Automated Analysis of Conflicts in Software Product Lines

Enrique Martínez
Department of Computer Science
University of Castilla - La Mancha
Albacete, Spain
emartinez@dsi.uclm.es

Gerardo Schneider
Department of Applied IT
Chalmers | University of Gothenburg, Sweden
Department of Informatics
University of Oslo, Norway
gersch@chalmers.se

***Abstract*—In this paper we propose a framework where the behaviour of features can be modelled using a visual model language for contracts (*C-O Diagrams*). We present a partial translation from *C-O Diagrams* into the deontic contract language $\mathcal{CL}$ allowing to detect whether there are contradicting features, using the tool CLAN. We aim at handling conflicts arising from software evolution and variability. As a proof of concept we apply our technique to a trading system case study.**

***Keywords*-contracts; software product lines; deontic logic; conflict detection**

I. Introduction

The design, specification and analysis of software product lines (SPL) is not easy, one of the main problems being that of handling *variability* and software *evolution*. Variability is an important aspect of SPL, one of the reasons being that one naturally wants to have a family of software where concrete products are instantiations of the family varying only on some aspects. While modelling and analysing feature variability is already difficult, the real challenge comes when feature models evolve. Such evolution implies the addition or removal of certain features over time, eventually introducing constrains and hence conflicts with previous defined features.

The use of different logics for the automated analysis of SPL is widely spread in the literature [3]. Moreover, we can find some work based on applying deontic logic to SPL [2], but it is focused on model-checking properties. In this paper we present a conceptual framework, based on an extension of feature models in order to represent software product lines. Such a framework allows to express spatial variability, as well as evolution of the software family (time variability), together with a check on the possible conflicts arising from the addition/removal of features (the kind of conflicts we can detect are detailed in Subsection II-C).

As a proof of concept we apply our technique to a case study on a trading system, in which we expose some of the difficulties mentioned above. In particular, starting from a feature model describing a product family for the trading system, we incrementally add two different requirements: (i) the first shows how in some cases adding a new feature still allows compositional modelling and verification, and (ii) we then extend the modified model with an additional feature that restricts previous features on other parts of the model. The intention of this second modification is to highlight the difficulty of preserving compositional verification. In order to check the consistency of the modified models we (manually) translate everything into the contract language $\mathcal{CL}$ [9] and use the tool CLAN [6] to check for conflicts.

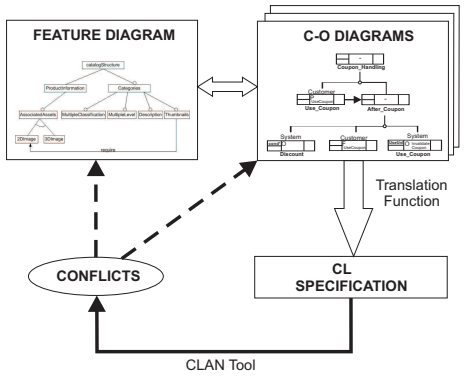

Figure 1. Workflow of the framework

In the conceptual framework we have defined the following artefacts can be distinguished:

- A **feature diagram** to engineer the requirements of the SPL. We consider the syntax and semantics of these diagrams defined in [10], which formalisation includes the management of feature model evolution.
- ***C-O Diagrams*** [7] to specify the expected behaviour of the different features composing the feature model.
- A $\mathcal{CL}$ **specification** that can be analysed to detect any behavioural conflict in the feature model.
- A **translation function** specifying how the transformation from *C-O Diagrams* into $\mathcal{CL}$ is done.
- The **CLAN tool** that automates the process of conflict detection over a $\mathcal{CL}$ specification.

The workflow followed by this framework is shown in Figure 1. First, we model the feature diagram corresponding to the SPL we want to analyse and the *C-O Diagrams* specifying the expected behaviour of the features we are interested in. Next, we use the translation function from *C-O Diagrams* to $\mathcal{CL}$ in order to obtain a specification in this language that can be automatically analysed. Finally, we apply the contract analyser of the CLAN tool over the resulting $\mathcal{CL}$ clauses. If the tool detects any conflict in the $\mathcal{CL}$ contract, the information provided by the tool is used to modify the models we have created to fix the problem in an

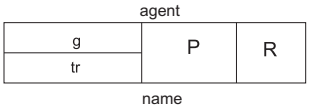

Figure 2. Box structure

appropriate way, performing afterwards the translation and the conflict detection processes again. This flow of events is repeated until all the conflicts have been solved. The integration of evolution in SPL inside this framework is quite straightforward. When we add a new feature in the feature model, we just have to create a new *C-O Diagram* corresponding to the expected behaviour of this feature and follow the steps described above to see if the new feature originates any conflict and how this conflict can be solved. In the case of evolution of an already existing feature, we have to modify the *C-O Diagram* corresponding to this feature according to its evolution, so we can check again if this evolution produces any conflict and how it can be fixed.

The rest of the paper is organized as follows: A background on some of the artifacts we use is given in Section II, and the translation from *C-O Diagrams* to $\mathcal{CL}$ is described in Section III. The application of our technique to a case study on a trading system is depicted in Section IV. Finally, we conclude in Section V.

