An Approach for Validating BCL Contract Specifications

Guido Governatori
School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, QLD 4072, Australia
email: guido@itee.uq.edu.au

Zoran Milosevic
CRC for Enterprise Distributed Systems Technology
Brisbane, QLD 4072, Australia
email: zoran@dstc.edu.au

Abstract

We continue the study, started in [5], on the formal relationships between a domain specific contract language (BCL) and the logic of violation (FCL) proposed in [6, 7]. We discuss the use of logical methods for the representation and analysis of business contracts. The proposed analysis is based on the notions of normal and canonical forms of contracts expressed in FCL. Finally we present a mapping from FCL to BCL that can be used to provide an executable model of a formal representation of a contract.

1. Introduction

Business Contract Language (BCL) is a domain specific language developed for the purpose of specifying contract conditions in a way suitable for their run-time evaluation as the corresponding business processes are executed. The development of the language followed a pragmatic path. We have considered many contracting scenarios while taking into account the modelling frameworks for the description of policies and communities [15, 14]. In our recent work, we have begun checking the consistency of the language by considering various formalisms such as the formal contract logic (FCL) [5]. In particular, we have considered a subset of BCL concepts and have used FCL as a basis to check their soundness and thus establish the semantic foundation for that part of the language.

While we are continuing with the process of analysing the properties and well-formedness of rest for the language, we are also developing a framework and a set of tools that allows checking of the validity of BCL specifications themselves. In fact, the tools being developed are aimed at producing a normal form of a contract in FCL based on the contract expression in natural language. The normal form of a contract is one in which conflicts are detected, inconsistencies are removed and redundancies eliminated. The premise of this paper is that, if we establish a mapping from BCL to a theory based on FCL, then this mapping and the tools developed in the FCL context, can be used to check the validity of BCL specifications. Another approach to the checking of BCL validity was presented in our earlier work [16].

The accuracy of the representation can affect the performance of the monitoring. Given the inherent complexity of contracts it is possible that BCL specifications may suffer from inconsistency, conflicts, redundancies or missing conditions. The aim of this paper is to propose a mapping from BCL to FCL and to show how this mapping, augmented with the FCL normalisation tools, can be used to address the formal validation of BCL specifications.

Next section provides a brief description of BCL. In Section 3 we discuss some issue related to the formalisation of contract in FCL. Section 4 outlines the FCL normalisation tools. Section 5 presents mapping from BCL to FCL constructs. The paper concludes with a list of discussion points.

2. BCL

Business Contract Language (BCL) has been described in a number of recent publications involving one of the authors [16, 17, 14] and this section only highlights its key features to ensure self-containment of this paper. This section is based on [5]

In fact, the BCL language consists of several related sub-languages and this section will only briefly present policy sub-language. When submitted to a monitoring engine that implements run time semantics of contract execution, this language can be interpreted according to the events that trigger policy activation and evaluations.

A *Policy* specifies business-level constraints explicitly associated with a specific *Role* and the type of constraint is is further refined using a *Modality*, indicating whether it is an obligation, permission or prohibition.

The behaviour associated with a policy is expressed in terms of events and relationships between events related to the policy. An event can represent actions of parties to the contract, some external event from the environment, or a temporal event.

This expression states a normative constraint that applies to the role in question, for example, the obligation of

the supplier to make sure the goods are available within one day of receipt of a purchase order issued by the purchaser. Thus the event pattern specifies all the events that constitute a normative constraint, including those that effectively trigger this policy, and that may originate from an external source, such as other party or timeout event. Although the event pattern is sufficient to express behavioural constraint in the policy, it may be useful for a policy specifier, to extract triggering information from the event pattern. We refer to that part of behaviour expression as trigger. So, in the example, the triggering event is *PurchaseOrder*. In some cases, policy can become active as soon as the system that implements the policy is activated. In this case trigger corresponds to the *SystemStart*.

A generic form of BCL policy is:

```
Policy:
  Role:
  Modality:
  Trigger:
  Behaviour:
    EventPattern
```

For example,

```
Policy: MakeGoodsAvailable
  Role: Supplier
  Modality: Obligation
  Trigger: PurchaseOrder
  Behaviour:
    GoodsAvailable.date before (PurchaseOrder.date + 1 days)
```

The policy specifies the goods availability behaviour condition as an event pattern. In this case the event pattern is satisfied if the *GoodsAvailable* event generated by the *Supplier* is at most one day after the *PurchaseOrder* event was received. This matching is done by checking the date parameters of both events. The satisfaction of event patterns means that this obligation policy is satisfied. The triggering condition for this policy is occurrence of *PurchaseOrder* event.

BCL also supports expression of guarded conditions applicable to the BCL event pattern and a number of language elements that contain event patterns such as policies, state updates, and notification generation. In general the BCL guard specifies precondition for the evaluation of the corresponding element. For example, guard can be used to specify when a policy is to be applied such as in the example below:

```
Policy: MaintenanceSupplierIT
  Role: Supplier
  Modality: Prohibition
  Trigger: SystemStart
  Guard: on weekday
    Behaviour:
        ITMaintenance
```

Note that in this case the guard effectively 'triggers' the policy as this policy is in force at all times during this system life-time (unless it is subsequently changed).

