Conflict Analysis of Deontic Contracts

Stephen Fenech
Dept. of Computer Science
University of Malta
sfen002@um.edu.mt

Gordon J. Pace
Dept. of Computer Science
University of Malta
gordon.pace@um.edu.mt

Gerardo Schneider
Dept. of Informatics
University of Oslo, Norway
gerardo@ifi.uio.no

Abstract

Industry is currently pushing towards Service Oriented Architecture where code execution is not limited to the organisational borders but may extend outside of the organisation to which the sources are typically not accessible. In order to protect the interests of the organisation contracts are used which can be seen as a list of obligations, permissions and prohibitions. The composition of different services with different contracts, and the combination of service contracts with local contracts can give rise to conflicts, exposing the need for automatic techniques for contract analysis. In this paper we investigate how conflict analysis can be performed automatically for contracts specified in the contract language $\mathcal{CL}$.

Index Terms

Contracts, Deontic Logic, Conflict Analysis

1. Introduction

Today's trend towards service oriented architectures, in which different decoupled services distributed not only on different machines within a single organisation but also outside of it, provides new challenges for reliability and trust. Since an organisation is executing code which it has no access to, it requires mechanisms to protect itself and to secure its trust in the external organisation's service.

Deontic logic allows us to reason about normative behaviour, such as obligations, permissions and prohibitions. Unfortunately, deontic logic is plagued with many paradoxes making it very challenging to develop a system which is highly expressive, and free from paradoxes. One approach to solve this problem is by restricting the application domain, and develop a restricted deontic logic expressive enough for that particular domain but also paradox free. In [7] $\mathcal{CL}$ (Contract Language) was introduced to specify electronic contracts in a formal manner. Thus it is possible to formally analyse these contracts to ensure that they have certain desirable properties.

Since services are frequently composed of different sub-services, each with its own contracts, an important property one has to ensure is that a contract is conflict-free — meaning that the contract will never lead to conflicting or contradictory deontic directives. If the contract is not conflict free, during the enacting of the contract, one might end up in a state in which there is no possible way of satisfying the contract which is clearly not a desired situation.

Since the semantics of $\mathcal{CL}$ are written formally, they can be interpreted by machines where now we can have multiple machines automatically negotiating these electronic contracts. For example, if an organisation is exposing a web service, it could also expose a contract to which all clients must abide. Similarly, a client could also have a contract to which the service provider must abide to and thus in this case both contracts should be merged and proved to be contradiction-free. On the other hand, in a non-dynamic environment legal personnel may manually analyse the contracts, in this case we would like to have an automated method to search for conflicts in the contract in order to aid the personel drafting the contracts.

In this paper, we extend the trace semantics of $\mathcal{CL}$ to support conflict analysis, and present an automatic technique for conflict detection of $\mathcal{CL}$ contracts.

2. Deontic Logic and $\mathcal{CL}$

Deontic logic enables reasoning about normative and non-normative behaviour [4] such as obligations, prohibitions and permissions — a logic making a distinction between ideal behaviour and actual behaviour [3].

The normative notions of deontic logic have been investigated as far back as the time of Aristotle, but were first analysed in a formal manner by von Wright in [9][1].

Deontic logic tends to give rise to numerous paradoxes making it difficult to have an expressive system which does not allow the expression of awkward situations (some authors even described certain paradoxes as embarrassing). Furthermore, it is not trivial to solve these paradoxes since, all too frequently, solving paradoxes ends up in introducing new ones [4].

Instead of trying to solve the problem of having a complete paradox-free dontic logic, the designers of $\mathcal{CL}$ restricted the application domain of deontic contracts to electronic contracts. In this way the expressivity of the logic is reduced, resulting in a language which is claimed to be free from most classical paradoxes [7], [8], but still retains the required flexibility and expressivity for practical use.

