been reduced to this form. We also allow the negation of compound actions in the formal syntax (and semantics). However, one can push negations on action expressions down to the constituent actions. Throughout the rest of the paper, whenever the negation of action expressions is used, it is assumed that the expression will be reduced appropriately. Following [10], we assume there is an action dictionary containing all possible actions, including which actions are contradictory: we write $a\#b$ to denote that a and b are contradictory (for instance "send a message shorter than 5 characters" and "send a message longer than 10 characters").

In order to avoid paradoxes the operators combining obligations, permissions and prohibitions are restricted syntactically. See [10, 6] for more details on $\mathcal{CL}$.

As a simple example, let us consider the following clause from an airline company contract: 'When checking in, the traveller is obliged to have a luggage within the weight limit — if exceeded, the traveller is obliged to pay extra.' This would be represented in $\mathcal{CL}$ as $[\mathit{checkIn}]O_{O(\mathit{pay})}(\mathit{withinWeightLimit})$.

[3] Note that the only differences between the syntactic categories representing actions (α and β) is $\cdot^*$. α is restricted to be used only under obligations, permissions and prohibitions, while β only in "conditions".

3 Deontic Trace Semantics

In this section, we will introduce a new finite trace semantics of $\mathcal{CL}$ that includes deontic information — which obligations, permissions and prohibitions are enacted at each step of the trace. This will enable us to detect conflicts in a contract, by looking at finite traces allowed by the semantics leading to incompatible normative behaviour — for example both obliging and forbidding the same action at the same time.

Let us consider a simple example to better understand the need of a finite trace semantics with deontic information. Let $C = [a]O(b) \wedge [b]F(b)$ be a contract on the action alphabet $\{a, b\}$ we want to check for conflicts. According to the $\mathcal{CL}$ (infinite) trace semantics given in [6], the set of traces "accepted" by the contract C is $\{\langle a, b, any\rangle \mid any = (a+b)^\omega\} \cup \{\langle b, a, any\rangle \mid any = (a+b)^\omega\}$. According to the semantics, no trace starting with action $\{a, b\}$ (i.e., with a and b occurring concurrently) will be accepted by the contract, since this would imply a contract violation due to the enacted conflicting obligation and prohibition. Moreover, there is no deontic information in the trace, making it difficult to capture the notion of conflict. Since our aim is to obtain a witness of such a conflict, and in particular a systematic way to obtain an automaton that recognises such prefixes containing conflicts, it is necessary to extend the trace semantics. This extension includes: (1) The addition of deontic information (which obligations, permissions and prohibitions are satisfied at any moment), (2) The addition of a trace semantics for permission (this was not present in the original trace semantics), (3) The addition of the possibility to "accept" certain finite prefixes (in order to get the witness for conflicts). With this new semantics we will be able to automatically obtain an automaton accepting exactly the (finite prefix) traces "accepted" by the contract, including those witnesses for conflict detection.

For a contract with action alphabet Σ, we will introduce its deontic alphabet Σ_d which consists of O_a, P_a and F_a for each action $a \in \Sigma$, that will be used to represent which normative behaviour is enacted at a particular moment. Given a set of concurrent actions α, we will write O_α to represent $\{O_a \mid a \in \alpha\}$.

Given a $\mathcal{CL}$ contract C with action alphabet Σ, the semantics will be expressed in the form $\sigma, \sigma_d \vDash C$, where σ is a finite trace of sets of concurrent actions in Σ and σ_d is a finite trace consisting on sets of sets[4] of deontic information in Σ_d. The statement $\sigma, \sigma_d \vDash C$ is said to be well-formed if *length*(σ) = *length*(σ_d). In the rest of the paper we will consider only well-formed semantic statements.

A well-formed statement $\sigma, \sigma_d \vDash C$ will correspond to the statement that action sequence σ is possible under (will not break) contract C, with σ_d being the deontic statements enforced from the contract.

Let us consider again the contract $C = [a]O(b) \wedge [b]F(b)$, and the trace $\sigma = \langle\{a\}, \{b\}\rangle$, then $\sigma_d = \langle\{\emptyset\}, \{\{O_b\}\}\rangle$, and we have that $\sigma, \sigma_d \vDash C$. The contract $C' = F(c) \wedge [1](O(a) \wedge F(b))$, for example, stipulates that it is forbiden to perform action c and that after the execution of any action, there is an obligation to perform an a (while prohibiting the execution of b), so we can write $\sigma_d = \langle\{\{F_c\}\}, \{\{O_a\}, \{F_b\}\}\rangle$. The contract allows the execution of actions a and b concurrently, and then a concurrently with c ($\sigma = \langle\{a, b\}, \{a, c\}\rangle$), and we have that $\sigma, \sigma_d \vDash C'$. As a final example, let

[4] This is needed to distinguish choices from conjunction.