II. Background

As feature diagrams description can be found in many places [10], in this section we focus on describing the other artefacts of our framework: the visual model *C-O Diagrams*, the contract language $\mathcal{CL}$ and the CLAN tool.

A. C-O Diagrams

A *C-O Diagram* [7] is a hierarchical tree diagram that we use to specify the clauses of a contract. These clauses can be related to the expected behaviour of some features part of a feature model, so these diagrams can be used to analyse feature models and their evolution.

In Figure 2 we show the basic element of our *C-O Diagrams*, namely a **box**, which is divided into four fields. On the left-hand side of the box we specify the conditions and restrictions. The *guard* **g** specifies the conditions under which the contract clause must be taken into account. The *time restriction* **tr** specifies the time frame in which the contract clause must be satisfied. The *propositional content* **P**, in the middle, is the main field of the box, and it is used to specify normative aspects (obligations, permissions and prohibitions) that are applied over actions, and/or the actions themselves. The last field of these boxes, on the right-hand side, is the *reparation* **R**. This reparation, if specified by the contract clause, is another contract that must be satisfied in case the main norm is not satisfied (a prohibition is violated or an obligation is not fulfilled, there is not reparation for permission), considering the clause eventually satisfied if this reparation is satisfied. Each box has also a name and an agent. The *name* is useful both to describe the clause and to reference the box from other clauses, so it must be unique. The *agent* indicates who the performer of the action is.

These basic elements of *C-O Diagrams* can be refined by using AND/OR/SEQ refinements, as shown in Figure 3. The aim of these refinements is to capture the hierarchical clause structure followed by most contracts. An **AND-refinement** means that all the subclauses must be satisfied in order to satisfy the parent clause. An **OR-refinement** means that it is only necessary to satisfy one of the subclauses in order to satisfy the parent clause, so as soon as one of its subclauses is fulfilled, we conclude that the parent clause is fulfilled as well. A **SEQ-refinement** means that the norm specified in the target box (*SubClause2* in Figure 3) must be fulfilled after satisfying the norm specified in the source box (*SubClause1* in Figure 3). There is another structure that can be used to model **repetition**. This structure is represented as an arrow going from a subclause to one of its ancestor clauses (or to itself), meaning the repetitive application of all the subclauses of the target clause after satisfying the source subclause. For example, on the right of Figure 3, we have an **OR-refinement** with an arrow going from *SubClause1* to *Clause*. It means that after satisfying *SubClause1* we apply *Clause* again, but not after satisfying *SubClause2*.

We follow an *ought-to-do* approach [8], that is, the normative aspects are applied over *actions* performed by the participants in the contract. We only consider the specification of *atomic actions* in the **P** field of the leaf boxes of our diagrams. We denote these actions with lower case Latin letters ("a","b","c", ...) and we use a dash ("-") to denote that there is no action specified in the no leaf boxes. The composition of actions can be achieved by means of the different kinds of refinement. In this way, an AND-refinement can be used to model *concurrency* "&" between actions, an OR-refinement can be used to model a *choice* "+" between actions, and a SEQ-refinement can be used to model *sequence* ";" of actions. The *deontic norms* that are applied over these actions can be specified in any box of our *C-O Diagrams*, affecting all the actions in the leaf boxes that are descendants of this box. If it is the case that the box where we specify the deontic norm is a leaf, the norm only affects the atomic action we have in this box. We use an upper case "*O*" to denote an obligation, an upper case "*P*" to denote a permission, and an upper case "*F*" to denote a prohibition (forbidden). These letters are written in the top left corner of field **P**. The composition of deontic norms is also achieved by means of the different refinements we have in *C-O Diagrams*. Thus, an AND-refinement corresponds to the *conjunction* operator "$\wedge$" between norms, an OR-refinement corresponds to the *choice* operator "$+$" between norms, and a SEQ-refinement corresponds to the *sequence* operator ";" between norms. Finally, concerning the specification of guard conditions and time restrictions, they affect not only the box where they are specified but also all its descendants.

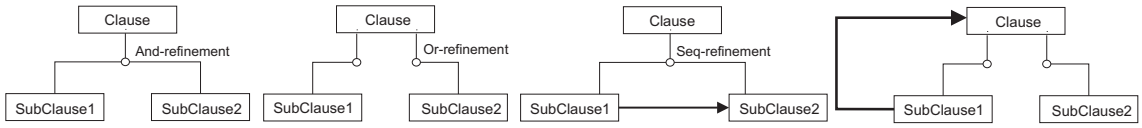

Figure 3. AND/OR/SEQ refinements and repetition in *C-O Diagrams*

We have given here a condensed description of *C-O Diagrams*. A more detail description can be found in [7], including a case study showing how to specify a concrete e-contract using our diagrams together with a qualitative and quantitative evaluation of the visual model. We formally define in what follows the syntax of *C-O Diagrams*, so we can later be able to define the translation into $\mathcal{CL}$.