$\mathcal{CL}$ defines contracts on actions rather than state-of-affairs — an ought-to-do as opposed to an ought-to-be approach. This has been argued that it solves a number of paradoxes [4]. Furthermore, deontic properties written in the ought-to-be approach can be defined using the ought-to-do approach and vice versa so no expressivity is lost. Since the contracts are defined on actions, $\mathcal{CL}$ can be seen as the merging of deontic logic and dynamic logic. Another important feature is the handling of exceptional behaviour with Contrary-to-Duty (CTDs) and Contrary-to-Prohibition (CTPs) clauses[2] These allow direct descriptions of exceptional behaviour which most other logics fail to capture.

Definition 2.1: The syntax of $\mathcal{CL}$

$$\begin{array}{rcl} C & := & C_O|C_P|C_F|C \wedge C|[\beta]C|\top|\bot \\ C_O & := & O_C(\alpha)|C_O \oplus C_O \\ C_P & := & P(\alpha)|C_P \oplus C_P \\ C_F & := & F_C(\delta)|C_F \vee [\alpha]C_F \\ \alpha & := & 0|1|a|\alpha\&\alpha|\alpha \cdot \alpha|\alpha + \alpha \\ \beta & := & 0|1|a|\beta\&\beta|\beta \cdot \beta|\beta + \beta|\beta^* \end{array}$$

Looking at the syntax of $\mathcal{CL}$ (Definition 2.1) we have the deontic logic operators of CTDs, CTPs and permissions whereas from dynamic logic we have the compound actions and the '[ ]' operators which semantically get the same meaning as in propositional dynamic logic.

1. Some authors credit Mally as the first who has attempted to formalise deontic logic, however his system was found out to be a fragment of alethic propositional logic and thus not really a deontic logic.

2. In [2] the authors prefer to define CTDs and CTPs as basic operators as opposed to previous papers [7], [5] in which CTDs and CTPs where constructed from the other basic operators.

A contract clause C can be either an obligation (C_O), permission (C_P) or prohibition (C_F) clause, a conjunction of two clauses 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. $F_C(\alpha)$ is interpreted as forbidden to perform α and if α is performed then the reparation C must be executed. The interpretation of $[\beta]C$ is if action β is performed then the contract C must be executed — if β is not performed, the contract is trivially satisfied. Compound actions can be constructed from basic ones using the operators $\&$, $\cdot$, $+$ and * where $\&$ stands for the actions occuring concurrently, $\cdot$ stands for the actions to occur in sequence, $+$ stands for a choice between actions and * is the Kleene star. 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 refarred to as the canonical form:

Theorem 2.1: For any compound action defined in terms of +, & and $\cdot$ there exists an equivalent action called the canonical form of the form: $+_{i \in I} \alpha^i_\& \cdot \alpha^i$, where each $\alpha^i_\& \in A^\&_B$ and α^i is another action in canonical form.

In the rest of this paper we assume that action expressions have been reduced to this form. One should also note that 1 is an action expression matching any action, while 0 is the impossible action. 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 as $[checkIn]O_{O(pay)}(withinWeightLimit)$

2.1. Semantics

In the first paper about $\mathcal{CL}$ [7] the authors defined the semantics of the language using an extension of μ-calculus. However, in our case we do not need this branching semantics but use the simpler trace semantics given in [2].

Given a set A of basic actions, a state s and a transition t labeled with a set of basic actions ($l(t) \subseteq A$). An infinite trace in such a system is an infinite sequence of transition labels. Given a contract C, and an infinite trace σ, one can define $\sigma \vDash_\infty C$ (σ satisfies contract C) as follows [2]:

Definition 2.2: Trace semantics as specified in [2]:

$\sigma \models_\infty C_1 \wedge C_2$ if $\sigma \models_\infty C_1$ and $\sigma \models_\infty C_2$

$\sigma \models_\infty C_1 \vee C_2$ if $\sigma \models_\infty C_1$ or $\sigma \models_\infty C_2$

$\sigma \models_\infty C_1 \oplus C_2$ if $(\sigma \models_\infty C_1$ and $\sigma \not\models C_2)$ or $(\sigma \not\models C_1$ and $\sigma \models_\infty C_2)$

