Detection of Conflicts in Electronic Contracts*

Stephen Fenech, Gordon J. Pace
Dept. of Computer Science
University of Malta, Msida, Malta
{sfen002,gordon.pace}@um.edu.mt

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

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 may need to execute code provided by third parties, it requires mechanisms to protect itself. One of such mechanisms is the use of *contracts*.
Since services are frequently composed of different subservices, each with its own contract, there is a need to guarantee that each single contract is conflict-free. Moreover, one needs to ensure that the conjunction of all the contracts is also conflict-free —meaning that the contracts will never lead to conflicting or contradictory normative directives.
$\mathcal{CL}$ [5] is a formal language to specify deontic electronic contracts. A trace semantics for the language was presented in [4], useful for runtime monitoring of $\mathcal{CL}$ contracts. Such semantics, however, lacks the deontic information concerning the obligations, permissions and prohibitions of the involved parties in the contract, and thus it is not suitable for conflict analysis.
We present here an extension of the trace semantics of $\mathcal{CL}$ given in [4] to support conflict analysis. Based on that semantics we have developed a decision procedure to automatically detect conflicts in contracts written in $\mathcal{CL}$. We have implemented such an algorithm into an *ad hoc* model checker. Due to space restriction we only present in what follows the $\mathcal{CL}$ syntax, the extended trace semantics, and a brief discussion on the automata construction basis of our model checker.

2 Deontic Logic and $\mathcal{CL}$

Deontic logic [6] 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.
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 mean 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:

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

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 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). $F_C(\alpha)$ is interpreted as forbidden to perform α and if α is performed then the reparation C must be executed (a CTP). $[\beta]C$ is interpreted as 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 referred to as the canonical form. In the rest of this paper we assume that action expressions have been reduced to this form. 1 is an action expression matching any action, while 0 is the impossible action. In order to avoid paradoxes the operators combining obligations, permissions and prohibitions

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

are restricted syntactically. See [5, 4] 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 $[checkIn]O_{O(pay)}(withinWeightLimit)$.

2.1 Trace Semantics

The trace semantics presented in [4] enables checking whether or not a trace satisfies a contract. However, deontic information is not preserved in the trace and thus it is not suitable to be used for conflict detection. By a conflict we mean for instance that the contract permits and forbids performing the same action at the same time (see below for a more formal definition of conflict). We present in what follows an extension of the trace semantics given in [4].

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 $A_\&$. We use $\#$ to denote mutually exclusive actions (for example, if a stands for 'opening the check-in desk' and b for 'closing the check-in desk', we write $a\#b$).

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. An important observation is that the negation of an action is defined as performing any other action except the negated action.

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.

Also, since conflicts may result in sequences of finite behaviour which cannot be extended (due to the conflict), we reinterpret the semantics over finite traces. 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 use a semicolon (;) 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: $(\sigma \cup \sigma')(n) = \sigma(n) \cup \sigma'(n)$. 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 extended trace semantics for $\mathcal{CL}$ is given below, where $\sigma, \sigma_d \vDash_f C$ can be interpreted as 'finite action sequence σ and deontic sequence σ_d do not violate contract C':

$\sigma, \sigma_d \nvDash_f C$ if $len(\sigma) \neq len(\sigma_d)$

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

$\sigma, \sigma_d \nvDash_f \bot$

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

$\sigma, \sigma_d \vDash_f C_1 \oplus C_2$ if $\sigma, \sigma_d \vDash_f C_1$ or $\sigma, \sigma_d \vDash_f C_2$

$\sigma, \sigma_d \vDash_f [\alpha_\&]C$ if $len(\sigma) = 0$ or $\sigma_d(0) = \emptyset$ and ($\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C$, or $\alpha_\& \nsubseteq \sigma(0)$))

$\sigma, \sigma_d \vDash_f [\beta; \beta']C$ if $\sigma, \sigma_d \vDash_f [\beta][\beta']C$

$\sigma, \sigma_d \vDash_f [\beta + \beta']C$ if $\sigma, \sigma_d \vDash_f [\beta]C \wedge [\beta']C$

$\sigma, \sigma_d \vDash_f [\beta^*]C$ if $\sigma, \sigma_d \vDash_f C \wedge [\beta][\beta^*]C$

$\sigma, \sigma_d \vDash_f [C_1?]C_2$ if $\sigma, \sigma_d \nvDash_f C_1$, or $\sigma, \sigma_d \vDash_f C_1 \wedge C_2$

$\sigma, \sigma_d \vDash_f O_C(\alpha_\&)$ if $len(\sigma) = 0$ or $\sigma_d(0) = O\alpha$ and (($\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top$) or $\sigma(1..), \sigma_d(1..) \vDash_f C$)

$\sigma, \sigma_d \vDash_f O_C(\alpha; \alpha')$ if $\sigma, \sigma_d \vDash_f O_C(\alpha) \wedge [\alpha]O_C(\alpha')$

$\sigma, \sigma_d \vDash_f O_C(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f O_\bot(\alpha)$ or $\sigma, \sigma_d \vDash_f O_\bot(\alpha')$ or ($\sigma_d(0) = (O\alpha$ or $O\alpha')$ and $\sigma, \emptyset; \sigma_d(1..) \vDash_f [\overline{\alpha + \alpha'}]C$)

