A FORMAL ANALYSIS OF A BUSINESS CONTRACT LANGUAGE

GUIDO GOVERNATORI

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

ZORAN MILOSEVIC

Deontik, Australia. email: zoran@deontik.com

This paper presents a formal system for reasoning about violations of obligations in con-
tracts. The system is based on the formalism for the representation of contrary-to-duty
obligations. These are the obligations that take place when other obligations are violated
as typically applied to penalties in contracts. The paper shows how this formalism can be
mapped onto the key policy concepts of a contract specification language, called Business
Contract Language (BCL), previously developed to express contract conditions for run
time contract monitoring. The aim of this mapping is to establish a formal underpinning
for this key subset of BCL.

Keywords: Formal Contract Language, Business Contract Language

1. Introduction

The wide penetration of broadband networks and new computing technologies
such as XML, Web Services, Service Oriented Architectures and Even-Driven-
Architectures enable better and more versatile collaborative models between enter-
prises. Examples are virtual organisations, supply chains and extended enterprise.
These cross-enterprise models require better links between partners' business activ-
ities including more transparency of their data and processes than in the past. Such
models also require faster reaction to business events of relevance to organisations'
interactions. The business events can be either related to the occurrences associated
with the existing operational interactions, or can be triggered by the need to add or
modify the existing architecture reflecting an evolutionary character of occurrences.

These new collaboration models however give prominence to a number of prob-
lems some of which are new and some which may have been partially addressed in
the past. One such problem is the positioning of business contracts as a governance
mechanism for cross-organisational collaboration, rather than treating them as legal
documents with a weak link with the cross-organisational interactions they govern.
As a result, there is a renewed interest in contract architectures and languages as

2   *G. Governatori and Z. Milosevic*

the foundation for facilitating the automation of contract management activities.

This paper presents a formal system for describing contracts in terms of deontic concepts such as obligations, permissions and prohibitions. Furthermore, the logic supports reasoning about violations of obligations in contracts. The system is based on the formalism for the representation of contrary-to-duty obligations[10]. These are the obligations that take place when other obligations are violated as typically applied to penalties in contracts. We then use this formalism as a source of the mapping to the key policy concepts of a contract specification language, called Business Contract Language (BCL), previously developed to express contract conditions of relevance for run time contract monitoring[17,21]. BCL is a domain specific language, designed to support abstractions needed for the expressions of business contracts. It was developed by taking into account several policy and community frameworks[18,15,17] and an expressive event language for the specification of event-based behaviour as part of policy expressions. The initial research prototype for the BCL as part of a Business Contract Architecture was developed and tested using several contract examples[17,21,20]. Although BCL has its basis in these well-founded concepts, it was also developed in an incremental manner, as we were considering increasingly complex contract scenarios and case studies from the contract management domain.[a] However, this style of BCL development has led to the need for a more formal treatment of the language and this paper is a step towards this direction.

In the next section we introduce an example of a business contract, which will be used to illustrate the concepts discussed throughout the paper. In Section 3 we consider contracts as legal instruments and express their semantics using a logic-based formalism for reasoning about the contrary to duty obligations[10]. The main idea behind this formalism is to express contract semantics in terms of deontic modalities (or normative constructs) such as obligations, permissions and prohibitions. In addition, this formalism supports the expression of and reasoning about violations of such deontic modalities and the subsequent actions that need to be taken to deal with violations. This system allows for checking of contract consistency and determining whether there are missing or implied statements. We will refer to this formalism as formal contract language (FCL). FCL is then used to check the expressive power of relevant parts of BCL to be briefly described in section 5. BCL fragments for an example contract are presented in section 6. In sections 7 and 8 we establish a correspondence between the semantics of FCL and the core concepts of BCL. Section 9 provides an overview of related work. The paper concludes with listing the main points and by outlining our future research directions in this area.

2. A Sample Contract

Consider the following sample contract, based on[20] and revised in[7].

---

[a]Note that the language is XML-centric, exploiting relevant XML standards, in particular Xpath, but these are not discussed in this paper.

CONTRACT OF SERVICES

This Deed of Agreement is entered into as of the Effective Data identified below.

BETWEEN ABC Company (To be known as the Purchaser)

AND ISP Plus (To be known as the Supplier)
WHEREAS (Purchaser) desires to enter into an agreement to purchase from (Supplier) Application Server (To be known as (Service) in this Agreement).

NOW IT IS HEREBY AGREED that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:

1 Definitions and Interpretations

1.1 Price is a reference to the currency of Australia unless otherwise stated.

1.2 This agreement is governed by Australia Law and the parties hereby agree to submit to the jurisdiction of the Courts of the Queensland with respect to this agreement.

2 Commencement and Completion

2.1 The commencement date is scheduled as January 30, 2004.

2.2 The completion date is scheduled as January 30, 2005.

3 Price Policy

3.1 A "Premium Customer" is a customer who has spent more that $10000 in services. Premium Customers are entitled a 5% discount on new orders.