$\sigma \models_\infty [\alpha_\&]C$ if $\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..) \models_\infty C,$ or $\alpha_\& \nsubseteq \sigma(0)$

$\sigma \models_\infty [\beta;\beta']C$ if $\sigma \models_\infty [\beta][\beta']C$

$\sigma \models_\infty [\beta+\beta']C$ if $\sigma \models_\infty [\beta]C$ and $\sigma \models_\infty [\beta']C$

$\sigma \models_\infty [\beta^*]C$ if $\sigma \models_\infty C$ and $\sigma \models_\infty [\beta][\beta^*]C$

$\sigma \models_\infty O_C(\alpha_\&)$ if $\alpha_\& \subseteq \sigma(0),$ or if $\sigma(1..) \models_\infty C$

$\sigma \models_\infty O_C(\alpha;\alpha')$ if $\sigma \models_\infty O_C(\alpha)$ and $\sigma \models_\infty [\alpha]O_C(\alpha')$

$\sigma \models_\infty O_C(\alpha+\alpha')$ if $\sigma \models_\infty O_\perp$ or $\sigma \models_\infty O_\perp(\alpha')$ or $\sigma \models_\infty [\overline{\alpha+\alpha'}]C$

$\sigma \models_\infty F_C(\alpha_\&)$ if $\alpha_\& \nsubseteq \sigma(0),$ or if $\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..) \models_\infty C$

$\sigma \models_\infty F_C(\alpha;\alpha')$ if $\sigma \models_\infty F_\perp(\alpha)$ or $\sigma \models_\infty [\alpha]F_C(\alpha')$

$\sigma \models_\infty F_C(\alpha+\alpha')$ if $\sigma \models_\infty F_C(\alpha)$ and $\sigma \models_\infty F_C(\alpha')$

$\sigma \models_\infty [\overline{\alpha_\&}]C$ if $\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..) \models_\infty C$ or if $\alpha_\& \subseteq \sigma(0)$

$\sigma \models_\infty [\overline{\alpha;\alpha'}]C$ if $\sigma \models_\infty [\overline{\alpha}]C$ and $\sigma \models_\infty [\alpha][\overline{\alpha'}]C]$

$\sigma \models_\infty [\overline{\alpha+\alpha'}]C$ if $\sigma \models_\infty [\overline{\alpha}]C$ or $\sigma \models_\infty [\overline{\alpha'}]C$

We will use lower case letters $(a, b\ldots)$ to represent atomic actions, Greek letters $(\alpha, \beta\ldots)$ for compound actions, and Greek letters with a subscript & $(\alpha_\&, \beta_\&, \ldots)$ for compound concurrent actions built from atomic actions and the concurrency operator &. The set of all such concurrent actions will be written $A_\&$. Certain basic actions are mutually exclusive (for example, opening the check-in desk and closing the check-in desk), which we will write as $\alpha\#\alpha'$.

In order for a sequence σ to satisfy an obligation, $O_C(\alpha_\&)$, $\alpha_\&$ must be a subset or equal to $\sigma(0)$ or the rest of the trace satisfies the reparation C, thus for the obligation to be satisfied all the atomic actions in $\alpha_\&$ must be present in the first set of the sequence. For a prohibition to be satisfied, the converse is required, that is, not all the actions of $\alpha_\&$ are executed in the first step of the sequence. One should note that permission is not defined in this semantics since a trace cannot violate a permission clause[3]. An important observation is that the negation of an action is defined as performing any other action except the negated action. A detailed description of the algebra of actions used in $\mathcal{CL}$ can be found in [6].

This trace semantics enables checking whether or not a trace satisfies a contract. However, deontic information is not preserved in the trace making it inadequate for conflict analysis. Thus this semantics cannot be used to identify the conflicts in a contract . Using only this trace semantics we can only check if the contract is satisfiable but not that it is conflict free[4].