This states that the policy will be active for the monitoring purpose only on weekdays (i.e., when its guard condition is true).

A particular use of guards can be to specify conditions for the activation of reparation or contrary-to-duty policies. This will be discussed in detail in next sections and here we only provide some examples to facilitate subsequent discussions.

Take for example, the following policy expression (note that for simplicity we do not elaborate on the exact meaning of the *QualityOfServiceAgreement* condition).

```
Policy: QualityOfServicePolicy
  Role: Supplier
  Modality: Obligation
  Trigger: SystemStart
  Behaviour:
    QualityOfServiceAgreement at http://suplier/qos1.htm
```

When a service does not satisfy this condition a violation event (*QualityOfServicePolicyViolated*) is to be generated indicating that this obligation is violated. This condition can then be used in an expression of a guard for a policy that applies under these circumstances, namely:

```
Policy: Replace3daysPolicy
  Role: Supplier
  Modality: Obligation
  Guard: HasOccurred QualityOfServicePolicyViolated
  Behaviour:
    now + 3 days
```

This new policy will be activated for the monitoring when *QualityOfServicePolicyViolated* guard was true, i.e., when the violation event of *QualityOfServicePolicy* was detected. For the detection of this event we use *HasOccurred* event pattern expression where *QualityOfServicePolicyViolated* event is an input parameter and the result is Boolean. From that point in time the *Replace3daysPolicy* will need to be monitored to establish whether *Supplier* has fulfilled its contrary-to-duty obligation. Similarly, the violation of this policy could trigger yet another obligation for the *Supplier*, namely that is shall refund the purchaser and pay a penalty of $1000. This last policy is not shown in this example.

3. Formal Representation of Contracts

Business contracts are agreements between two or more parties specifying the obligations, permissions and prohibitions including the actions and the penalties that may be

taken in the case when any of the stated conditions are not being met.

The clauses of a contract are usually expressed in a codified or specialised natural language, e.g., legal English. At times this natural language is, by its nature, imprecise and ambiguous. However, if we want to monitor the execution and performance of a contract, ambiguities must be avoided or at least the conflicts arising from them resolved. In addition conditions influencing the expected behaviour of the parties can be specified in different documents and can be subject to the legislation currently in force. A further issue is that often the clauses in a contract show some mutual interdependencies and it might not be evident how to disentangle such relationships. To implement an automated monitoring system all the above issues must be addressed.

To address some of these issues we propose a formal representation of contracts. A language for specifying contracts needs to be formal, in the sense that its syntax and its semantics should be precisely defined. This ensures that the protocols and strategies can be interpreted unambiguously (both by machines and human beings) and that they are both predictable and explainable. In addition, a formal foundation is a prerequisite for verification or validation purposes. One of the main benefits of this approach is that we can use formal methods to reason with and about the clauses of a contract. In particular we can

- analyse the expected behaviour of the signatories in a precise way, and
- identify and make evident the mutual relationships among various clauses in a contract.

Secondly, a language for contracts should be conceptual. According to the *Conceptualization Principle* of [9], this means that the language should allow their users to focus only and exclusively on aspects related to content of the contract, without having to deal with any aspects related to their implementation. As stated in [9], examples of conceptually irrelevant aspects are, e.g., aspects of (external or internal) data representation, physical data organisation and access, as well as all aspects related to platform heterogeneity (e.g., message-passing formats).

Every contract contains provisions about the obligations, permissions, entitlements and others mutual normative positions that the signatories of the contract subscribe to. Therefore a formal language intended to represent contracts should provide notions closely related to the above concepts. Since the seminal work by Lee [13] Deontic Logic has been regarded as one on the most prominent paradigms to formalise contracts.

3.1. Obligations, Violations and CTD

Deontic Logic extends classical logic with the modal operators O, P and F. Thus, for example the interpretation of the formulas OA, PA and FA are, respectively, that A is obligatory, A is permitted and A is forbidden. A full characterisation of the deontic operators is not crucial in this paper. All we need is that the deontic operators obey the usual mutual relationships, i.e.,

$$OA \equiv \neg P \neg A \quad \neg O \neg A \equiv PA \quad O \neg A \equiv FA \quad \neg PA \equiv FA$$

and are closed under logical equivalence, i.e., if $A \equiv B$ then $OA \equiv OB$, and satisfy the axiom $OA \rightarrow PA$ (i.e., if A is obligatory, then A is permitted) that implies the internal coherency of the obligations in a contracts, or, in other words, it is possible to execute obligations without doing something that is forbidden.

The obligations in a contract, as well as the other normative positions that eventually appear in contracts apply to the signatories of the contract. To capture this we will consider directed deontic operators [11]; i.e., the deontic operators will be labelled with the subject of deontic modality. In this perspective the intuitive reading of the expression $O_s A$ is that s has the obligation to do A, or that A is obligatory for s.

Further contracts usually specify actions to be taken in case of breaches of the contract (or part of it). These can vary from (pecuniary) penalties to the termination of the contract itself. This type of construction, i.e., obligations in force after some other obligations have been violated, is know in the deontic literature as contrary-to-duty obligations (CTD) or reparational obligations (because they are activated when normative violations occur and are meant to 'repair' violations of primary obligations [2]). Thus a contrary-to-duty is a conditional obligation arising in response to a violation, where a violation is signalled by an unfulfilled obligation. The ability do deal with violations or potential violations and the reparational obligation generated from them is one of the essential requirements for reasoning about and monitoring the implementation and performance of business contracts.

