CLAN: A Tool for Contract Analysis and Conflict Discovery[⋆]

Stephen Fenech[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[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. As Service-Oriented Architectures are more widely adopted, it becomes more important to adopt measures for ensuring that the services satisfy functional and non-functional requirements. One approach is the use of contracts based on deontic logics, expressing obligations, permissions and prohibitions of the different actors. A challenging aspect is that of service composition, in which the contracts composed together may result in conflicting situations, so there is a need to analyse contracts and ensure their soundness. In this paper, we present CLAN, a tool for automatic analysis of conflicting clauses of contracts written in the contract language $\mathcal{CL}$. We present a small case study of an airline check-in desk illustrating the use of the tool.

1 Introduction and Background

In Service-Oriented Architectures services are frequently composed of different sub-services, each with its own contract. Not only does the service user require a a guarantee that each single contract is conflict-free, but also that the combination of the contracts is also conflict-free — meaning that the contracts will never lead to conflicting or contradictory normative directives. This is even more challenging in a dynamic setting, in which contracts may only be acquired at runtime.

A common view of contracts is that of properties which the system must (or is guaranteed) to satisfy. However, when analysing contracts for conflicts, the need to analyse and reason about contracts is extremely important, and looking at contracts simply as logical properties may hide conflicts altogether. The use of deontic logic to enable reasoning explicitly about normative information in a contract and about exceptional behaviour is one alternative approach to contract analysis. $\mathcal{CL}$ [3] is a formal language to specify deontic electronic contracts. The language has a trace semantics [2], which although useful for runtime monitoring of contracts, 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 have recently developed conflict analysis techniques for $\mathcal{CL}$ [1], and in this paper, we present a tool implementing these techniques for the automatic analysis of contracts written in $\mathcal{CL}$.

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

The contract language $\mathcal{CL}$ Deontic logics enable explicit reasoning about, not only the actual state of affairs in a system e.g. 'the client has paid,' but also about the *ideal* state of affairs e.g. 'the client is *obliged* to pay' or 'the client is *permitted* to request a service.' $\mathcal{CL}$ is based on a combination of deontic, dynamic and temporal logics, allowing the representation of the deontic notions 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 as *Contrary-to-Duties* (CTDs) and *Contrary-to-Prohibitions* (CTPs) respectively. $\mathcal{CL}$ contracts are written using the following syntax:

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

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). Permission is represented as $P(\alpha)$, identifying that the action expression α is permitted. Note that repetition in actions (using the $*$ operator) is not allowed inside the deontic modalities. They are, however allowed in dynamic logic-style conditions. $[\beta]C$ 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. $\wedge$ allows the conjunction of clauses, $\oplus$ is used as an exclusive choice between certain clauses, $\top$ and $\bot$ are the trivially satisfied (violated) contract. Compound actions can be constructed from basic ones using the operators $\&$, $\cdot$, $+$ and * where $\&$ stands for the actions occurring concurrently, $\cdot$ stands for the actions to occur in sequence, $+$ stands for choice, and * for repetition. 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 are restricted syntactically. See [3, 2] 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 $[checkIn]O_{O(pay)}(withinWeightLimit)$.

Trace Semantics The trace semantics presented in [2] 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).

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 $a\#b$). A trace is a sequence of sets of actions, giving the set of actions present at each point in time. The Greek letter σ will be used to represent traces, using functional notation for indexing starting at zero i.e. $\sigma(n)$ is the $(n-1)$th element of trace σ.

For a trace σ to satisfy an obligation, $O_C(\alpha_\&)$, $\alpha_\&$ must be a subset of $\sigma(0)$ or the rest of the trace must satisfy 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 prohibitions the converse is required, e.g. not all the actions of $\alpha_\&$ are executed in the first step.