In order to enable conflict analysis, we start by adding deontic information in an additional trace, giving two parallel traces — a trace of actions (σ) and a trace of deontic notions (σ_d). Similar to σ, σ_d is defined as a sequence of sets whose elements are from the set D_a which is defined as $\{O_a \mid a \in A\} \cup \{F_a \mid a \in A\} \cup \{P_a \mid a \in A\}$ where O_a stands for the obligation to do a, F_a stands for the prohibition to do a and P_a for permission to do a.

Furthermore, since conflicts may result in sequences of finite behaviour which cannot be extended (due to the conflict), we reinterpret the semantics over finite traces. As described earlier, a conflict may result in reaching a state where we have only the option of violating the contract, thus any infinite trace which leads to this conflicting state will result not being accepted by the semantics. We need to be able to check that a finite trace has not yet violated the contract and then check if the following state is conflicting.

We will use ";" to denote catenation of two sequences, and len to return the length of a finite sequence. Two traces are pointwise (synchronously) joined using the combine operator where we will use the $\cup$ symbol and defined as: $(\sigma \cup \sigma')(n) = \sigma(n) \cup \sigma'(n)$. Furthermore, if α is a set of atomic actions then we will use O_α to denote the set $\{O_a \mid a \in \alpha\}$.

The trace semantics is defined in Definition 2.3, where $\sigma, \sigma_d \models_f C$ can be interpreted as 'the finite action sequence σ and deontic sequence σ_d do not violate contract C':

Definition 2.3: Finite trace semantics with deontic information

$\sigma, \sigma_d \not\models_f C$ if $len(\sigma) \neq len(\sigma_d)$

$\sigma, \sigma_d \models_f \top$ if $len(\sigma) = 0$ or $\forall i \sigma_d(i) = \emptyset$

$\sigma, \sigma_d \not\models_f \perp$

$\sigma, \sigma_d \models_f C_1 \wedge C_2$ if $\sigma, \sigma'_d \models_f C_1$ and $\sigma, \sigma''_d \models_f C_2$ and $\sigma_d = \sigma'_d \cup \sigma''_d$

3. In the full semantics, permission is defined as the existence/possibility of performing the action, however, when looking at a trace we cannot determine if this possibility exists or not.

4. A contract is not satisfiable if any sequence of actions will lead to a conflict thus there will be no possible trace that satisfies the contract, however a satisfiable contract may still have conflicts where certain traces will lead to a conflict but not all.