The idea behind the logic of violation [6, 7] is that the meaning of a clause of a contract (or, in general a norm in a normative system) cannot be taken in isolation: it depends on the context where the clause is embedded in (the contract). For example a violation cannot exist without an obligation to be violated. The second aspect we have to consider is that a contract is a finite set of explicitly given clauses and, often, some other clauses are implicit (or can be derived) from the already given clauses. The ability to extract all the implicit clauses from a contract is of paramount importance for the monitoring of it; otherwise some aspects of the contract could be missing from

its implementation. Accordingly a logic of violation to be useful for the monitoring and analysis of a contract should provide facilities to

1. relate interdependent clauses of a contract and
2. extract or generate all the clauses (implicit or explicit) of a contract.

As we have just discussed a violation cannot exist without an obligation to be violated. Thus we have a sequential order among an obligation, its violation and eventually an obligation generated in response to the violation and so on. To capture this intuition we introduce the non-boolean connective $\otimes$, whose interpretation is such that $OA \otimes OB$ is read as "OB is the reparation of the violation of OA" (we will refer to formulas built using $\otimes$ as $\otimes$-expressions); in other words the interpretation of $OA \otimes OB$, is that A is obligatory, but if the obligation OA is not fulfilled (i.e., when $\neg A$ is the case, and consequently we have a violation of the obligation OA), then the obligation OB is in force. The above interpretation shows that violations are special kinds of exceptions [6, 7], and several authors have used exceptions to raise conditions to repair a violation in the context of contract monitoring [17, 10].

3.2. FCL

We now introduce the logic (FCL) we will use to reason about contracts. The language of FCL consists of two set of atomic symbols: a numerable set of propositional letters $p, q, r, \ldots$, intended to represent the state variables of a contract and a numerable set of event symbols $\alpha, \beta, \gamma, \ldots$ corresponding to the relevant events in a contract. Formulas of the logic are constructed using the deontic operators O (for obligation), P (for permission), negation $\neg$ and the non-boolean connective $\otimes$ (for the CTD operator). The formulas of FCL will be constructed in two steps according to the following formation rules:

- every propositional letter is a literal;
- every event symbol is a literal;
- the negation of a literal is a literal;
- if X is a deontic operator and l is a literal then Xl and $\neg Xl$ are modal literals.

After we have defined the notion of literal and modal literal we can use the following set of formation rules to introduce $\otimes$-expressions, i.e., the formulas used to encode chains of obligations and violations.

- every modal literal is an $\otimes$-expression;
- if $Ol_1, \ldots, Ol_n$ are modal literals and l_{n+1} is a literal, then $Ol_1 \otimes \ldots \otimes Ol_n$ and $Ol_1 \otimes \ldots \otimes Ol_n \otimes Pl_{n+1}$ are $\otimes$-expressions.

Each condition or policy of a contract is represented by a rule in FCL, where a rule is an expression

$$r : A_1, \ldots, A_n \vdash C$$

where r is the name/id of the policy, $A_1, \ldots, A_n$, the *antecedent* of the rule, is the set of the premises of the rule (alternatively it can be understood as the conjunction of all the literals in it) and C is the conclusion of the rule . Each A_i is either a literal or a modal literal and C is an $\otimes$-expression.

The meaning of a rule is that the normative position (obligation, permission, prohibition) represented by the conclusion of the rule is in force when all the premises of the rule hold.

Thus, for example, the second part of clause 5.1 of the contract ("the supplier shall refund the purchaser and pay a penalty of $1000 in case she does not replace within 3 days a service that do not conform with the published standards") can be represented as

$$r : \neg p, \neg \alpha \vdash O_{Supplier} \beta$$

where p is propositional letter meaning that "a service has been provided according to the published standards", α is the event symbol corresponding to the event "replacement occurred within 3 days", and β is the event symbol corresponding to the event "refund the customer and pay her the penalty". The policy is activated, i.e., the supplier is obliged to refund the customer and pay her a penalty of $1000, when the condition $\neg p$ is true (i.e., we have a faulty service), and the event "replacement occurred within 3 days" lapsed, i.e., its negation occurred.

The connective $\otimes$ permits combining primary and CTD obligations into unique regulations. The operator $\otimes$ is such that $\neg\neg A \equiv A$ for any formula A and enjoys the properties of associativity

$$A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C,$$

duplication and contraction on the right,

$$A \otimes B \otimes A \equiv A \otimes B.$$

The right-hand side of the equivalence above states that B is the reparation of the violation of the obligation A. That is, B is in force when $\neg A$ is the case. For the left-had side we have that, as before, a violation of A, i.e., $\neg A$, generates a reparational obligation B, and then the violation of B can be repaired by A. However, this is not possible since we already have $\neg A$.

The formation rules for $\otimes$-expressions allows a permission to occur only at the end of such expression. This is due

to fact that a permission can be used in a reparation of a violation, but it is not possible to have violation of a permission, thus it makes no sense to have reparations to permission. Sometimes contracts contain other mutual normative positions such as delegations, empowerment, rights and so. Very often these notions can be effectively represented in terms of complex combinations of directed obligations and permissions [3]. Hence violations to such complex notions result in violations to the obligations describing such notions.