$$\sigma, \sigma_d \vDash C \quad \text{if} \quad \mathit{length}(\sigma) = \mathit{length}(\sigma_d) = 0 \tag{1}$$
$$\sigma, \sigma_d \vDash \top \quad \text{if} \quad \sigma_d(0) = \emptyset \text{ and } \sigma(1..), \sigma_d(1..) \vDash \top \tag{2}$$
$$\sigma, \sigma_d \vDash C_1 \wedge C_2 \quad \text{if} \quad \sigma, \sigma'_d \vDash C_1 \text{ and } \sigma, \sigma''_d \vDash C_2 \text{ and } \sigma_d = \sigma'_d \cup \sigma''_d \tag{3}$$
$$\sigma, \sigma_d \vDash C_1 \oplus C_2 \quad \text{if} \quad (\sigma, \sigma_d \vDash C_1 \text{ and } \sigma, \sigma_d \nvDash C_2) \text{ or } (\sigma, \sigma_d \vDash C_2 \text{ and } \sigma, \sigma_d \nvDash C_1) \tag{4}$$
$$\sigma, \sigma_d \vDash [\epsilon]C \quad \text{if} \quad \sigma, \sigma_d \vDash C \tag{5}$$
$$\sigma, \sigma_d \vDash [\alpha_\&]C \quad \text{if} \quad (\alpha_\& \not\subseteq \sigma(0) \Rightarrow \sigma, \sigma_d \vDash \top) \text{ and} \tag{6}$$
$$(\alpha_\& \subseteq \sigma(0) \Rightarrow (\sigma_d(0) = \emptyset \text{ and } \sigma(1..), \sigma_d(1..) \vDash C)) \tag{7}$$
$$\sigma, \sigma_d \vDash [\overline{\alpha_\&}]C \quad \text{if} \quad (\alpha_\& \subseteq \sigma(0) \Rightarrow \sigma, \sigma_d \vDash \top) \text{ and} \tag{8}$$
$$(\alpha_\& \not\subseteq \sigma(0) \Rightarrow (\sigma_d(0) = \emptyset \text{ and } \sigma(1..), \sigma_d(1..) \vDash C)) \tag{9}$$
$$\sigma, \sigma_d \vDash [\beta; \beta']C \quad \text{if} \quad \sigma, \sigma_d \vDash [\beta][\beta']C \tag{10}$$
$$\sigma, \sigma_d \vDash [\beta + \beta']C \quad \text{if} \quad \sigma, \sigma_d \vDash [\beta]C \wedge [\beta']C \tag{11}$$
$$\sigma, \sigma_d \vDash [\beta^*]C \quad \text{if} \quad \sigma, \sigma_d \vDash C \wedge [\beta][\beta^*]C \tag{12}$$
$$\sigma, \sigma_d \vDash O_C(\alpha_\&) \quad \text{if} \quad \sigma_d(0) = O_{\alpha_\&} \text{ and} \tag{13}$$
$$(\alpha_\& \subseteq \sigma(0) \Rightarrow \sigma(1..), \sigma_d(1..) \vDash \top) \text{ and} \tag{14}$$
$$(\alpha_\& \not\subseteq \sigma(0) \Rightarrow \sigma(1..), \sigma_d(1..) \vDash C) \tag{15}$$
$$\sigma, \sigma_d \vDash O_C(\alpha; \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash O_C(\alpha) \wedge [\alpha]O_C(\alpha') \tag{16}$$
$$\sigma, \sigma_d \vDash O_C(\alpha + \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash O_\top(\alpha) \wedge O_\top(\alpha') \wedge [\overline{\alpha + \alpha'}]C \tag{17}$$
$$\sigma, \sigma_d \vDash F_C(\alpha_\&) \quad \text{if} \quad \sigma_d(0) = F_{\alpha_\&} \text{ and} \tag{18}$$
$$(\alpha_\& \subseteq \sigma(0) \Rightarrow \sigma(1..), \sigma_d(1..) \vDash C) \text{ and} \tag{19}$$
$$(\alpha_\& \not\subseteq \sigma(0) \Rightarrow \sigma(1..), \sigma_d(1..) \vDash \top) \tag{20}$$
$$\sigma, \sigma_d \vDash F_C(\alpha; \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash F_\bot(\alpha) \text{ or } \sigma, \sigma_d \vDash [\alpha]F_C(\alpha') \tag{21}$$
$$\sigma, \sigma_d \vDash F_C(\alpha + \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash F_C(\alpha) \wedge F_C(\alpha') \tag{22}$$
$$\sigma, \sigma_d \vDash P(\alpha_\&) \quad \text{if} \quad \sigma_d(0) = P_{\alpha_\&} \text{ and } \sigma(1..), \sigma_d(1..) \vDash \top \tag{23}$$
$$\sigma, \sigma_d \vDash P(\alpha; \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash P(\alpha) \wedge [\alpha]P(\alpha') \tag{24}$$
$$\sigma, \sigma_d \vDash P(\alpha + \alpha') \quad \text{if} \quad \sigma, \sigma_d \vDash P(\alpha) \wedge P(\alpha') \tag{25}$$

Fig. 1. The deontic trace semantics of $\mathcal{CL}$

us consider the contract $C'' = [a]O(b{+}c) \wedge [b]F(b)$. In this case, due to the choice inside the obligation, we get that given the trace $\sigma = \langle\{a\}, \{b\}\rangle$ then $\sigma_d = \langle\{\emptyset\}, \{\{O_b, O_c\}\}\rangle$, and we have that $\sigma, \sigma_d \vDash C''$.