$$\begin{array}{lll}
\sigma, \sigma_d & \vDash_f & C_1 \oplus C_2 \text{ if } \sigma, \sigma_d \vDash_f C_1 \text{ or } \sigma, \sigma_d \vDash_f C_2 \\
\sigma, \sigma_d & \vDash_f & [\alpha_\&]C \text{ if } len(\sigma) = 0 \text{ or } \sigma_d(0) = \emptyset \text{ and} \\
 & & (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash_f C, \text{ or} \\
 & & \alpha_\& \nsubseteq \sigma(0))) \\
\sigma, \sigma_d & \vDash_f & [\beta;\beta']C \text{ if } \sigma, \sigma_d \vDash_f [\beta][\beta']C \\
\sigma, \sigma_d & \vDash_f & [\beta+\beta']C \text{ if } \sigma, \sigma_d \vDash_f [\beta]C \wedge [\beta']C \\
\sigma, \sigma_d & \vDash_f & [\beta^*]C \text{ if } \sigma, \sigma_d \vDash_f C \wedge [\beta][\beta^*]C \\
\sigma, \sigma_d & \vDash_f & [C_1?]C_2 \text{ if } \sigma, \sigma_d \nvDash_f C_1, \text{ or } \sigma, \sigma_d \vDash_f C_1 \wedge C_2 \\
\sigma, \sigma_d & \vDash_f & O_C(\alpha_\&) \text{ if } len(\sigma) = 0 \text{ or } \sigma_d(0) = O\alpha \text{ and} \\
 & & ((\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash_f \top) \text{ or} \\
 & & \sigma(1..), \sigma_d(1..) \vDash_f C) \\
\sigma, \sigma_d & \vDash_f & O_C(\alpha;\alpha') \text{ if } \sigma, \sigma_d \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha') \\
\sigma, \sigma_d & \vDash_f & O_C(\alpha+\alpha') \text{ if } \sigma, \sigma_d \vDash_f O_\perp(\alpha) \text{ or} \\
 & & \sigma, \sigma_d \vDash_f O_\perp(\alpha') \text{ or } (\sigma_d(0) = (O\alpha \text{ or } O\alpha') \\
 & & \text{ and } \sigma, \emptyset; \sigma_d(1..) \vDash_f [\overline{\alpha+\alpha'}]C) \\
\sigma, \sigma_d & \vDash_f & F_C(\alpha_\&) \text{ if } len(\sigma) = 0 \text{ or } \sigma_d(0) = F\alpha \text{ and} \\
 & & ((\alpha_\& \nsubseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash_f \top) \text{ or} \\
 & & (\alpha_\& \subseteq \sigma(0) \text{ and } \sigma(1..), \sigma_d(1..) \vDash_f C)) \\
\sigma, \sigma_d & \vDash_f & F_C(\alpha;\alpha') \text{ if } \sigma_d(0) = F\alpha \text{ and} \\
 & & (\sigma, \sigma_d \vDash_f F_\perp(\alpha) \text{ or } \sigma, \sigma_d \vDash_f [\alpha]F_C(\alpha')) \\
\sigma, \sigma_d & \vDash_f & F_C(\alpha+\alpha') \text{ if } \sigma, \sigma_d \vDash_f F_C(\alpha) \wedge F_C(\alpha') \\
\sigma, \sigma_d & \vDash_f & [\overline{\alpha_\&}]C \text{ if } \sigma_d(0) = \emptyset \text{ and } ((\alpha_\& \nsubseteq \sigma(0) \text{ and} \\
 & & \sigma(1..), \sigma_d(1..) \vDash_f C) \text{ or } \alpha_\& \subseteq \sigma(0)) \\
\sigma, \sigma_d & \vDash_f & [\overline{\alpha;\alpha'}]C \text{ if } \sigma, \sigma_d \vDash_f [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C] \\
\sigma, \sigma_d & \vDash_f & [\overline{\alpha+\alpha'}]C \text{ if } \sigma_d(0) = \emptyset \text{ and} \\
 & & (\sigma\sigma_d \vDash_f [\overline{\alpha}]C \text{ or } \sigma, \sigma_d \vDash_f [\overline{\alpha'}]C)
\end{array}$$

$$\begin{array}{lll}
\sigma, \sigma_d & \vDash_f & P(\alpha) \text{ if } len(\sigma) = 0 \text{ or } \sigma_d(0) = P\alpha \text{ and} \\
 & & \sigma(1..), \sigma_d(1..) \vDash_f \top \\
\sigma, \sigma_d & \vDash_f & P(\alpha;\alpha') \text{ if } \sigma, \sigma_d \vDash_f P(\alpha) \wedge [\alpha]P(\alpha') \\
\sigma, \sigma_d & \vDash_f & P(\alpha+\alpha') \text{ if } \sigma, \sigma_d \vDash_f P(\alpha) \wedge P(\alpha')
\end{array}$$

The main difference between the finite and infinite trace semantics are: the definition of $[\cdot]$, Obligation and Prohibition where we have to ensure that the trace is not empty (otherwise it cannot violate the contract) and the restrictions on σ_d. One should also note that if both sequences are not of equal length then they cannot satisfy this relation. in this semantics we do consider permission but the conditions for a trace not to violate the contract are defined on σ_d rather than the trace of actions so for any σ there exists a σ_d which will not violate a permission clause. Another point to note is that when there are no deontic notions in effect the corresponding element in σ_d is the empty set. It is because of this that we have to define the conjunction operator using the combine operator.
It can be proved that the infinite and finite trace semantics are sound and complete with respect to each other.