4. Normalisation tools for FCL

In the previous section we presented a formalism suitable to represent contract when contracts are understood as sets of "normative" policies. Here we will examine how the formalism can be used to analyse contracts and to reason about them.

As we have argued in Section 3 the aim of a formal representation is to provide facilities for a precise and unambiguous description of the specification of a contract. However, due to their nature, contracts are often complex documents written in (a specialised) natural language. Accordingly it is possible that two contract domain engineers might come up with different representations for one and the same contract –this might also be the case when one designer formalises a (part of) contract at different times. Therefore there is the need to compare two different versions of the same contract to determine whether they are equivalent. Versions of a contract can also be obtained from different drafts of a contract –for example in the contract negotiation phase when the parties involved in the negotiation propose their drafts of the contract to be concluded– and one wants to know whether two versions produce the same effects or the versions are compatible with each other. To this end we will introduce transformations on the formal representations of a contract (FCL) to produce a normal form of the same (NFCL). When normal forms are generated we can compare them for equivalence and compatibility. This is possible since normal form contains all contract conditions that can be generated/derived from the conditions explicitly given in the formal representation of the contract.

Normal forms are also beneficial in other respects. They can be used to identify formal loopholes, deadlocks and inconsistencies in a contract. A domain expert can examine the normal form of a formal representation of a contract to determine the correctness and completeness of the formal representation of the contract and eventually to discover semantic drawbacks of the contract. A complete and correct normal form is called a canonical form of the contract (CFCL). Since canonical forms are complete and hence contain all conditions of a contract they can be mapped to a BCL representation, aimed at the implementation and monitoring of the contract using the mapping from FCL to BCL given in [5].

Notice that there can be many normal forms for a contract, but there is only one canonical form of a contract, since normal forms are the expansions of (potentially partial) formal specifications of a contract. The idea is that a normal form is the closure under some logical operations of a fragment of a contract, while the canonical form is the closure of all fragments of a contract (fragments can overlap).

This means that, in order to improve the quality and completeness of the representation, it is possible to compare and integrate normal forms obtained from different contract designers.

Figure 1 illustrates a scenario where there are two equivalent (formal) versions of the contract FCL_1 and FCL_2. The two versions are equivalent since they produce the same normal form ($NFCL_1$). On the other side FCL_3 corresponds to a normal form that does not coincide with $NFCL_1$. Thus we can compare and integrate the two normal forms to produce the canonical form of the contract CFCL, which in turn is mapped to an executable BCL program.

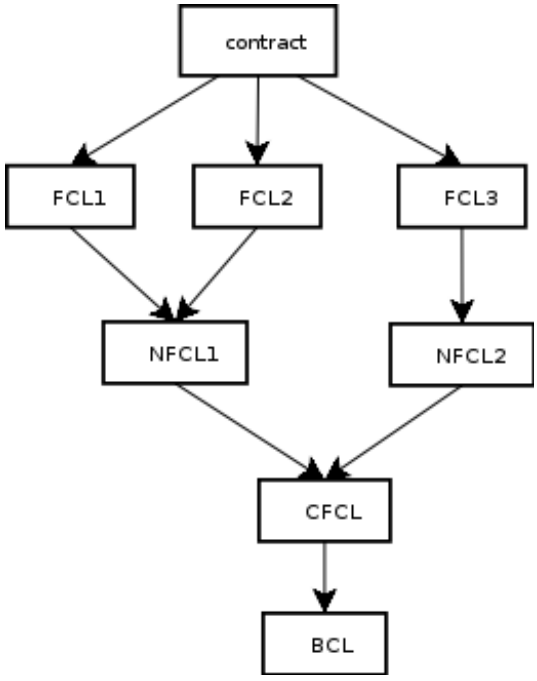

Figure 1. FCL Normalisation Process

In the rest of this section we introduce the procedures to generate normal forms. First (Section 4.1) we describe a mechanism to derive new contract conditions by merging together existing contract clauses. In particular we link an obligation and the obligations triggered in response to vio-

lations of the obligation. Then, in Section 4.2, we examine the problem of redundancies, and we give a condition to identify and remove redundancies from the formal specification of a contract. Finally in Section 4.1 we consider the issue of normative conflicts in contracts. More precisely we define when two contract clauses are mutually inconsistent and we briefly discuss two possible alternatives to deal with such cases.

4.1. Merging Contract Conditions

One of the features of the logic of violation is to take two rules, or clauses in a contract, and merge them into a new clause. In what follows we will first examine some common patterns of this kind of construction and then we will show how to generalise them.

Consider a policy like (in what follows Γ and Δ are sets of premises)

$$\Gamma \vdash O_s A.$$

Given an obligation like this, if we have that

$$\Delta, \neg A \vdash O_{s'} C,$$

then the latter must be a good candidate as reparational obligation of the former. This idea is formalised is as follows:

$$\frac{\Gamma \vdash O_s A \qquad \Delta, \neg A \vdash O_{s'} C}{\Gamma, \Delta \vdash O_s A \otimes O_{s'} C}$$

This reads as if there exists a conditional obligation whose antecedent is the negation of the propositional content of a different norm, then the latter is a reparational obligation of the former. In this way, the CTD obligation can be forced to be an *explicit reparational obligation* with respect to the violation of its primary counterpart. Accordingly, it seems reasonable to discard both premises when they are subsumed by the conclusion. Their reciprocal interplay makes them two related norms so that they cannot be viewed anymore as independent obligations. Notice that the subjects and beneficiaries of the primary obligation and its reparation can be different, even if very often in contracts they are the same.