Given two traces σ_1 and σ_2, we will use $\sigma_1; \sigma_2$ to denote their concatenation, and $\sigma_1 \cup \sigma_2$ (provided the length of σ_1 is equal to that of σ_2) to denote the point-wise union of the traces: $\langle \sigma_1(0) \cup \sigma_2(0),\ \sigma_1(1) \cup \sigma_2(1),\ \ldots \sigma_1(n) \cup \sigma_2(n)\rangle$. In what follows we explain our new trace semantics, shown in Fig. 1.[5]

Basic conditions: Empty traces satisfy any contract, as shown in Fig. 1-(1).

[5] Due to lack of space, we do not present the trivial cases of actions 0 and 1, and they are omitted in the rest of the paper.

Done, Break: The simplest definitions are those of the trivially satisfiable contract $\top$, and the unsatisfiable contract $\bot$. In the case of $\bot$, only an empty sequence will not have yet broken the contract, while in the case of $\top$, any sequence of actions satisfies the contract (whenever no obligation, prohibition, or permission is present on the trace). See Fig. 1 line (2).

Conjunctions: For the conjunction of two contracts, the action trace must satisfy both contracts, and the deontic traces are combined point-wise. See Fig. 1 line (3).

Exclusive disjunction: Similar to conjunctions. See Fig. 1 line (4). (Note that the rule is valid only for C_1 and C_2 being both of the form C_O, or C_P. In the rest of the paper we will continue to write $C_1 \oplus C_2$ with the understanding that the above restriction applies.)

Conditions: Conditions are handled structurally. Note that using the normal form defined in [6], one can push concurrent actions to the bottom level. See Fig. 1 lines (5)–(12).

Obligations: Obligations, like conditions, are defined structurally on action expressions. The base case of the action simply consisting of a conjunction of actions that can be dealt with by ensuring that if the actions are present in the action trace, then the contract is satisfied, otherwise the reparation is enacted. The case for the sequential composition of two action sequences is handled simply by rewriting into a pair of obligations. The case of choice (+) is the most complex case, in which we have to consider the possibility of having either obligation satisfied or neither satisfied, hence triggering the reparation. Recall that the star operator cannot appear within obligations. See Fig. 1 lines (13)–(17).

Prohibitions: Dealing with prohibitions is similar to obligations, with the main difference being that prohibition of choice is more straightforward to express. See Fig. 1 lines (18)–(22).

Permissions: The aim of the original trace semantics of $\mathcal{CL}$ [6] was to provide a linear time semantics to the language, appropriate for applications such as runtime verification. Since a single linear trace does not give any information whether a permission clause has been found to be in conflict with other clauses or not, the original semantics simply discarded permission clauses. However, to reason about conflicts, the fact that a permission operator has been enacted is important. See Fig. 1 lines (23)–(25) for the semantics.

4 Conflict Analysis

Conflicts in contracts arise from four different reasons. The first two reasons are being obliged and forbidden to perform the same action (e.g., $O(a) \wedge F(a)$), and being permitted and forbidden to perform the same action (e.g., $P(a) \wedge F(a)$). In the first conflict we would end up in a situation where whatever is performed will violate the contract. The second conflict would not result in having a trace that violates the contract since in the trace semantics permissions cannot be broken, however, we can still identify these situations due to the deontic trace. The remaining two kinds of conflicts correspond to obligations of contradictory actions (e.g., $O(a) \wedge O(b)$ with $a \# b$), and permissions and obligations of contradictory actions (e.g., $P(a) \wedge O(b)$ with $a \# b$).

Before defining formally what a conflict-free contract is, we recall our motivating example, the contract $[a]O(b) \wedge [b]F(b)$ with allowed actions a and b. It is clear that both traces $\sigma_1 = \langle\{a\},\{b\}\rangle$ and $\sigma_2 = \langle\{b\},\{a\}\rangle$ satisfy the contract. However, any trace starting with concurrent actions $\{a,b\}$ (e.g., $\langle\{a,b\},\{b\}\rangle$) will not be accepted by the contract since any action following it will violate either the obligation to perform b or the prohibition from performing b. In this case, since unspecified, the reparation is the $\perp$ clause which cannot be satisfied regardless of what action is performed.

In what follows we define the notion of conflict-free contract at the semantic level, formalising the four cases. We show how to obtain an automaton from a contract and discuss an automata-based model checking algorithm for detecting conflicts.