We consider a finite set of real-valued variables $\mathcal{C}$ standing for clocks, a finite set of non-negative integer-valued variables $\mathcal{V}$, a finite alphabet Σ for atomic actions, a finite set of identifiers $\mathcal{A}$ for agents, and another finite set of identifiers $\mathcal{N}$ for names. The greek letter ϵ means that an expression is not given, i.e., it is empty. We use C to denote the contract modelled by a *C-O Diagram*. The diagrams syntax is defined by the following EBNF grammar:

Definition 1: (C-O Diagrams Syntax)

$$
\begin{array}{lcl}
C & := & (agent, name, g, tr, O(C_2), R) \mid \\
 & & (agent, name, g, tr, P(C_2), \epsilon) \mid \\
 & & (agent, name, g, tr, F(C_2), R) \mid \\
 & & (\epsilon, name, g, tr, C_1, \epsilon) \\
C_1 & := & C\,(And\ C)^+ \mid C\,(Or\ C)^+ \mid C\,(Seq\ C)^+ \\
C_2 & := & a \mid C_3\,(And\ C_3)^+ \mid C_3\,(Or\ C_3)^+ \mid C_3\,(Seq\ C_3)^+ \\
C_3 & := & (\epsilon, name, g, tr, C_2, \epsilon) \\
R & := & C \mid \epsilon
\end{array}
$$

□

In the above $a \in \Sigma$, $agent \in \mathcal{A}$ and $name \in \mathcal{N}$. Guard g is ϵ or a conjunctive formula of atomic constraints of the form: $v \sim n$ or $v - w \sim n$, for $v, w \in \mathcal{V}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$, whereas timed restriction tr is ϵ or a conjunctive formula of atomic constraints of the form: $x \sim n$ or $x - y \sim n$, for $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. O, P and F are the deontic operators corresponding to obligation, permission and prohibition, respectively, where $O(C_2)$ states the obligation of performing C_2, $F(C_2)$ states prohibition of performing C_2, and $P(C_2)$ states the permission of performing C_2. And, Or and Seq are the operators corresponding to the refinements we have in *C-O Diagrams*, AND-refinement, OR-refinement and SEQ-refinement, respectively.

The most simple contract we can have in *C-O Diagrams* is that composed of only one box including the elements $agent$ and $name$. Optionally, we can specify a guard g and a time restriction tr. We also have a deontic operator (O, P or F) applied over an atomic action a, and in the case of obligations and prohibitions it is possible to specify another contract C as a reparation. E.g., $C := (Buyer, Example1, \epsilon, \epsilon, O(pay), C')$ (Figure 4, on the bottom left) is a very simple contract specifying for a buyer the obligation of paying, otherwise contract C' comes into effect.

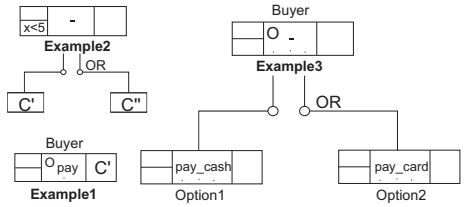

Figure 4. Syntax examples

We use C_1 to define a more complex contract where we combine different deontic norms by means of any of the different refinements we have in *C-O Diagrams*. In the box where we have the refinement into C_1 we cannot specify an agent nor a reparation because these elements are always related to a single deontic norm, but we still can specify a guard g and a time restriction tr that affect all the deontic norms we combine. E.g., $C := (\epsilon, Example2, \epsilon, x < 5, C'\ Or\ C'', \epsilon)$ (Figure 4, on the top left) is a composed contract specifying that contract C' or contract C'' must be satisfied in order to satisfy C within 5 time units.

Once we write a deontic operator in a box of our diagram, we have two possibilities as we can see in the specification of C_2: we can just write a simple action a in the same box, being the deontic operator applied only over it, or we can refine this box in order to apply the deontic operator over a compound action. In this case we have that the subboxes (C_3) cannot define a new deontic operator as it has already been defined in the parent box (affecting all the subboxes). Therefore, these subboxes cannot specify an agent nor a reparation, but it is possible to specify a guard g and a time restriction tr affecting only the action in the subbox or the action composed in its refinements. For example, $C := (Buyer, Example3, \epsilon, \epsilon, O(C'\,Or\,C''), \epsilon)$, where we have that $C' := (\epsilon, Option1, \epsilon, \epsilon, pay\_cash, \epsilon)$ and $C'' := (\epsilon, Option2, \epsilon, \epsilon, pay\_card, \epsilon)$ (Figure 4, on the right), is a contract specifying for a buyer the obligation of paying by cash or by credit card.

B. The contract language $\mathcal{CL}$

The contract language $\mathcal{CL}$ [9] enables formal specification of deontic electronic contracts. It 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). CL contracts are written using the syntax shown in Definition 2.

Definition 2: ($\mathcal{CL}$ Syntax)

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

□

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, 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. 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. In the same way, a permission clause may be an exclusive disjunction of two other permission clauses. This is interpreted as being permitted to perform one of the permissions but not both. $P(\alpha)$ is interpreted as the permission to perform α. $F_C(\alpha)$ is interpreted as forbidden to perform α and if α is performed then the reparation C must be executed. $[\beta]C$ is interpreted as if action β is performed then the contract C must be executed. The conjunction of two clauses is interpreted as both clauses have to be satisfied. ϵ is an empty action, 1 is the action that matches any action, while 0 is the impossible action. Action expressions (α and β) 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. $\overline{\ }$ is the complement, so $\overline{a}$ means "any action except a".