Suppose the contract includes

$$r : \mathit{Invoice} \vdash O_{\mathit{Purchaser}} \mathit{PayWithin7Days}$$

and

$$r' : \neg \mathit{PayWithin7Days} \vdash O_{\mathit{Purchaser}} \mathit{PayWithInterest}.$$

From these we obtain

$$r'' : \mathit{Invoice} \vdash O_{\mathit{Purchaser}} \mathit{PayWithin7Days} \otimes O_{\mathit{Purchaser}} \mathit{PayWithInterest}.$$

The schema in (4.1) can also generate chains of CTDs in order to deal iteratively with violations of reparational obligations. The following case is just an example of this process.

$$\frac{\Gamma \vdash O_s A \otimes O_s B \qquad \neg A, \neg B \vdash O_s C}{\Gamma \vdash O_s A \otimes O_s B \otimes O_s C}$$

For example consider a contract for service containing the following clauses

> 5.1 The (Supplier) shall ensure that the (Services) are available to the (Purchaser) under Quality of Service Agreement (http://supplier/qos1.htm). (Services) that do not conform to the Quality of Service Agreement shall be replaced by the (Supplier) within 3 days from the notification by the (Purchaser).
>
> 5.2 If for any reason the conditions stated in 5.1 are not meet, the (Supplier) shall refund the (Purchaser) and pay the (Purchaser) a penalty of $1000.

The above two clauses can be represented by the following two rules

$$r : \mathit{Invoice} \vdash O_{\mathit{Supplier}} \mathit{QualityOfService} \otimes O_{\mathit{Supplier}} \mathit{Replace3days}$$

and

$$r' : \neg \mathit{QualityOfService}, \neg \mathit{Replace3days} \vdash O_{\mathit{Supplier}} \mathit{Refund\&Penalty}$$

from which we derive the new rule

$$r'' : \mathit{Invoice} \vdash O_{\mathit{Supplier}} \mathit{QualityOfService} \otimes O_{\mathit{Supplier}} \mathit{Replace3days} \otimes O_{\mathit{Supplier}} \mathit{Refund\&Penalty}.$$

The above patterns are just special instances of the general mechanism described by the following inference mechanism

$$\frac{r : \Gamma \vdash O_s A \otimes (\bigotimes_{i=1}^{n} O_s B_i) \otimes O_s C \quad r' : \Delta, \neg B_1, \ldots, \neg B_n \vdash \mathbf{X}_s D}{r'' : \Gamma, \Delta \vdash O_s A \otimes (\bigotimes_{i=1}^{n} O_s B_i) \otimes \mathbf{X}_s D}$$

where $\mathbf{X}$ denotes either an obligation or a permission. In this last case, we will impose that D is an atom. Since the minor premise states that $\mathbf{X}_s D$ is a reparation for $O_s B_n$, i.e., the last literal in the sequence $\bigotimes_{i=1}^{n} O_s B_i$, we can attach $\mathbf{X}_s D$ to such sequence.

4.2. Removing Redundancies

Given the structure of the inference mechanism it is possible to combine rules in slightly different ways, and in some cases the meaning of the rules resulting from such operations is already covered by other rules in the contract. In other cases the rules resulting from the merging operation are generalisations of the rules used to produce them, consequently, the original rules are no longer needed in the contract. Thus some clauses can be removed from the contract without changing the meaning of it. To deal with this issue we introduce the notion of subsumption between rules. Intuitively a rule subsumes a second rule when the behaviour of the second rule is implied by the first rule.

We first introduce the idea with the help of some example and then we show how to give a precise formal definition of the notion of subsumption appropriate for FCL.

Let us consider the rules

$$\begin{aligned} r : \mathit{Invoice} \vdash\ & O_{\mathit{Supplier}}\mathit{QualityOfService} \otimes \\ & O_{\mathit{Supplier}}\mathit{Replace3days} \otimes \\ & O_{\mathit{Supplier}}\mathit{Refund\&Penalty}, \\ r' : \mathit{Invoice} \vdash\ & O_{\mathit{Supplier}}\mathit{QualityOfService} \otimes \\ & O_{\mathit{Supplier}}\mathit{Replace3days}. \end{aligned}$$

The first rule, r, subsumes the second r'. Both rules state that after the seller has sent an invoice she has the obligation to provide goods according to the published standards, and if she fails to do so –i.e., if she violates such an obligation–, then the violation of *QualityOfService* can be repaired by replacing the faulty goods within three days ($O_{\mathit{Supplier}}\mathit{Replace3days}$). In other words $O_{\mathit{Supplier}}\mathit{Replace3days}$ is a secondary obligation arising from the violation of the primary obligation $O_{\mathit{Supplier}}\mathit{QualityOfService}$. In addition r prescribes that the violation of the secondary obligation $O_{\mathit{Supplier}}\mathit{Replace3days}$ can be repaired by $O_{\mathit{Supplier}}\mathit{Refund\&Penalty}$, i.e., the seller has to refund the buyer and in addition she has to pay a penalty.