Definition 1. *For a given trace σ_d of a contract C, let $D, D' \subseteq \sigma_d(i)$ (with $i \geq 0$). We say that D is* in conflict with *D' if and only if there exists at least one element $e \in D$ such that:*

$$\begin{aligned} & e = O_a \wedge (F_a \in D' \vee (P_b \in D' \wedge a\#b) \vee (O_b \in D' \wedge a\#b)) \\ \textit{or}\ \ & e = P_a \wedge (F_a \in D' \vee (P_b \in D' \wedge a\#b) \vee (O_b \in D' \wedge a\#b)) \\ \textit{or}\ \ & e = F_a \wedge (P_a \in D' \vee O_a \in D'). \end{aligned}$$

A contract C is said to be conflict-free *if for all traces σ and σ_d such that $\sigma, \sigma_d \vDash C$, then for any $D, D' \subseteq \sigma_d(i)$ $(0 \leq i \leq len(\sigma_d))$, D and D' are not in conflict.*

Let us consider the contract $C = [a]O(b+c) \wedge [b]F(b)$, then we have that C is not conflict-free since $\langle\{a,b\},\{b\}\rangle, \langle\{\emptyset\},\{\{O_b,O_c\},\{F_b\}\}\rangle \vDash C$, and there are $D, D' \subseteq \sigma_d(1)$ such that D and D' are in conflict. To see this, let us take $D = \{O_b, O_c\}$ and $e = O_b$. We have then that for $D' = \{F_b\}$, $F_b \in D'$ (satisfying the first line of definition 1).

We have then characterised the notion of conflict in contracts by analysing the set of traces accepted by the contract. We now show how to generate a finite-state automaton from a $\mathcal{CL}$ contract C, with the property that the language accepted by the automaton corresponds to the traces given by the semantics of the contract. We also define the notion of conflict in the generated automaton.

Generation of an automaton from a $\mathcal{CL}$ contract Given a contract C, over an action alphabet Σ and corresponding deontic alphabet Σ_d, we can construct an automaton $A(C) = \langle S, A_\&, s_0, T, V, l, \delta\rangle$ where S is the set of states, $A_\&$ is the set of concurrent actions from Σ, s_0 is the initial state, $T \subseteq S \times A_\& \times S$ is the set of labelled transitions, V is a special violation state, l is a function labelling states with the $\mathcal{CL}$ clause that holds in that state ($l : S \to \mathcal{CL}$) and $\delta : S \to 2^{\Sigma_d}$ is a function labelling states with the set of deontic notions that hold in that state. We say that a *run* (sequence of states) is accepted by the automaton if none of the states of the run is V. Similarly, we say that the automaton *accepts a word* w, consisting of a sequence of actions, if none of the actions of w is the label of a transition containing the state V, in which case we write $\texttt{Accept}(A(C), w)$. Note that the automaton is deterministic.

The construction of the automaton uses the residual contract function f which, given a $\mathcal{CL}$ formula C and an action α, will return the clause that needs to hold in the following step, similarly to the CTL sub-formula construction [2]. f is defined in Fig. 2. The

$$
\begin{aligned}
f &: \mathcal{CL} \times A_{\&} \to \mathcal{CL} \\
f(\top, \varphi) &= \top \\
f(\bot, \varphi) &= \bot \\
f(C_1 \wedge C_2, \varphi) &= f(C_1, \varphi) \wedge f(C_2, \varphi) \\
f(C_1 \oplus C_2, \varphi) &= \begin{cases} \top & \text{if } (f(C_1,\varphi) = \top \wedge f(C_2,\varphi) = \bot) \vee \\ & \quad (f(C_1,\varphi) = \bot \wedge f(C_2,\varphi) = \top) \\ \bot & \text{if } (f(C_1,\varphi) = f(C_2,\varphi) = \top) \vee \\ & \quad (f(C_1,\varphi) = f(C_2,\varphi) = \bot) \\ f(C_1,\varphi) \oplus f(C_2,\varphi) & \text{otherwise} \end{cases} \\
f([\alpha_{\&}]C, \varphi) &= \begin{cases} C & \text{if } \alpha_{\&} \subseteq \varphi \\ \top & \text{otherwise} \end{cases} \\
f([\overline{\alpha_{\&}}]C, \varphi) &= \begin{cases} C & \text{if } \alpha_{\&} \not\subseteq \varphi \\ \top & \text{otherwise} \end{cases} \\
f([\overline{\alpha;\alpha'}]C, \varphi) &= \begin{cases} C & \text{if } (\alpha;\alpha')/\varphi = 0 \\ [\overline{(\alpha;\alpha')/\varphi}]C & \text{otherwise} \end{cases} \\
f([\overline{\alpha+\alpha'}]C, \varphi) &= f([\overline{\alpha}]C, \varphi) \wedge f([\overline{\alpha'}]C, \varphi) \\
f([\beta;\beta']C, \varphi) &= f([\beta][\beta']C, \varphi) \\
f([\beta+\beta']C, \varphi) &= f([\beta]C \wedge [\beta']C, \varphi) \\
f([\beta^*]C, \varphi) &= f(C \wedge [\beta][\beta^*]C, \varphi) \\
f(O_C(\alpha_{\&}), \varphi) &= \begin{cases} \top & \text{if } \alpha_{\&} \subseteq \varphi \\ C & \text{otherwise} \end{cases} \\
f(O_C(\alpha;\alpha'), \varphi) &= f(O_C(\alpha) \wedge [\alpha]O_C(\alpha'), \varphi) \\
f(O_C(\alpha+\alpha'), \varphi) &= \begin{cases} \top & \text{if } f(O_\bot(\alpha),\varphi) = \top \text{ or } f(O_\bot(\alpha'),\varphi) = \top \\ C & \text{if } f(O_\bot(\alpha),\varphi) = \bot \text{ and } f(O_\bot(\alpha'),\varphi) = \bot \\ O_C(\alpha+\alpha'/\varphi) & \text{otherwise} \end{cases} \\
f(F_C(\alpha_{\&}), \varphi) &= \begin{cases} C & \text{if } \alpha_{\&} \subseteq \varphi \\ \top & \text{otherwise} \end{cases} \\
f(F_C(\alpha;\alpha'), \varphi) &= f([\alpha]F_C(\alpha'), \varphi) \\
f(F_C(\alpha+\alpha'), \varphi) &= f(F_C(\alpha) \wedge F_C(\alpha')) \\
f(P(\alpha_{\&}), \varphi) &= \top \\
f(P(\alpha \cdot \alpha'), \varphi) &= f(P(\alpha) \wedge [\alpha]P(\alpha'), \varphi) \\
f(P(\alpha+\alpha'), \varphi) &= f(P(\alpha) \wedge P(\alpha'), \varphi)
\end{aligned}
$$