C. CLAN tool

CLAN [6] is a tool for $\mathcal{CL}$ contract analysis. It implements the algorithm for conflict analysis presented in [5]. Basically, conflicts in contracts arise from four different reasons. The first two reasons are being obliged and forbidden to perform the same action, and being permitted and forbidden to perform the same action. 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. The remaining two kinds of conflicts correspond to obligations of mutually exclusive actions, and permissions and obligations of mutually exclusive actions.

III. From *C-O Diagrams* to $\mathcal{CL}$

As we want to detect any conflict in the *C-O Diagrams* we model, we have to define a syntactic translation from the diagrams to $\mathcal{CL}$ in order to be able to use the CLAN tool to detect these conflicts. However, note that *C-O Diagrams* are richer than $\mathcal{CL}$ (for instance, there is no time in $\mathcal{CL}$) and thus the translation necessarily will have to abstract away some of the features of the diagrams. We comment on that later in this section.

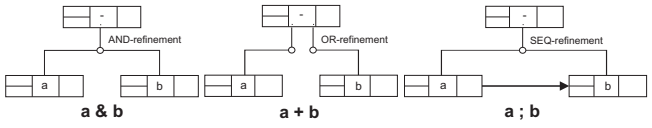

Figure 5. Equivalence of compound actions

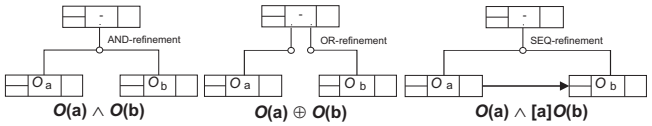

Figure 6. Equivalence of compositons of obligations

The translation of atomic and compound actions is straightforward, as we can see in Figure 5. In that figure we show the different ways we can compound two atomic actions *a* and *b* in *C-O Diagrams* and the equivalent formulas in $\mathcal{CL}$: an AND-refinement corresponds to the concurrency operator "&", an OR-refinement corresponds to the choice operator "+", and a SEQ-refinement corresponds to the sequence operator ";".

Concerning to the application of deontic norms (obligations, permissions and prohibitions) over the actions, in *C-O Diagrams* we have that when we write one of these norms in a box, it is applied over all the actions in the descendant boxes (or in the same box). Therefore, the translation into $\mathcal{CL}$ just consists of writing the composition of these actions and apply the corresponding deontic norm over it.

The translation of the different ways of composing deontic norms in *C-O Diagrams* into $\mathcal{CL}$ is a bit more complex, specially because there is not sequence operator to compose clauses in $\mathcal{CL}$ that can correspond to the SEQ-refinement in *C-O Diagrams*. The other refinements are easier to translate: an AND-refinement composing two deontic norms corresponds to the conjunction operator "$\wedge$" between these norms, while an OR-refinement corresponds to the exclusive choice operator "$\oplus$". For the SEQ-refinement between two norms we write in $\mathcal{CL}$ a conjunction between both norms, but adding the expression $[\beta]$ before the second norm, where β is the action under the first norm, so the second norm is only applied after performing this action. For example, in Figure 6 we show the translations into $\mathcal{CL}$ of the different refinements in *C-O Diagrams* applied over the obligation of performing an action *a* and the obligation of performing an action *b*.

Expressing conditions in $\mathcal{CL}$ also differs from the way we do it in *C-O Diagrams*. In the diagrams we have a field in the boxes allowing us to specify the conditions, but in $\mathcal{CL}$ this is not possible, so we use again the expression $[\beta]$, now before the conditional part of the contract, being β the condition that must be fulfilled to take this part of the contract into account.

In *C-O Diagrams* we have a field in the boxes allowing us to specify real-time restrictions in the contract. Unfor-

tunately, the specification of timing constraints is not yet supported by $\mathcal{CL}$, so what we do in this case for the translation is encoding these constraints in the definition of the actions they affect. E.g., in *C-O Diagrams* we can have a box specifying the obligation of the action "send an e-mail" and the restriction "before 2 hours" in its specific field in the box. In the corresponding $\mathcal{CL}$ translation we have to specify directly the obligation of the action "send an e-mail before two hours". We have the same situation for the agents in *C-O Diagrams*, i.e., they have to be encoded in the definition of the actions in $\mathcal{CL}$.

Finally, the translation of reparations from *C-O Diagrams* into $\mathcal{CL}$ is done directly, since in both cases we just have a reference to the secondary contract that comes into effect when the violation occurs. For that purpose, in *C-O Diagrams* we use a field in the boxes containing an obligation or a prohibition where we can specify a reparation, whereas in $\mathcal{CL}$ a reference to a reparation must be a subindex of an obligation or a prohibition expression.

Formally, we consider a parameterized translation function $\mathcal{F}_{[agent,TR]}$ from *C-O Diagrams* into expressions of $\mathcal{CL}$. The parameter $agent$ is used to propagate the entity affected by a norm, being empty if it has not been specified yet (we will write only $\mathcal{F}_{[TR]}$ in that case), and the parameter TR is used to propagate any time restriction we have specified, being initially empty. The result of applying this translation function over the diagrams is shown in Definition 3.