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. Furthermore, if α is a set of concurrent atomic actions then we will use O_α to denote the set $\{O_a \mid a \in \alpha\}$. Note that the semantics is given in the form of $\sigma, \sigma_d \vDash C$, which says that an action trace σ and a trace with deontic information σ_d satisfies a contract C. We show the trace semantics for the obligation as an example:

$$\langle\rangle, \langle\rangle \vDash O_C(\alpha_\&)$$
$$(\beta : \sigma), (\beta_d : \sigma_d) \vDash O_C(\alpha_\&) \text{ if } (\alpha_\& \subseteq \beta \text{ and } O(\alpha_\&) \in \beta_d) \text{ or } \sigma, \sigma_d \vDash C$$

Note that in the above, pattern matching is used to split the traces into the head (β and β_d) and the tails (σ and σ_d). The above says that empty traces satisfy an obligation, while non-empty traces satisfy the contract if either (i) the obligation is satisfied, and the obligation is registered in the deontic trace; or (ii) the reparation (CTD) is satisfied by the remainder of the trace. See [1] for more details.

Conflict Analysis Conflicts in contracts arise for four different reasons:[3] (i) obligation and prohibition on the same action; (ii) permission and prohibition on the same action; (iii) obligation to perform mutually exclusive actions; and (iv) permission and obligation to perform mutually exclusive actions.

With conflicts of the first type one would end up in a state where performing any action leads to a violation of the contract. The second conflict type results in traces

[3] By tabulating all combinations of the deontic operators, one finds that there are two basic underlying cases of conflict — concurrent obligation and prohibition, and concurrent permission and prohibition. Adding constraints on the concurrency of actions, one can identify the additional two cases. More complex temporal constraints on actions may give rise to others, but which can also be reduced to these basic four cases.

which also violate the contract even though permissions cannot be broken, since the deontic information is kept in the semantics. The remaining two cases correspond to mutually exclusive actions. Freedom from conflict can be defined formally as follows: Given a 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' iff there exists at least one element $e \in D$ such that:

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

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.

As an example, let us consider the contract $C = [a]\mathbb{O}(b + c) \wedge [b]\mathbb{F}(b)$, stipulating the obligation of the choice of doing b or c after an a, and the prohibition of doing b if previously a b has been done. 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 the above definition).

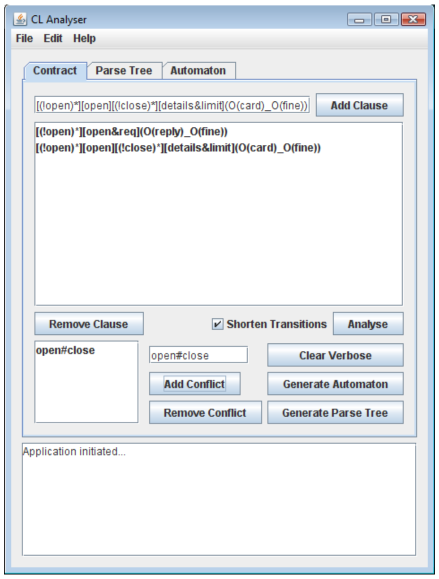
By unwinding a $\mathcal{CL}$ formula according to the finite trace semantics, we create an automaton which accepts all non-violating traces, and such that any trace resulting in a violation ends up in a violating state. Furthermore, we label the states of the automaton with deontic information provided 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 (e.g., as for CTL). Each transition is labelled with the set of actions that are to be performed in order to move along the transition.

Once the automaton is generated we can check for the four types of conflicts on all the states. If there is a conflict of type (i) or (iii), 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 subformulae (for termination), in this manner, memory issues are reduced since only a part of the automaton is stored in memory.

2 A Tool for Contract Analysis

CLAN[4] is a tool for detection of normative conflicts in $\mathcal{CL}$ contracts, enabling: (i) The automatic analysis of the contract for normative conflicts; (ii) The automatic generation of a monitor for the contract. The analyses are particularly useful when the contract is being written (enabling the discovery of undesired conflicts), before adhering to a given contract (to ensure unambiguous enforcement of the contract), and during contract enforcement (monitoring).

[4] CLAN can be downloaded from `http://www.cs.um.edu.mt/~svrg/Tools/CLTool`.



[(!open)*][open][(!close)*][details&limit](O(card)_O(fine))

[(!open)*][open&req](O(reply)_O(fine))
[(!open)*][open][(!close)*][details&limit](O(card)_O(fine))

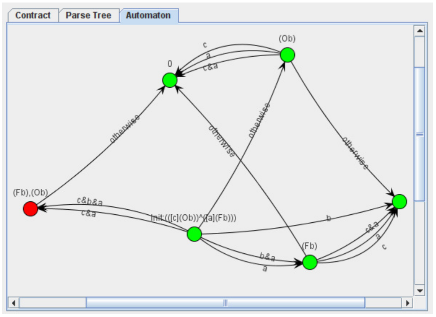
☑ Shorten Transitions

open#close

open#close

Application initiated...