Fig. 2. The residual function f

binary operator $/$ used in f, that gives the tail of the left-hand side sequence of actions if its head matches the right-hand side action, is defined inductively as follows:

$$
\begin{aligned}
\alpha'_{\&}/\alpha_{\&} &= \epsilon \text{ if } \alpha'_{\&} \subseteq \alpha_{\&}, \text{ otherwise } 0 \\
(0;\alpha)/\alpha_{\&} &= 0 \\
(1;\alpha)/\alpha_{\&} &= \alpha \\
(\alpha;\alpha')/\alpha_{\&} &= (\alpha/\alpha_{\&});\alpha' \\
(\alpha+\alpha')/\alpha_{\&} &= \alpha/\alpha_{\&} + \alpha'/\alpha_{\&}
\end{aligned}
$$

For example, $(a;b)/a$ will give b whereas $((a;b)+(a;c))/a$ will result in $b+c$.

The automaton is built using the construction function f_c shown in Fig. 3, that takes as argument an initial state s_0 where $l(s_0) = C$. Besides the residual function f, f_c

```
$f_c(s) =$ if $l(s) = 1$ then
             $T := T \cup (s, 1, s)$
           if $l(s) = 0$ then
             $V := s$
             $T := T \cup (V, 1, V)$
           otherwise   $\forall a \in A_{\&}$
              if $\exists\, s' \in S$ s.t. $l(s') = f(l(s), a)$
              then $T := T \cup (s, a, s')$
              otherwise
                new $s'$
                $l(s') := f(l(s), a)$
                $S := S \cup s'$
                $T := T \cup (s, a, s')$
                $d(s') := f_d(l(s'))$
                $f_c(s')$
```

Fig. 3. The construction function f_c

$$
\begin{aligned}
f_d(C_1 \wedge C_2) &= f_d(C_1) \cup f_d(C_2)\\
f_d(O(\alpha_{\&})) &= \{\{O_{a_1}\}, \ldots, \{O_{a_n}\}\}\\
f_d(F(\alpha_{\&})) &= \{\{F_{a_1}\}, \ldots, \{F_{a_n}\}\}\\
f_d(P(\alpha_{\&})) &= \{\{P_{a_1}\}, \ldots, \{P_{a_n}\}\}\\
f_d(O(\alpha + \alpha')) &= \{x \cup y \mid x \in f_d(O(\alpha))\\
&\qquad \text{and } y \in f_d(O(\alpha'))\}\\
f_d(\text{ otherwise }) &= \emptyset
\end{aligned}
$$

Fig. 4. The deontic labelling function f_d

uses function f_d (shown in Fig. 4) that adds all the relevant deontic information to each state (we take $\alpha_{\&}$ to be equal to $a_1 \& \ldots \& a_n$).[6]

As an example, let us consider the contract $[a]O(b) \wedge [b]F(b)$. The automaton is constructed by applying f_c to the state s_0 where $l(s_0) = [a]O(b) \wedge [b]F(b)$. Every possible transition is created (in this case, transitions labelled with a, b and $a\&b$) from this state to a new state labelled with the result of applying function f to the original formula and the label of the transition as parameters. Thus, the state that is reached with the transition labelled with action a is $f([a]O(b) \wedge [b]F(b), a) = O(b)$. If there is another state with the same label, the transition will connect to the existing state and the new one will be discarded (this ensures termination). If there is no such a state, f_c is then recursively called on this new state. Eventually we either reach a satisfying state, a violating state, or a state already labeled with the formula. The corresponding automaton is shown in Fig. 5.[7]

Since our objective is to find conflicts analysing the constructed automaton, we need to define what a conflict is at the automaton level. The definition is straightforward and it is very similar to the definition given for $\mathcal{CL}$ traces.