Definition 3: (Translation function from *C-O Diagrams* to $\mathcal{CL}$)

$$\mathcal{F}_{[TR]}((agent, name, g, tr, O(C), R)) = [g]O_R(\mathcal{F}_{[agent,TR\cup tr]}(C)) \quad (1)$$

$$\mathcal{F}_{[TR]}((agent, name, g, tr, P(C), \epsilon)) = [g]P(\mathcal{F}_{[agent,TR\cup tr]}(C)) \quad (2)$$

$$\mathcal{F}_{[TR]}((agent, name, g, tr, F(C), R)) = [g]F_R(\mathcal{F}_{[agent,TR\cup tr]}(C)) \quad (3)$$

$$\mathcal{F}_{[TR]}(\epsilon, name, g, tr, C, \epsilon) = [g]\mathcal{F}_{[TR\cup tr]}(C) \quad (4)$$

$$\mathcal{F}_{[agent,TR]}(\epsilon, name, g, tr, C, \epsilon) = [g]\mathcal{F}_{[agent,TR\cup tr]}(C) \quad (5)$$

$$\mathcal{F}_{[agent,TR]}(C_1\ And\ C_2) = \mathcal{F}_{[agent,TR]}(C_1)\ \&\ \mathcal{F}_{[agent,TR]}(C_2) \quad (6)$$

$$\mathcal{F}_{[agent,TR]}(C_1\ Or\ C_2) = \mathcal{F}_{[agent,TR]}(C_1)\ +\ \mathcal{F}_{[agent,TR]}(C_2) \quad (7)$$

$$\mathcal{F}_{[agent,TR]}(C_1\ Seq\ C_2) = \mathcal{F}_{[agent,TR]}(C_1)\ ;\ \mathcal{F}_{[agent,TR]}(C_2) \quad (8)$$

$$\mathcal{F}_{[TR]}(C_1\ And\ C_2) = \mathcal{F}_{[TR]}(C_1)\ \wedge\ \mathcal{F}_{[TR]}(C_2) \quad (9)$$

$$\mathcal{F}_{[TR]}(C_1\ Or\ C_2) = \mathcal{F}_{[TR]}(C_1)\ \oplus\ \mathcal{F}_{[TR]}(C_2) \quad (10)$$

$$\mathcal{F}_{[TR]}(C_1\ Seq\ C_2) = \mathcal{F}_{[TR]}(C_1)\ \wedge\ [\beta_{C_1}]\mathcal{F}_{[TR]}(C_2) \quad (11)$$

$$\mathcal{F}_{[agent,TR]}(a) = a_{[agent,TR]} \quad (12)$$

□

Lines (1)–(3) correspond to the translation into $\mathcal{CL}$ of a box where we specify an obligation, a permission or a prohibition, considering also any possible reparation. As agents and time restrictions are not yet supported natively by $\mathcal{CL}$, these two things are encoded as part of the actions, so we propagate them into the translation of C under the deontic norm, joining the time restriction in the box (tr) with any other time restriction we could have specified before (TR). The conditions are encoded at the beginning of the clause between square brackets, whereas the name of the box is not taken into account.

Lines (4)–(5) correspond to the translation into $\mathcal{CL}$ of a box where we do not specify any obligation, permission or prohibition. Again, agents and time restrictions are propagated into the translation of C, and the conditions are encoded at the beginning of the clause between square brackets. The difference between these two equations is that in the former $agent$ has not been specified yet, whereas in the latter $agent$ has been already specified, so we have to propagate it.

Lines (6)–(8) correspond to the translation of a refinement when the deontic operator has been already specified and the refinement is used to compose actions, while lines (9)–(11) correspond to the translation of a refinement when the deontic operator has not been specified yet and the refinement is used to compose deontic norms. The translation is quite straightforward in all the cases except in the case of (11). As we do not have a sequence operator between deontic norms in $\mathcal{CL}$, we use the conjunction operator "$\wedge$" between both contracts and we write $[\beta_{C_1}]$ before C_2 to denote that it is only applied after performing the actions in C_1. These translations are given for refinements composing only two element (C_1 and C_2) for the sake of simplicity, but they can be easily generalized for any number of elements (C_1,C_2,...,C_n).

Finally, line (12) is the translation of a simple action into $\mathcal{CL}$. In this case we just have to write the action in $\mathcal{CL}$ including in its codification any time restriction and the agent we have propagated from its ancestor boxes (denoted as $a_{[agent,TR]}$).

There are some limitations in the translation from *C-O Diagrams* into $\mathcal{CL}$, so the syntax of the diagram must be restricted when we want to do it: (i) The recursion we can have over boxes in *C-O Diagrams* cannot be translated into $\mathcal{CL}$, so we can only have recursion in guards, and (ii) the disjunction in $\mathcal{CL}$ is only allowed between obligations and permissions, so we cannot write an OR-refinement combining prohibitions nor different kinds of deontic norms if we intend to do the translation.