3. Conflict Analysis

A contract is said to contain a *conflict* in the following situations:

1) $O_\alpha \in \sigma_d(i)$ and $F_\alpha \in \sigma_d(i)$
2) $P_\alpha \in \sigma_d(i)$ and $F_\alpha \in \sigma_d(i)$
3) $O_\alpha \in \sigma_d(i)$ and $O'_\alpha \in \sigma_d(i)$ and $\alpha\#\alpha'$
4) $O_\alpha \in r_\sigma(i)$ and $P'_\alpha \in \sigma_d(i)$ and $\alpha\#\alpha'$

The first two conflicts are quite straightforward where we are obliged and forbidden to perform the same action or we are permitted and forbidden to perform the same action. In the first conflict we would end up in a state where whatever action is performed we will violate the contract. The second conflict situation would not result in having a trace that violates the contract since in the trace semantics permissions cannot be broken, however, since we are augmenting the original trace semantics with the deontic notions we can still identify these situations. The remaining two classes correspond to obligations (and permissions and obligations) of mutually exclusive actions. Freedom from conflict can be defined formally as follows:
Definition 3.1: A contract C is said to be conflict free if for all traces σ_f and σ_d satisfying $\sigma_f,\ \sigma_d \vDash_f C$, there is no conflict in σ_d, meaning that it is not the case that any of the following are true:

1) $\exists i \cdot Oa \in \sigma_d(i)$ and $Fa \in \sigma_d(i)$
2) $\exists i \cdot Pa \in \sigma_d(i)$ and $Fa \in \sigma_d(i)$
3) $\exists i \cdot Oa \in \sigma_d(i)$ and $Ob \in \sigma_d(i)$ and $a\#b$
4) $\exists i \cdot Oa \in \sigma_d(i)$ and $Pb \in \sigma_d(i)$ and $a\#b$

3.1. Automated Conflict Analysis

By unwinding a $\mathcal{CL}$ formula according to the finite trace semantics, one can create an automaton which accepts all non violating traces and with any trace resulting in a violation ending up in the violating state V. Furthermore, by labelling the states with deontic information provided in σ_d one can ensure that a contract is conflict free through the analysis of the resulting reachable states (non-violating states) for conflict freedom.
The states will contain a set of formulae still to be satisfied. For each operator in $\mathcal{CL}$ rules given for the generation of the automaton will be applied (Table 3.1). Each transition is labelled with the set of actions that are to be performed in order to move along

$C_1 \wedge C_2$	Add C_1 and C_2 to N
$[\alpha]C$	for every transition t where $\alpha_i.now \subseteq t.\alpha_{\&}$ add C to $t.n.N$ if $\alpha_i.next$ is empty, otherwise add $[\alpha_i.next]C$ to $t.n.N$
$[\beta^*]C$	Add C and $[\beta][\beta^*]C$ to N
$O_C(\alpha)$	for every transition t, if $\alpha_i.now \subseteq t.\alpha_{\&}$ and $\alpha_i.next$ is not empty, add $O_C(\alpha_i.next)$ to $t.n.N$, otherwise, if $\alpha_i.now \nsubseteq t.\alpha_{\&}$ add C to $t.n.N$
$F_C(\alpha)$	for every transition t, if $\alpha_i.now \subseteq t.\alpha_{\&}$ and $\alpha_i.next$ is empty, add C to $t.n.N$, otherwise, if $\alpha_i.next$ is not empty add $F_C(\alpha_i.next)$ to $t.n.N$
$P(\alpha)$	for every transition t, if $\alpha_i.now \subseteq t.\alpha_{\&}$ and $\alpha_i.next$ is not empty, add $P(\alpha_i.next)$ to $t.n.N$
$[\overline{\alpha}]C$	for every transition t where $\alpha_i.now \nsubseteq t.\alpha_{\&}$ add C to $t.n.N$ if $\alpha_i.next$ is empty, otherwise add $[\alpha_i.next]C$ to $t.n.N$