Definition 2. *Given a state s of an automaton $A(C)$, let $D, D' \subseteq f_d(s)$. We say that D is* in conflict with *D' if and only if there exists at least one element e of D such that:*

[6] We have omitted the case for $\oplus$ in the deontic labelling function description. In practice, two different automata are created for each one of the choices, and the analysis proceeds as usual. Also note that there is no explicit labelling function for $F(\alpha + \alpha')$ and $P(\alpha + \alpha')$ since these cases are reduced to conjunction.

[7] Note that what is written in each state is the *sub-formula* remaining to be satisfied. Formally speaking, each state will be "marked" with the deontic information as defined by the function f_d. So, $O(a)$ is a syntactic expression in $\mathcal{CL}$, while O_a is the corresponding "marking" at the state saying there is an obligation of doing a.

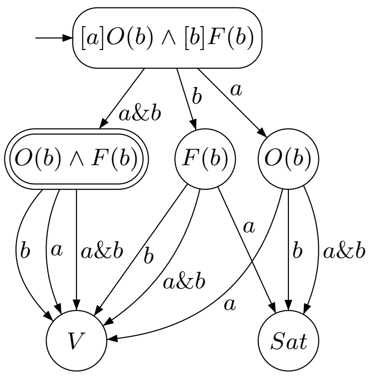

Fig. 5. Automaton for $[a]O(b) \wedge [b]F(b)$

$$\begin{aligned} & e = O_a \wedge (F_a \in D' \vee (P_b \in D' \wedge a\#b) \vee (O_b \in D' \wedge a\#b)) \\ \mathit{or}\ \ & e = P_a \wedge (F_a \in D' \vee (P_b \in D' \wedge a\#b) \vee (O_b \in D' \wedge a\#b)) \\ \mathit{or}\ \ & e = F_a \wedge (P_a \in D' \vee O_a \in D'). \end{aligned}$$

An automaton $A(C)$ is said to be conflict-free *if for every state $s \in S$, then for any $D, D' \subseteq f_d(s)$, D and D' are not in conflict.*

The automaton shown in Fig. 5 is not conflict-free since there exists a state which is not conflict-free. Consider that s is the double-lined state labelled with $O(b) \wedge F(b)$ then $f_d(s) = \{\{O_b\}, \{F_b\}\}$. Using definition 2, let $e = O_b$. For this state to be conflict-free, any subset $D \in f_s(s)$ should not contain F_b, which is not the case.

Conflict detection algorithm The main algorithm takes a contract written in $\mathcal{CL}$ and decides whether or not the given contract may reach a state of conflict. Once the automaton was generated from the contract as explained above, the conflict detection algorithm simply consists of a standard forward or backward reachability analysis based on a fix-point computation, looking for states containing conflicts.

For example, performing reachability analysis on the simple contract whose automaton is shown in Fig. 5 would identify that the conflict state labelled $O(b) \wedge F(b)$ is reachable from the initial state upon receiving action $a\&b$, since the state contains the deontic information $\{\{O_b\}, \{F_b\}\}$.

Correctness of the algorithm We now prove the correctness and completeness of the algorithm, which includes proving the following auxiliary results: (1) The traces accepted by the automaton coincide with those "accepted" by the contract in $\mathcal{CL}$ (according to the trace semantics); (2) A contract C in $\mathcal{CL}$ is conflict-free iff the generated automaton $A(C)$ is conflict-free.

We first prove that the automaton will accept all and only those traces which satisfy the contract.

Lemma 1. *Given a $\mathcal{CL}$ contract C, the automaton $A(C)$ accepts all and only those traces σ that satisfy the contract: $\sigma, \sigma_d \vDash C$ if and only if $\texttt{Accept}(A(C), \sigma)$.*

The proof is based on a long and tedious induction on the structure of the formula, proving that f_c (and the auxiliary functions f and f_d) are complete and correct.

Note that our algorithm checks that no state contains a conflict rather than checking all possible satisfying runs. In order to prove that this is correct we need to prove that we generate only and all the reachable states.

Proposition 1. *The function f_c generates all and only reachable states.*

Based on the above proposition and the definition of conflict at the trace and the automaton level, we can prove that the automata construction function preserves conflict-freedom, and that no spurious conflicts are generated.

Lemma 2. *A contract C written in $\mathcal{CL}$ is conflict-free if and only if the automaton $A(C)$ is conflict-free.*

Based on the above results, and the correctness and completeness proofs of standard forward reachability analysis, we can finally prove our main result. Termination is trivially guaranteed since the generated automaton is finite and the reachability analysis is based on a standard fix-point computation.

Theorem 1. *The $\mathcal{CL}$ conflict detection algorithm is correct and complete.*

5 Case Study

In this section, the use of conflict analysis will be illustrated through a small case study, starting from a draft contract written in English, translated in $\mathcal{CL}$ and analysed using the techniques developed in this paper.