IV. Trading System Case Study

We consider the Trading System Case Study described in [1]. It includes the processes at a single cash desk like scanning products using a bar code scanner, as well as administrative tasks like generating reports. Several features of this case study have already been analysed with $\mathcal{CL}$ [4], so here we focus on analysing the case of adding a new feature for coupon handling in the cash desk and its evolution to a full loyalty system. In the corresponding feature diagram we just have to add two new subfeatures to the cash desk feature, coupons and loyalty cards, but our purpose is to provide a mechanism to ensure the absence of behavioural conflicts due to the addition of these new subfeatures.

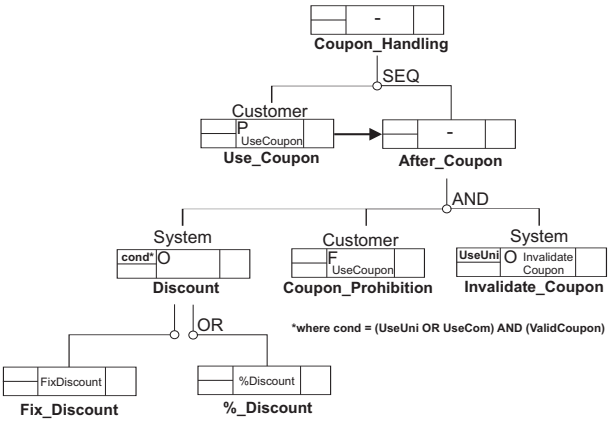

Figure 7. *C-O Diagram* corresponding to the coupon handling feature

A. Coupon Handling Feature

The addition of coupon handling is described as follows:

> "Coupons allow customers to get a discount on a purchase. The discount may either be fixed to a certain amount of money, e.g., 5 Euro, or relative to the actual amount of a purchase, e.g., 5%. At each purchase at most one coupon can be redeemed, i.e., multiple coupons cannot be combined. Coupons are *customer agnostic*, which means that they are not bound to a certain customer.
>
> Coupons have a *validity period*, which defines the days when a coupon can be redeemed by the customer. This includes a start date and an end date.
>
> Coupons can either be *unique* or *common*. Former can only be redeemed once, which is ensured by the system. Latter can be redeemed multiple times, the system only has to ensure the validity period."

As we can see, this description can be considered as a specification of the expected behaviour of the new feature. What follows we model these behavioural requirements using *C-O Diagrams* and we translate these diagrams into the contract language $\mathcal{CL}$ in order to analyse them.

The actions we define for our contract are the following:

UseCoupon = Customer redeems a coupon
UseUni = Customer has redeemed a unique coupon
UseCom = Customer has redeemed a common coupon
ValidCoupon = System checks the coupon validity period is correct
FixDiscount = System applies a fixed discount on the purchase
%Discount = System applies a relative discount on the purchase
InvalidateCoupon = System invalidates a coupon

Therefore, from the given description of the coupon handling feature we create the *C-O Diagram* shown in Figure 7. This diagram is translated into the following clauses in $\mathcal{CL}$ (considering an implicit $\wedge$ operator between them):

1. $[1^*]P(UseCoupon)$
2. $[1^*][UseCoupon][UseUni + UseCom]$
   $[ValidCoupon]O(FixDiscount + \%Discount)$
3. $[1^*][UseCoupon]F(UseCoupon)$
4. $[1^*][UseCoupon][UseUni]O(InvalidateCoupon)$

In all these clauses the expression $[1^*]$ should be interpreted as *'At any time'*. The first clause says that the customer is allowed to redeem a coupon. The second clause states that after redeeming a coupon, either unique or common, if the validity period is correct the system is obliged to applied a discount either fixed or relative to the actual amount of the purchase. The third clause specifies that multiple coupons cannot be combined, that is, after redeeming a coupon it is forbidden to redeem another coupon. The last clause states that the system has to invalidate a unique coupon after redeeming it. In the analysis we also include in the specification that *FixDiscount* and *%Discount* are mutually exclusive actions, because they cannot be performed at the same instant of time.

Figure 8. New specification of the **Use_Coupon** clause

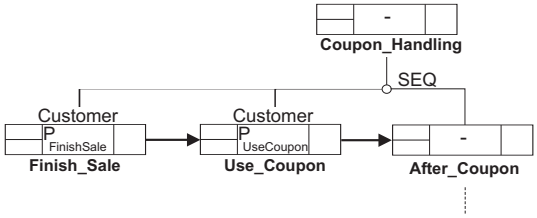

Figure 9. Coupon handling feature *C-O Diagram* modified

Analysing this contract in CLAN we obtain that it is not conflict-free. The problem is the concurrent permission and prohibition of performing action *UseCoupon*. It can be fixed by modifying the clause **Use_Coupon** in the way we can see in Figure 8, so now the corresponding $\mathcal{CL}$ clause is:

$1'$. $[\overline{UseCoupon}^*]P(UseCoupon)$

It was a problem of *underspecification* in the description, because it says that *'coupons allow customers to get a discount on a purchase'* but says nothing about the only coupon restriction in this sentence. The new clause can be interpreted as *'coupons allow customers to get a discount on a purchase if no coupon has been used before'*.