As we discussed in the previous paragraphs the conditions of a contract cannot be taken in isolation in so far as they exist in a contract. Consequently the whole contract determines the meaning of each single clause in it. In agreement with this holistic view of norms we have that the normative content of r' is included in that of r. Accordingly r' does not add any new piece of information to the contract, it is redundant and can be dispensed from the explicit formulation of the contract.

Another common case is exemplified by the rules:

$$\begin{aligned} r : \mathit{Invoice} \vdash\ & O_{\mathit{Purchaser}}\mathit{PayWithin7Days} \otimes \\ & O_{\mathit{Purchaser}}\mathit{PayWithInterest}, \\ r' : \mathit{Invoice}, \neg\mathit{PayWithin7Days} \vdash\ & O_{\mathit{Purchaser}}\mathit{PayWithInterest}. \end{aligned}$$

The first rule says that after the seller sends the invoice the buyer has one week to pay it, otherwise the buyer has to pay the principal plus the interest. Thus we have the primary obligation $O_{\mathit{Purchaser}}\mathit{PayWithin7Days}$, whose violation is repaired by the secondary obligation $O_{\mathit{Purchaser}}\mathit{PayWithInterest}$, while, according to the second rule, given the same set of circumstances *Invoice* and $\neg\mathit{PayWithin7Days}$ we have the primary obligation $O_{\mathit{Purchaser}}\mathit{PayWithInterest}$. However, the primary obligation of r' obtains when we have a violation of the primary obligation of r. Thus the condition of applicability of the second rule includes that of the first rule, and then they have the same normative content. Therefore the first rule is more general than the second and we can discard r' from the contract.

The intuitions we have just exemplified can be fully captured by the following definition.

Definition 1 *Let $r_1 : \Gamma \vdash A \otimes B \otimes C$ and $r_2 : \Delta \vdash D$ be two rules, where $A = \bigotimes_{i=1}^{m} A_i$, $B = \bigotimes_{i=1}^{n} B_i$ and $C = \bigotimes_{i=1}^{p} C_i$. Then r_1* subsumes *r_2 iff*

1. $\Gamma = \Delta$ *and* $D = A$; *or*
2. $\Gamma \cup \{\neg A_1, \ldots, \neg A_m\} = \Delta$ *and* $D = B$; *or*
3. $\Gamma \cup \{\neg B_1, \ldots, \neg B_n\} = \Delta$ *and* $D = A \otimes \bigotimes_{i=0}^{k \leq p} C_i$.

The idea behind this definition is that the normative content of r_2 is fully included in r_1. Thus r_2 does not add anything new to the system and it can be safely discarded.

4.3. Detecting Conflicts

Conflicts arises naturally in contracts. What we have to determine is whether we have genuine conflicts, i.e., the contracts is in some way flawed or whether we have *prima-facie* conflicts. A prima-facie conflict is an apparent conflict that can be resolved when we consider it in the context where it occurs and if we add more information the conflict disappears. For example let us consider the following two rules:

$$\begin{aligned} r : \mathit{PremiumCustomer} &\vdash O_s\mathit{Discount} \\ r' : \mathit{SpecialOrder} &\vdash O_s\neg\mathit{Discount} \end{aligned}$$

Saying that Premium Customers are entitled to a discount (r), but there is no discount for goods bought with a special order (r'). Is a Premium customer entitled to a discount when she places a special order? If we only have the two rules above there is no way to solve the conflict just using the contract and there is the need of a domain expert to advise the knowledge engineer about what to do in such case. The logic can only point out that there is a conflict in

the contract. On the other hand, if we have an additional provision

$$r'' : PremiumCustomer, \neg Discount \vdash O_s Rebate$$

Specifying that if for some reasons a premium customer did not received a discount then the customer is entitled to a rebate on the next order, then it is possible to solve the conflict, because the contract allows a violation of rule r to be amended by r'', using the merging mechanism we analyse in Section 4.1.

The following rule is devised for making explicit conflicting norms (contradictory norms) within the system:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash \neg A}{\Gamma, \Delta \vdash \bot} \tag{1}$$

where

1. there is no rule $\Gamma' \vdash X$ such that either $\neg A \in \Gamma'$ or $X = A \otimes B$; and
2. there is no conditional rules $\Delta' \vdash X$ such that either $A \in \Delta'$ or $X = \neg A \otimes B$; and
3. for any formula B, $\{B, \neg B\} \not\subseteq \Gamma \cup \Delta$.

The meaning of these three conditions is that given two rule, we have a conflict if the normative content of the two rules is opposite, such that none of them can be repaired, and the the states of affairs/preconditions they require are consistent.

Once conflicts have been detected there are several ways to deal with them. The first thing to do is to determine whether we have a *prima-facie* conflict or a genuine conflict. As we have seen we have a conflict when we have two rules with opposite conclusions. Thus a possible way to solve the conflict is to create a superiority relation over the rules and to use it do "defeat" the weaker rule (this is the strategy adopted in [4]). A second alternative is to supplement the antecedent of one rule with an additional guard (this kind of technique has been proposed in a general logical setting in [1] to remove priority over rules, though the precise details could depend on the underly logic). Notice that currently BCL does not support priority over rules/policies, thus the guard approach could be more suitable for BCL.