Consider a contract between an airline company and a company taking care of the ground crew (mainly the check-in process), where the normative specification is given as the following *contract*:

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.*

The contract can be represented in $\mathcal{CL}$ as shown below. Note that the last clause is introduced as a reparation for breaking the previous clauses.[8] Also, all the natural language clauses include an implicit universal quantification — statements of the form 'After the check-in desk is open…' should be interpreted as 'At any time, after the check-in desk is open…'. Hence, $[1^*]$ precedes such clauses.

Note that the last clause corresponds to the penalty (reparation) of all the obligations and prohibitions appearing in the contract, and thus they are represented as CTDs and CTPs with the secondary obligation $O(\mathit{fine})$.

1. $[1^*][\mathit{2hBefore}]O_{O(\mathit{fine})}(\mathit{openCheckIn}\ \&\ \mathit{requestInfo})$
2. $[1^*][\mathit{openCheckIn}\&\mathit{requestInfo}]O_{O(\mathit{fine})}(\mathit{replyInfo})$
3. $[1^*][\mathit{openCheckIn}][1^*](O(\mathit{correctDetails}\ \&\ \mathit{luggageInLimit}) \wedge$
   $[\mathit{correctDetails}\ \&\ \mathit{luggageInLimit}]O_{O(\mathit{fine})}(\mathit{boardingCard}))$
4. $[1^*][\mathit{openCheckIn}][1^*][\mathit{correctDetails}\ \&\ \mathit{luggageOverLimit}]$
   $O_{O(\mathit{fine})}(\mathit{collectPayment}\&\mathit{boardingCard})$
5. $[1^*][\overline{\mathit{correctDetails}}]F_{O(\mathit{fine})}(\mathit{boardingCard})$
6. $[\overline{\mathit{openCheckIn}}^*]F_{O(\mathit{fine})}(\mathit{boardingCard})$
7. $([1^*][\mathit{20mBefore}]O_{O(\mathit{fine})}(\mathit{closeCheckIn})) \wedge ([\overline{\mathit{20mBefore}}^*]F_{O(\mathit{fine})}(\mathit{closeCheckIn}))$
8. $[1^*][\mathit{closeCheckIn}]O_{O(\mathit{fine})}(\mathit{sendLuggageInfo})$
9. $[1^*][\mathit{closeCheckIn}][1^*](F_{O(\mathit{fine})}(\mathit{openCheckIn}) \wedge F_{O(\mathit{fine})}(\mathit{boardingCard}))$

Running the contract through the conflict discovery algorithm, we discover a number of problems. The first conflict we encounter is being obliged and forbidden to issue a boarding pass.

The tool will identify a state in conflict labelled with the obligation to perform action *boardingCard* and the prohibition of performing action *boardingCard* together with a trace leading to this state. Looking at clause 3, once the crew opens the check-in desk, they are always obliged to issue a boarding pass if the client has the correct details. However, according to clause 9 it is prohibited to issue of boarding pass once the check-in desk is closed. These two clauses are in conflict once the check-in desk is closed and a client arrives to the desk with the correct details. To fix this problem we require to change clause 3 so that after the check-in desk is opened, the ground crew is obliged to issue the boarding pass as long as the desk has not been closed. This issue can also be found in clause 4 and the solution is similar.

The trace returned identifies the situation in which the check-in desk is closed at the same time the client provides his correct details:
$\langle \mathit{openCheckIn},\ \mathit{closeCheckIn}\ \&\ \mathit{correctDetails},\ O(\mathit{boardingCard})\ \&\ F(\mathit{boardingCard})\rangle$.

[8] Note that the payment is supposed to be immediate.

In reality, a check-in desk cannot close and accept the passport details at the same time, and thus these two are mutually exclusive actions. Adding these two actions as mutually exclusive will solve this conflict.

To ensure that *2hBefore* and *20mBefore* occur in the correct order, we make use of path constraints. Similar constraints are used for *openCheckIn* and *closeCheckIn*. Thus, clauses number 3 and 4 have to be modified as follows:

$3'$. $[1^*][\mathit{openCheckIn}][\overline{\mathit{closeCheckIn}}^*][\mathit{correctDetails} \,\&\, \mathit{luggageInLimit}]O_{O(\mathit{fine})}(\mathit{boardingCard})$

$4'$. $[1^*][\mathit{openCheckIn}][\overline{\mathit{closeCheckIn}}^*][\mathit{correctDetails} \,\&\, \mathit{luggageOverLimit}]$
$$O_{O(\mathit{fine})}(\mathit{collectPayment}\&\mathit{boardingCard})$$

This could be represented in textual form as:

$3'$. *After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present until the check-in desk is closed*[9]*. This is done by checking that passport details match the ticket and that luggage is within the weight limits. Then the crew is 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.*

Note that $4'$ is stated in the same way as in the original contract since what have changed are the common conditions stated in $3'$. From this small case study, it should be evident that the resolution of conflicts in a contract require human intervention, to ensure that the amendments to the contract correspond the what one had in mind in the first place. Although one could define automated ways of changing, removing or adding clauses to resolve conflicts, the sheer number of possibilities one has (making certain actions mutually exclusive, removing parts of a contract, delaying the triggering of a contract, etc) and the fact that most of the options would not make sense in the real-world interpretation of the contract makes automated conflict resolution impractical.