(a)



(b)

Fig. 1. (a) Screen shot of the tool; (b) Automaton generated for $[c]\mathbb{O}(b) \wedge [a]\mathbb{F}(b)$.

The core of CLAN is implemented in Java, consisting of 700 lines of code. This does not include the additional code, over and above the conflict discovery algorithm, for the graphical user interface (a screen shot can be seen in Fig. 1-(a)). $\mathcal{CL}$ contracts and additional information (such as which actions are mutually exclusive) are given to the tool which then performs the conflict analysis. Upon discovery of a conflict, it gives a counter-example trace. The tool also offers an aid to analyse traces, and the possibility of visualising the generated automaton, as the one shown in Fig. 1-(b).

CLAN has been used to analyse a large contract resulting in a graph with 64,000 states, consuming approximately 700MB of memory; the analysis took around 94 minutes. The analysis seems to scale linearly in memory and polynomially in the number of states. The complexity of the automaton increases exponentially on the number of actions, since all the possible combinations to generate concurrent actions must be considered. We are currently working on how to optimise the analysis to avoid such exponential state-explosion.

3 Case Study

We briefly present here a portion of a small case study, starting from a draft contract written in English, which is formalised in $\mathcal{CL}$ and analysed using CLAN. The full example can be found in [1]. The case study concerns a contract between an airline company and a company taking care of the ground crew (mainly the check-in process) Some clauses of the contract expressed in English and $\mathcal{CL}$, are given below:

1. *The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves.*
   $[1^*][\mathit{twoHBefore}]O_{O(\mathit{issueFine})}(\mathit{openCheckIn}\ \&\ \mathit{requestInfo})$
2. *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.*
   $[1^*][\mathit{openCheckIn}][1^*](\mathbb{O}(\mathit{correctDetails}\ \&\ \mathit{luggageInLimit}) \wedge$
   $[\mathit{correctDetails}\ \&\ \mathit{luggageInLimit}]O_{O(\mathit{issueFine})}(\mathit{issueBoardingCard}))$
3. *The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before.*
   $([1^*][\mathit{20mBefore}]O_{O(\mathit{issueFine})}(\mathit{closeCheckIn}))\wedge([\overline{\mathit{20mBefore}}^*]\mathbb{F}_{O(\mathit{issueFine})}(\mathit{closeCheckIn}))$
4. *If any of the above obligations and prohibitions are violated a fine is to be paid.*
   $[1^*][\mathit{closeCheckIn}][1^*](F_{O(\mathit{issueFine})}(\mathit{openCheckIn})\wedge F_{O(\mathit{issueFine})}(\mathit{issueBoardingCard}))$

On this size of example, the tool gives practically instantaneous results, identifying conflicts such as the concurrent obligation and prohibition to perform action *issueBoardingCard*, together with a counter-example trace. Looking at clause 2, 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 4 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 one has to change clause 2 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. In the full case study, other conflicts due to mutual exclusion are also found and fixed in the English contract.

4 Conclusions

The analysis of contracts for the discovery of potential conflicts can be crucial to enable safe and dependable contract adoption and composition at runtime. In this paper we have presented CLAN, a tool for automatic detection of conflicting clauses in contracts written in the deontic language $\mathcal{CL}$. Currently the tool only provides the functionality of detecting conflicts. However, many other analysis may be done by slightly modifying the underlying algorithm, as for instance the generation of a model from the contract which can be processed by a model checker. Furthermore, other contract analysis techniques are planned to be added to the tool to enable analysis for overlapping, superfluous and unreachable sub-clauses of a contract. See [1] for related works.

References

1. S. Fenech, G. J. Pace, and G. Schneider. Automatic Conflict Detection on Contracts. In *ICTAC'09*, LNCS. Springer, 2009. To appear.
2. M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA'08*, volume 5311 of *LNCS*, pages 397–407. Springer-Verlag, 2008.
3. C. Prisacariu and G. Schneider. A Formal Language for Electronic Contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.