4.4. FCL Normal Forms and Canonical Forms

We are now ready to outline how the apply the logical machinery we have developed to deal with business contracts before we transform the logical representation in a language apt to monitor the execution of a contract. This consists of the following three steps:

1. Starting from a formal representation of the explicit clause of a contract we generate all the implicit conditions that can be derived from the contract by applying the merging mechanism of FCL.
2. We can clean the resulting representation of the contract by throwing away all redundant rules according to the notion of subsumption.
3. Finally we use the conflict identification rule to label and detect conflicts.

In general the process at step 2 must be done several times in the appropriate order as described before. The normal form of a set of rules in FCL is the fixed-point of the above constructions. A contract contains only finitely many rules and each rule has finitely many elements. In addition it is possible to show that the operation on which the construction is defined is monotonic [7], thus by standard set theory results the fixed-point exists and it is unique. However, we have to be careful since merging first and doing subsumption after produces different results from the opposite order (i.e., subsumption first and merging after), or by interleaving the two operations.

If there is only one normal form of a contract then the normal form coincides with the canonical form of the contract. In case there are multiple normal forms of a contract, for example if the contract has been built in a modular way from several (sub-) contract templates [12], we have to combine the normal forms to check for their completeness and mutual consistency. This means that we have to union the sets of rules from each normal form and to repeat the fixed-point construction of step 2, and then to identify the eventual conflicts. After these operations we obtain the canonical form of the contract. A domain expert can use the canonical form to check that the representation of a contract covers all aspects of the contract, and, in case of conflicts, she suggests which interpretation is the more faithful to the intent of the contract, and she can point out features included in the contract but missing in its formal representation.

5. Mapping BCL to FCL

In this section we will provide a mapping from BCL to FCL that allows us to apply the formal validation and verification procedure to a BCL program. However we will restrict ourselves to the mapping of only the policy fragment of BCL. The proposed mapping can be integrated with the mapping from FCL to BCL presented in [5] to analyse a fragment of a BCL program with formal method and to return a normalised/canonical BCL program for a contract (See Figure 2).

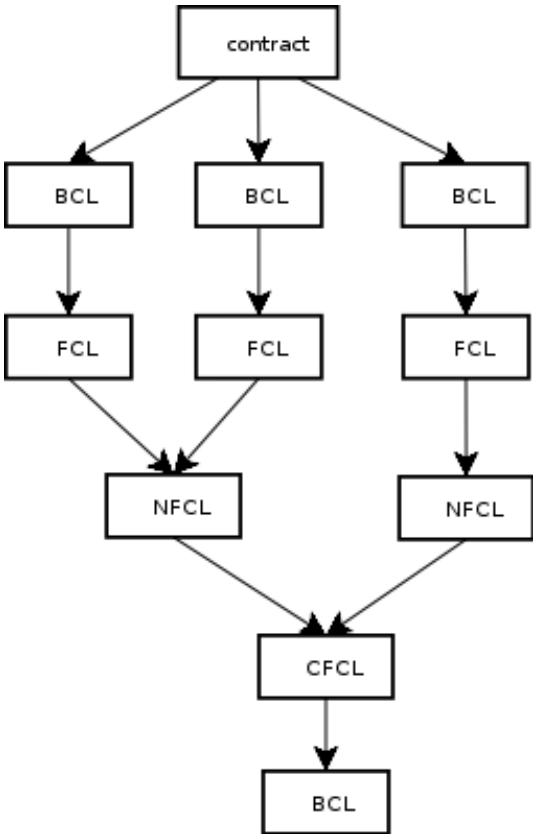

Figure 2. BCL-FCL-BCL Transformation Cycle

We will assume a fixed but arbitrary mapping that extracts the elements of a BCL policy fragment and map them to basic components of FCL (literals, rules labels, and modal operators). Thus for example the auxiliary function $behaviour(p)$ takes a policy p, extracts the behaviour of the policy and returns the FCL literals corresponding to the behaviour of the policy. Similarly for the function *name*, $trigger$, $role$ and $state$.

The mapping *pmap* of a BCL policy[1]

Policy: pId
Role: roleId
Modality: Obligation, Permission, Prohibition
Trigger: eventPatterns
Guard: states, HasOccurred pId' Violation
Behaviour: eventPattern

to a FCL rule is defined as follows:

If the policy does not contain *HasOccurred pId' Violated* then

$$pmap(pId) = name(pId) : trigger(pId), states(pId) \vdash X_{role(pId)} behaviour(pId)$$

where X is O if *Modality: Obligation*, P if *Modality: Permission* and $O\neg$ if *Modality: Prohibition*. Otherwise the mapping is

$$pmap(pId) = name(pId) : trigger(pId), states(pId), \neg behaviour(pId') \vdash X_{role(pId)} behaviour(pId)$$

Let us illustrate the above procedure with two examples

Given the following BCL policy:

Policy: id=7.1
Role: Supplier
Modality: Permission
Trigger: not PayWithin7Days
Guard: 2Delays
Behaviour: Terminate