6 Related Work

The use of model checking techniques for logics other than temporal logic is quite new, and it focuses mainly in multi-agent systems (see for instance [12]). There is not much work on the verification of logics containing the deontic notions of obligation, permission and prohibition, and including CTDs and CTPs. An extended temporal logic with conditional obligations and permissions is presented in [4] for checking whether an organisation conforms to a body of regulation. In the context of SOA, model checkers have recently been used to verify compliance of web-service composition [7], where the specifications are given in the so-called temporal deontic interpreted systems. However, we are not aware of any work that automatically detects conflicts in deontic contracts as presented here.

[9] Recall that we made *closeCheckIn* and *correctDetails* mutually exclusive, and cannot thus happen at the same instance of time. This ensures the submission of the correct details before the desk is closed.

The trace semantics used in our paper extends the one introduced for monitoring purposes in [6]. The automaton they generate is different and cannot be used for conflict analysis since it does not consider permissions, and does not keep deontic information in the states, determining only if a trace has been satisfied, violated or neither. Moreover, we can create a monitor directly from the automaton generated thus enabling both monitoring and conflict analysis.

In [9], a labelled transition system is generated in an *ad hoc* manner from a $\mathcal{CL}$ contract in order to be model checked using nuSMV, against properties expressed in LTL. The process is subject to error since many of the steps are manual, and the encoding of the deontic information into nuSMV is complicated. Our method is completely automatic, and though it is specific for conflict analysis it could be extended for other uses as we explain in the next section.

7 Conclusions

We have presented a finite trace semantics for $\mathcal{CL}$ augmented with deontic information, and showed its use for automatic contract analysis for conflict discovery. Remarkably, we do not use $\mathcal{CL}$ branching semantics [11] for conflict detection, which has the advantage of allowing a simpler automaton and algorithm for conflict detection. The automata we create can also be used as a basis for other kinds of analysis, including the possibility of performing queries, the detection of unreachable clauses, and the identification of superfluous clauses. In particular, the detection of unreachable clauses can be very useful in identifying parts of a contract which may be useless. This would generate more lightweight monitors, for runtime verification.

Based on the constructions presented, we have implemented a model checker for detecting conflicts in $\mathcal{CL}$ (the tool CLAN [1]). In other ongoing work using the semantics presented in this paper, we are using the automata created from $\mathcal{CL}$ contracts for runtime verification using LARVA [3]. This enables the writing of contracts about Java programs and automatically obtaining monitors ensuring conformance at runtime.

We believe that contract analysis is essential in dynamic contract composition. Even in the case of a single contract, conflict analysis can be a useful aid, as shown in the case study we present. Moreover, when dynamically generated contracts are to be used, the analysis becomes even more valuable. The main advantage of using a deontic approach is that the obligations, permissions and prohibitions are explicitly identified, and differentiated from conditionals. This enables an analysis focusing only on conflicts at the deontic level.

Please refer to [5] for more details and full proofs.

References

1. CLAN. CL ANalyser – A tool for Contract Analysis. Available from `www.cs.um.edu.mt/~svrg/Tools/CLTool/`.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.
3. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS*, LNCS. To appear, 2008.

4. N. Dinesh, A. Joshi, I. Lee, and O. Sokolsky. Reasoning about conditions and exceptions to laws in regulatory conformance checking. In *DEON*, volume 5076 of *LNCS*, 2008.
5. S. Fenech. Conflict analysis of deontic contracts. Master's thesis, Dept. of Computer Science, Univ. of Malta, 2008.
6. M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA*, volume 5311 of *LNCS*, pages 397–407. Springer-Verlag, 2008.
7. A. Lomuscio, H. Qu, and M. Solanki. Towards verifying compliance in agent-based web service compositions. In *AAMAS*, pages 265–272, 2008.
8. P. McNamara. Deontic logic. In *Handbook of the History of Logic*, volume 7, pages 197–289. North-Holland Publishing, 2006.
9. G. Pace, C. Prisacariu, and G. Schneider. Model Checking Contracts –a case study. In *ATVA*, volume 4762 of *LNCS*, pages 82–97. Springer, 2007.
10. C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In *FMOODS*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.
11. C. Prisacariu and G. Schneider. CL: A Logic for Reasoning about Legal Contracts – Semantics. Technical Report 371, Univ. Oslo, 2008.
12. B. Wozna, A. Lomuscio, and W. Penczek. Bounded model checking for knowledge and real time. In *AAMAS*, pages 165–172. ACM, 2005.
13. G. H. V. Wright. Deontic logic. *Mind*, (60):1–15, 1951.