$\sigma, \sigma_d \vDash_f F_C(\alpha_\&)$ if $len(\sigma) = 0$ or $\sigma_d(0) = F\alpha$ and (($\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top$) or ($\alpha_\& \subseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C$))

$\sigma, \sigma_d \vDash_f F_C(\alpha; \alpha')$ if $\sigma_d(0) = F\alpha$ and ($\sigma, \sigma_d \vDash_f F_\bot(\alpha)$ or $\sigma, \sigma_d \vDash_f [\alpha]F_C(\alpha')$)

$\sigma, \sigma_d \vDash_f F_C(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f F_C(\alpha) \wedge F_C(\alpha')$

$\sigma, \sigma_d \vDash_f [\overline{\alpha_\&}]C$ if $\sigma_d(0) = \emptyset$ and (($\alpha_\& \nsubseteq \sigma(0)$ and $\sigma(1..), \sigma_d(1..) \vDash_f C$) or $\alpha_\& \subseteq \sigma(0)$)

$\sigma, \sigma_d \vDash_f [\overline{\alpha; \alpha'}]C$ if $\sigma, \sigma_d \vDash_f [\overline{\alpha}]C \wedge [\alpha][\overline{\alpha'}]C]$

$\sigma, \sigma_d \vDash_f [\overline{\alpha + \alpha'}]C$ if $\sigma_d(0) = \emptyset$ and ($\sigma\sigma_d \vDash_f [\overline{\alpha}]C$ or $\sigma, \sigma_d \vDash_f [\overline{\alpha'}]C$)

$\sigma, \sigma_d \vDash_f P(\alpha)$ if $len(\sigma) = 0$ or $\sigma_d(0) = P\alpha$ and $\sigma(1..), \sigma_d(1..) \vDash_f \top$

$\sigma, \sigma_d \vDash_f P(\alpha; \alpha')$ if $\sigma, \sigma_d \vDash_f P(\alpha) \wedge [\alpha]P(\alpha')$

$\sigma, \sigma_d \vDash_f P(\alpha + \alpha')$ if $\sigma, \sigma_d \vDash_f P(\alpha) \wedge P(\alpha')$

Note that the conditions for a trace containing a permission not to violate the contract are defined on σ_d rather than on the trace of actions. So, for any σ there exists a σ_d which will not violate a permission clause. Also note

that in the absence of deontic notions the corresponding element in σ_d is the empty set. We have proved that the infinite and finite trace semantics are sound and complete with respect to each other.

3 Conflict Analysis

Conflicts in contracts arise for 4 different reasons. First, we can be obliged and forbidden to do the same action, and second, we can be permitted and forbidden to perform the same action. In the first conflict we would end up in a state where whatever we do 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 cases correspond to obligations (and permissions and obligations) of mutually exclusive actions. Freedom from conflict can be defined formally as follows (recall that $a\#b$ if a and b are mutually exclusive actions):

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* $a\#b$

By unwinding a $\mathcal{CL}$ formula according to the finite trace semantics, we create an automaton which accepts all non violating traces, and such that any trace resulting in a violation ends up in a violating state. Furthermore, we label the states of the automaton with deontic information provided in σ_d, so we can ensure that a contract is conflict free simply through the analysis of the resulting reachable states (non-violating states).

States of the automaton contain a set of formulae still to be satisfied, following the standard sub-formula construction (as done for instance for CTL). Each transition is labelled with the set of actions that are to be performed in order to move along the transition. From the canonical form assumption 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. If there is a conflict of type one or three, then all transitions out of the state go to a special violation state. In general we might need to generate all possible transitions before processing each sub-formula, resulting on a big automaton. In practice, we improve the algorithm in such a way that we create all and only those required transitions reducing the size considerably.

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 Final Remarks

In this paper, we have presented a finite trace semantics for $\mathcal{CL}$ augmented with deontic information, and sketched how it can be used for automatic analysis of contracts for conflict discovery. The automaton we have created here could also be used as a basis for other kinds of analysis not just conflict analysis. These include the possibility of performing queries, the detection of unreachable clauses, and the identification of superfluous clauses. Based on the construction presented in this paper, we have implemented a model checker for detecting conflicts in $\mathcal{CL}$ [1]. In other ongoing work using the semantics presented in this paper, we have implemented a translation from the automaton created from $\mathcal{CL}$ contracts into the runtime verification tool LARVA [2]. This enables us to write contracts about Java programs and automatically obtain monitors to ensure conformance to the contracts at runtime. More detailed trace semantics, the conflict analysis algorithm (including proof of soundness, completeness and termination), as well as a description of the tool can be found in [3].

References

[1] CLAN. CL ANalyser – A tool for Contract Analysis. Available from `www.cs.um.edu.mt/~svrg/Tools/CLTool/`.

[2] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS 2008*, LNCS, 2008.

[3] S. Fenech. Conflict analysis of deontic contracts. Master's thesis, Dept. of Computer Science, Univ. of Malta, 2008.

[4] M. Kyas, C. Prisacariu, and G. Schneider. Runtime monitoring of electronic contracts. In *ATVA'08*, LNCS. Springer-Verlag, Oct. 2008. To appear.

[5] C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, June 2007.

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