Since no *HasOccurred pId' Violated* guard occurs in the policy we can use the first part of the mapping to obtain the FCL rule

$$7.1 : 2Delays, \neg PayWithin7Days \vdash P_{Supplier} Terminate$$

On the other hand if we want to map the policy

Policy: id=6.1.0
Role: Purchaser
Modality: Obligation
Trigger: SystemStart
Guard: HasOccurred 6.1 Violated
Behaviour: PayWithInterest

we have to use the second condition of the mapping yielding the following FCL rule.[2]

$$6.1 : \neg PayWithin7Days \vdash O_{Purchaser} PayWithInterest.$$

6. Discussion

In [5] we presented a mapping from FCL to BCL with the aim to provide a way to implement contracts analysed in terms of FCL in a domain specific language. Here we have pursued the opposite direction. We have given a mapping from BCL to FCL. In this way we are able to validate and verify BCL with the formal tools provided by FCL. At the same time we have been able to identify some possible extension for the two formalism. For example FCL has

[1] The following policy is a schema of policy. The modality is one of *Obligation*, *Permission* or *Prohibition*, and in the guard can be either a set of states or a state signalling that policy *pId'* has been violated.

[2] The *SystemStart* event is mapped to a null literal in FCL.

been extended to cope with conflicts via the combination of FCL with Defeasible Logic [4], but this important facility is not present in BCL. On the other hand, so far FCL has no capability to reason about temporal notions but BCL is expressive enough to describe different type of temporal constraints (deadline, sliding windows and so on). To obviate to this problem we intend to supplement the formal model supplied by FCL with the temporal framework developed for temporalised normative positions on a similar formalism in [8].

Acknowledgements

We would like to thank Peter Linington and Antonino Rotolo for their fruitful comments on previous versions of this work. Thanks are also due to the CoALa05 anonymous referees for their valuable criticisms.

The first author was supported by the Australia Research Council under Discovery Project No. DP0558854 on "A Formal Approach to Resource Allocation in Web Service Oriented Composition in Open Marketplaces".

The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training).

References

[1] Grigoris Antoniou, David Billington, Guido Governatori, and Michael J. Maher. Representation results for defeasible logic. *ACM Transactions on Computational Logic*, 2(2):255–287, 2001.

[2] José Carmo and Andrew J.I. Jones. Deontic logic and contrary to duties. In D.M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic. 2nd Edition*, volume 8, pages 265–343. Kluwer, Dordrecht, 2002.

[3] Jonathan Gelati, Guido Governatori, Antonino Rotolo, and Giovanni Sartor. Normative autonomy and normative coordination: Declarative power, representation, and mandate. *Artificial Intelligence and Law*, 12(1-2):53–81, March 2004.

[4] Guido Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3):181–216, June-September 2005.

[5] Guido Governatori and Zoran Milosevic. Dealing with contract violations: formalism and domain specific language. In *9th International Enterprise Distributed Object Computing Conference (EDOC 2005)*. IEEE Computer Society, 2005.

[6] Guido Governatori and Antonino Rotolo. A Gentzen system for reasoning with contrary-to-duty obligations. A preliminary study. In Andrew J.I. Jones and John Horty, editors, *Δeon'02*, pages 97–116, London, May 2002. Imperial College.

[7] Guido Governatori and Antonino Rotolo. Logic of violations: A Gentzen system for reasoning with contrary-to-duty obligations. *Australasian Journal of Logic*, 2005.

[8] Guido Governatori, Antonino Rotolo, and Giovanni Sartor. Temporalised normative positions in defeasible logic. In *10th International Conference on Artificial Intelligence and Law (ICAIL05)*, pages 25–34. ACM Press, 2005.

[9] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5/WG3-N695, ANSI, 11 West 42nd Street, New York, NY 10036, 1982.

[10] Benjamin N. Grosof and Terrence C. Poon. SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *12th International Conference on World Wide Web*, pages 340–349. ACM Press, 2003.

[11] Henning Herrestad and Christen Krogh. Obligations directed from bearers to counterparts. In *5th International Conference on Artificial Intelligence and Law (ICAIL'95*, pages 210–218. ACM Press, 1995.

[12] Yigal Hoffner and Simon Field. Transforming agreements into contracts. *International Journal of Cooperative Information Systems*, 14(2-3):217–244, 2005.

[13] Ronald M. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4:27–44, 1988.

[14] Peter F. Linington, Zoran Milosevic, James B. Cole, Simon Gibson, Sachin Kulkarni, and Stephen Neal. A unified behavioural model and a contract language for extended enterprise. *Data & Knowledge Engineering*, 51(1):5–29, 2004.

[15] Peter F. Linington, Zoran Milosevic, and Kerry Raymond. Policies in communities: Extending the odp enterprise viewpoint. In *2nd International Enterprise Distributed Object Computing Workshop (EDOC98)*, La Jolla, November 1998.

[16] Zoran Milosevic and R. Geoff Dromey. On expressing and monitoring behaviour in contracts. In *6th International Enterprise Distributed Object Computing Conference (EDOC 2002)*, pages 3–14. IEEE Computer Society, 2002.

[17] Zoran Milosevic, Simon Gibson, Peter F. Linington, James B. Cole, and Sachin Kulkarni. On design and implementation of a contract monitoring facility. In *1st IEEE Workshop on Econtracting (WEC04)*, pages 62–70. IEEE Computer Society, July 2004.