Table 1. Sub-formulae processing

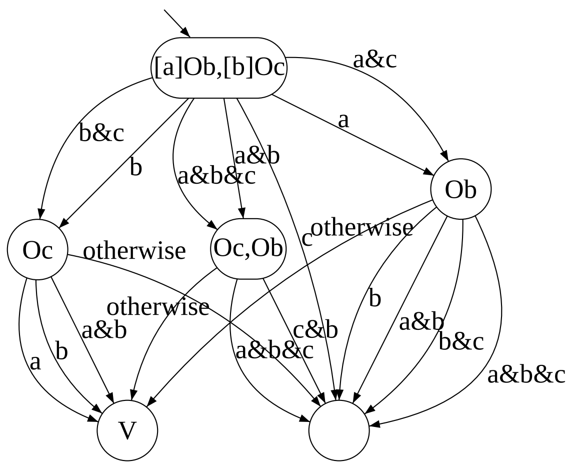

Figure 1. Automaton generated for $[a]Ob \wedge [b]Oc$ with all the transitions are created.

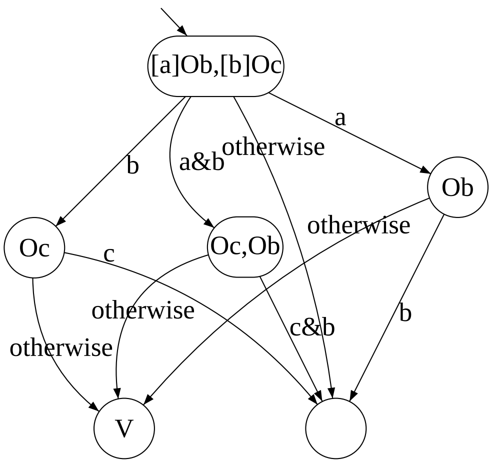

Figure 2. By creating only the necessary transitions the corresponding automaton for $[a]Ob \wedge [b]Oc$ is much smaller.

the transition. From the canonical form(Theorem 2.1) we can look at an action as a disjunction of actions that must occur now and for each of these a compound action that needs to occur in the next step. This view is very helpful when processing the actions since a compound action α can be seen as an array of possibilities α_i where for each entry we have the atomic actions which need to hold now ($\alpha_i.now$) and the possibly compound or empty actions that need to follow next ($\alpha_i.next$).

Once the automaton is generated we can go through all the states and check for the four types of conflicts. One should also note that if there is a conflict of type one or three, then all transitions out of the state go to the violation state.

In order to apply the rules of Table 3.1 we need to generate all possible transitions before processing the sub-formulae, thus for every state we have a fixed number of transitions, depending on the action alphabet. Most of the time this results in having more transitions than actually required because in most states there will be actions which will not affect the outcome of the transition. We could thus improve the algorithm in such a way that we create all and only those required transitions; however this will make the algorithm more complex.

Consider processing the formula $[a]Ob \wedge [b]Oc$, if we generate all the transitions the result would be the automaton in Figure 1. One should note that out of each state we have all possible transitions (we grouped transitions as otherwise in order to make the automaton more readable). Consider processing the initial node where $P = [a]Ob, [b]Oc$. We start without any transitions and add new transitions as required while processing the sub-formula. When processing the first sub formula we add a transition with a and place Ob in the new node. However when we process the second sub formula we cannot simply add a transition with action b and add Oc to the new node. We will have to go through all the transitions already created and create a transition which is a combination of both actions. In this case we will need to create a transition $a\&b$ and add both Ob and Oc to the new node. The result can be seen in Figure 2.

Conflict analysis can also be done on-the-fly without the need to create the complete automaton. One can process the states without storing the transitions and store only satisfied subformulas (for termination). In this manner memory issues are reduced since only a part of the automaton is stored in memory.