Another problematic point is that in the current specification we are allowing to redeem a coupon at any time except after redeeming another coupon, and this is not what we desire. For example, if we consider an action *EnterItem* (the customer enters a new item in the cash desk), it will be permitted to use the coupon before that, but we want the customer to use the coupon only after entering all the items in the cash desk to apply the discount on the final price. Once again we are facing a problem of *underspecification*, because the description of the new feature must include the specification of the moment at which coupons can be redeemed in the purchase process.

Hence we consider one more action that customer executes just before being allowed to redeem the coupon:

FinishSale = Customer stops entering items and starts payment procedure

Now the specification of the behaviour of the coupon handling feature in *C-O Diagrams* is modified as shown in Figure 9 and the translation into $\mathcal{CL}$ is the following (where clauses 2., 3. and 4. remain the same as before):

$1''$. $[1^*][FinishSale][\overline{UseCoupon}]P(UseCoupon)$
...
5. $[1^*]P(FinishSale)$

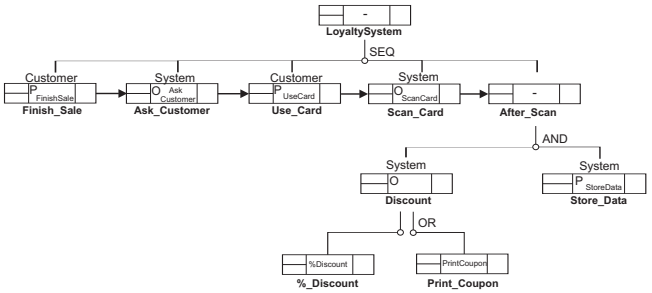

Figure 10. *C-O Diagram* corresponding to the loyalty card feature

We add one more clause to allow the customer to stop entering items and start payment procedure at any moment (**Finish_Sale**). Only after performing the action in this clause the customer is allowed to use a coupon. In this case, when we analyse the new specification with the CLAN tool, we obtain that the contract is conflict-free.

B. Loyalty System

In this subsection we focus on how the loyalty system is integrated in the cash desk, so we do not care about how loyalty cards are created, we just analyse how they are processed and the impact they have in the coupon handling.

The evolution of the coupon handling feature into a full loyalty system is described as follows:

> "In a loyalty program, customers can have a special *loyalty card*, which they can use for each purchase. Customers can then get discounts, which can be based on their individual shopping behaviour.
>
> Whenever the customer does a purchase, he is asked for the loyalty card, which is then scanned by the system. After the payment of the purchase has finished, the corresponding information is stored in the system.
>
> The discount provided to the customer can be given in various different ways. There could be a relative discount of 1% for every purchase, which is directly applied to the current purchase. Another possibility is to print out an *individual* coupon for the customer, which he can redeem on the next purchase. This coupon is bound to the customer and can only be redeemed together with the corresponding loyalty card. Like customer agnostic coupons, these coupons also have a validity period"

First of all, we want to analyse independently the description of this new feature, so we model again the behavioural requirements using *C-O Diagrams*. Like in the previous case, the description does not include the specification of the moment at which loyalty cards can be used in the purchase process, but in this case we include the action customer executes just before being allowed to use the loyalty card from the beginning. Therefore, the actions we consider are the following:

FinishSale = Customer stops entering items and starts payment procedure
AskCustomer = System asks customer for the loyalty card
UseCard = Customer uses the loyalty card
ScanCard = System scans loyalty card
StoreInfo = System stores information about the customer purchase
%Discount = System applies a relative discount on the purchase
PrintCoupon = System prints out an individual coupon for the customer

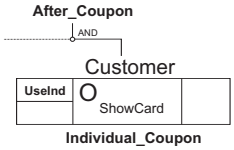

Figure 11. Individual coupon management in coupon handling feature

Once we have defined the actions, from the above description of the full loyalty system we infer the *C-O Diagram* shown in Figure 10, which is translated into the following $\mathcal{CL}$ clauses:

1. $[1^*][FinishSale]O(AskCustomer)$
2. $[1^*][AskCustomer]P(UseCard)$
3. $[1^*][UseCard]O(ScanCard)$
4. $[1^*][ScanCard]P(StoreData)$
5. $[1^*][ScanCard]O(\%Discount + PrintCoupon)$
6. $[1^*]P(FinishSale)$

The first clause says that the system has to ask the customer for the loyalty card when he stops entering items. The second clause states that the customer is allowed to use the loyalty card after being asked by the system. The third clause says that when the customer uses the loyalty card, the system has to scan the card. After scanning the card, the fourth clause states that the system is allowed to store data about the client purchase and the fifth clause states that the system is obliged to apply a relative discount on the purchase or to print out an individual coupon for the customer. Finally, the last clause is used again to allow the customer to stop entering items and start the payment procedure.

We only have obligations and permissions in this specification, so we cannot find out conflicts of the form of having obligation (or permission) and prohibition on the same action. However, *%Discount* and *PrintCoupon* are mutually exclusive actions, so we check with CLAN that we do not have the permission or obligation to perform these actions at the same time. We obtain that this contract is conflict-free.