4. Example of Conflict Analysis

The following is part of a contract that the ground crew working at a check-in desk agrees to:

1) *Once the desk is open, the check-in clerk is obliged to check in travellers.*
2) *Once the desk is closed, the check-in clerk is forbidden from checking in travellers.*
3) *If the traveller has not presented any acceptible form of identification the check-in clerk is forbidden from checking the traveller in.*

This contract would represented in $\mathcal{CL}$ as the conjunction of the following three clauses: (1) $[1^*][open]O(checkIn)$; (2) $[1^*][close]F(checkIn)$; and (3) $[1^*][\overline{presentID}]F(checkIn)$.

Analysis shows that when both the $open$ and $close$ actions occur we end up in a conflict. Adding mutual exclusivity of the open and close actions ($open\#close$) is still not enough since if the client did not present his ID then the desk personnel cannot check him in but the first clause does not have a provision for this. Adding that the client must also have presented the ID, results in a conflict-free contract.

$$\begin{array}{l} [1^*][open\&presentID]O(checkIn) \\ [1^*][close]F(checkIn) \\ [1^*][\overline{presentID}]F(checkIn) \\ open\#close \end{array}$$

Clause one of the original contract would need to be changed to *'Once the desk is open, he/she is obliged to check in travellers if a valid identification is provided.'*

5. Conclusions

In this paper, we have presented a finite trace semantics for $\mathcal{CL}$ augmented with deontic information, and showed how it can be used for automatic analysis of contracts for conflict discovery. We see this as a first step towards automatic contract negotiation since it is an essential property for a meaningful contract to have — any automatically created contract would be required to satisfy it. We also see this as a step towards a complete model checking of $\mathcal{CL}$ using the automaton approach which we use.

Once the automaton is created we can perform other analysis not just conflict analysis. One application we are currently looking at is runtime monitoring of contract and we have already implemented a translation from the automaton created from $\mathcal{CL}$ to the runtime verification tool LARVA [1]. Furthermore, we can also analyse the contract looking for unreachable clauses since, most of the time, either the clause can be removed or there is a mistake in the contract. For instance, the second clause in the contract $F(a) \wedge [a]O(b)$ is superfluous since it is not reachable without violating the contract.

The next step is to move from the linear trace semantics of $\mathcal{CL}$ to the full, branching semantics and provide means to model check contracts and also use contracts as properties to model check. This would be very desirable since there currently is no model checker for a deontic logic as expressive as $\mathcal{CL}$.

References

[1] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*. To be published by Springer Verlag in Lecture Notes in Computer Science, 2008.

[2] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA'08*, LNCS, Seoul, South Korea, October 2008. Springer-Verlag. To appear.

[3] J.-J. C. Meyer, F. Dignum, and R. Wieringa. The paradoxes of deontic logic revisited: A computer science perspective (or: Should computer scientists be bothered by the concerns of philosophers?). Technical Report UU-CS-1994-38, Department of Information and Computing Sciences, Utrecht University, 1994.

[4] J.-J. C. Meyer and R. J. Wieringa, editors. *Deontic logic in computer science: normative system specification*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[5] G. Pace, C. Prisacariu, and G. Schneider. Model Checking Contracts –a case study. In *5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 82–97, Tokyo, Japan, October 2007. Springer.

[6] C. Prisacariu and G. Schneider. An Algebraic Structure for the Action-Based Contract Language CL - theoretical results. Technical Report 361, Department of Informatics, University of Oslo, Oslo, Norway, July 2007.

[7] C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In M. Bonsangue and E. B. Johnsen, editors, *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189, Paphos, Cyprus, June 2007. Springer.

[8] C. Prisacariu and G. Schneider. Towards a Formal Definition of Electronic Contracts. Technical Report 348, Department of Informatics, University of Oslo, Oslo, Norway, January 2007.

[9] G. von Wright. Deontic logic. *Mind*, (60):1–15, 1951.