Nevertheless, the inclusion of the full loyalty system also implies an evolution of the coupon handling feature. It is extended with individual coupons bound to a certain customer, which must be handled without affecting the already existing customer agnostic coupons. For that purpose, we define first two new actions associated to the coupon handling feature:

UseInd = Customer has redeemed an individual coupon
ShowCard = Customer shows his loyalty card

Then, in order to analyse the impact of the new individual coupons on the behaviour of coupon handling feature, we add to the *C-O Diagram* we consider at the end of the subsection before a new clause called **Individual_Coupon** saying that the customer has the obligation to show his loyalty card after redeeming an individual coupon (Figure 11), which corresponding $\mathcal{CL}$ clause is the following:

6. $[1^*][UseCoupon][UseInd]O(ShowCard)$

This is the only extension we must add to the contract as the other characteristics of the coupon handling feature remain invariable. When we analyse the contract again

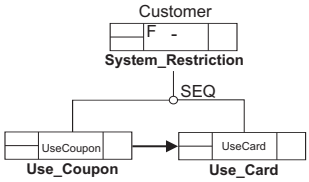

Figure 12. *C-O Diagram* modelling the new restriction

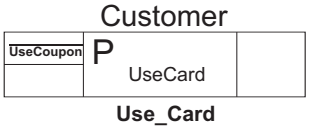

Figure 13. New specification of the **Use_Card** clause

including the new clause we obtain that it is still conflict-free.

To conclude, we must analyse that the composition of both features, coupon handling and loyalty cards, is free of conflicts. We consider an additional restriction stating that the customer is not allowed to use a loyalty card after redeeming a coupon, in order to avoid cumulative discounts. This new restriction is modelled with the *C-O Diagram* of Figure 12, which corresponding clause in $\mathcal{CL}$ is as follows:

-. $[1^*]F(UseCoupon.UseCard)$

Therefore, we analyse the composition by putting all the clauses together: the six clauses corresponding to the coupon handling feature (including individual coupons), the five first clauses corresponding to the loyalty card feature (the sixth clause is already included), and the new clause corresponding to the restriction we have just defined. Using the CLAN tool we obtain that the contract has a conflict due to the permission of performing actions *UseCoupon* and *UseCard* in sequence, which is in conflict with the new added restriction. This conflict can be solved by changing clause **Use_Card** as we can see in Figure 13, so the corresponding $\mathcal{CL}$ clause is now the following:

$2'$. $[1^*][AskCustomer][\overline{UseCoupon}]P(UseCard)$

In this new specification we say that, after being asked by the system, the client can use his loyalty card only if he has not redeemed a coupon before. Now we obtain that the contract is conflict-free.

V. Conclusions

We have presented here a framework to handle variability and evolution in SPL, and we have used a case study to show how adding new features may introduce conflicts and how these conflicts may be detected. While the use of *C-O Diagrams* seems interesting due to its modularity, analysing conflicts is not easy. On the other hand, the use of logic ($\mathcal{CL}$) allows us to automatically detect conflicts, but its use as a modelling language is not advisable as it is not easy in general to add a feature in a local manner, as done with the diagrams. This drawback is partially overcome by modelling features behaviour in *C-O Diagrams* and then translating them into $\mathcal{CL}$ to detect potential conflicts.

As shown in the case study it is not easy to prove conflict-freedom in a modular manner, as adding a new feature may indeed be in conflict with a previous feature, needing a global conflict analysis. Compositional (modular) verification of SPL is an interesting and challenging research area.

As future work, it remains a study of the scalability of the framework to see how complex can be the cases that we can tackle with our approach.

Acknowledgment

Partially supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2009-14312-C02-02, the JCCLM regional project PEII09-0232-7745, and the Nordunet3 project “COSoDIS”. The first author is supported by the European Social Fund and the JCCLM. We would also like to thank Ina Schaefer for suggesting the case study that appears in Section IV.

References

[1] Highly Adaptable and Trustworthy Software using Formal Methods Project (HATS). Deliverable D5.1. Requirements Elicitation.

[2] P. Asirelli, M.H. ter Beek, S. Gnesi, and A. Fantechi. A deontic logical framework for modelling product families. *Proceedings of 4th International Workshop on Variability Modelling of Software-intensive Systems*, pages 37–44, 2010.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[4] S. Fenech, G.J. Pace, J.C. Okika, A.P. Ravn, and G. Schneider. On the Specification of Full Contracts. In *Proceedings of the Sixth International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2009)*, pages 39–55, 2009.

[5] S. Fenech, G.J. Pace, and G. Schneider. Automatic Conflict Detection on Contracts. In *ICTAC09, LNCS*. Springer, 2009.

[6] S. Fenech, G.J. Pace, and G. Schneider. Clan: A tool for contract analysis and conflict discovery. In *7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799 of *LNCS*, pages 90–96, Macao, China, October 2009. Springer. CLAN is available from http://www.cs.um.edu.mt/svrg/.

[7] E. Martínez, G. Díaz, M. E. Cambronero, and G. Schneider. A Formal Model for Visual Specification of e-Contracts. *Proceedings of 7th IEEE 2010 International Conference on Services Computing (SCC 2010)*, pages 1–8, 2010.

[8] G.J. Pace and G. Schneider. Challenges in the specification of full contracts. *Proceedings of 7th International Conference on integrated Formal Methods*, pages 292–306, 2009.

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

[10] P.Y. Schobbens, P. Heymans, J.C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, 2006.