3.2 Services marked as "special order" are subject to a 5% surcharge. Premium customers are exempt from special order surcharge.

3.3 The 5% discount for premium customers does not apply for services in promotion.

4 Purchase Orders

4.1 The (Purchaser) shall follow the (Supplier) price lists at http://supplier/cat1.html

4.2 The (Purchaser) shall present (Supplier) with a purchase order for the provision of (Services) within 7 days of the commencement date.

5 Service Delivery

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), otherwise the (Supplier) shall refund the (Purchaser) and pay the (Purchaser) a penalty of $1000.

5.2 The (Supplier) shall on receipt of a purchase order for (Services) make them available within 1 day.

5.3 If for any reason the conditions stated in 5.1 or 5.2 are not met, the (Purchaser) is entitled to charge the (Supplier) the rate of $ 100 for each hour the (Services) are not delivered.

6 Payment

6.1 The payment terms shall be in full upon receipt of invoice. Interest shall be charged at 5 % on accounts not paid within 7 days of the invoice date. The prices shall be as stated in the sales order unless otherwise agreed in writing by the (Supplier).

6.2 Payments are to be sent electronically, and are to be performed under standards and guidelines outlined in PayPal.

7 Termination

7.1 The (Supplier) can terminate the contract after three delayed payments.

In a nutshell, the items covered within this contract are: (a) the roles of the parties; (b) the period of the contract (the times at which the contract is in force); (c) the nature of consideration (what is given or received), e.g., actions or items; (d) the obligations and permissions associated with each role, expressed in terms of criteria over the considerations, e.g., quality, quantity, cost and time; (e) some dependencies between policies, and (f) the domain of the contract (which determines the rules under which the validity, correctness, and enforcement of the contract will operate).

3. Formal Representation of Contracts

Business contracts are mutual agreements between two or more parties engaging in various types of economic exchanges and transactions. They specify obligations, permissions and prohibitions for parties involved in contact and state the actions or penalties that may be taken when any of the stated agreements are not being met.

Narative contracts often contain ambiguities (e.g., conflicts and gaps) and these must be avoided or at least the conflicts arising from them resolved. Furthermore, there may be complex interdependencies between contract clauses that can be hard to track down. In order to address these inherent complexities in many business contracts, there is a need for different tools such as contract authoring tools, to support verification of contracts, or tools for contract monitoring to check how parties fulfil their policies. These tools in turn require a formal representation of contracts. A formal foundation is thus a prerequisite for verification or validation purposes. Consequently, one of the benefits 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.

A formal language for contracts should be conceptual, allowing its users to focus exclusively on aspects related to the content of the contract, ignoring implementation aspects such as external or internal data representation, physical data organisation and access, or platform related aspects such as message-passing formats.

A formal language intended to represent contracts should provide notions closely related to the concepts of obligations, permissions, entitlements and other mutual normative positions that the signatories of the contract subscribe to. Since the seminal work by Lee[16] Deontic Logic has been regarded as one on the most prominent paradigms for the formaalisation of contracts.

3.1. *Obligations, Violations and Contrary-to-Duties*

Deontic Logic extends classical logic with the modal operators O, P and F. The interpretations of the formulas OA, PA and FA are, respectively, that A is obliga-

tory, A is permitted and A is forbidden. The modal operators obey the usual mutual relationships

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

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 in contracts apply to the signatories of the contract. To capture this we will consider directed deontic operators[12]; 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.

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 (CTDs) or reparational obligations (because they are activated when normative violations occur and are meant to 'repair' violations of primary obligations[3]). Thus a CTD is a conditional obligation arising in response to a violation, where a violation is signalled by an unfulfilled obligation. The ability to 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.

CTDs are one of the most debated field of deontic logic and, at the same time, they are subject to several logic paradoxes. In this paper we just focus on a simple logic of violation that seems to avoid most of these paradoxes and offers a simple computational model to compute chains of violations. The ability do deal with violations or potential violations and the reparational obligation generated from them is one the essential requirements for reasoning about and monitoring of business contracts.

The idea behind the logic of violation[10] 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 other 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 the implementation of the monitoring. Accordingly, a logic of violation 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 causal 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 suggests that violations are special kinds of exceptions[10], and several authors have used exceptions to raise conditions to repair a violation in the context of contract monitoring[21,11].

In the next section we lay down the foundations for FCL.

3.2. *Reasoning about Violations*

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.

The formation rules for $\otimes$-expressions allow a permission to occur only at the end of such expressions. This is due to the fact that a permission can be used to repair a violation, but it is not possible to violate a permission, thus it makes no sense to have reparations of permissions.

Each condition or policy of a contract is represented by a rule in FCL

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

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 does not conform with the published standards") can be represented as

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

where the propositional letter p means "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

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

and 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-hand 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 $\neg A$.

Sometimes contracts contain other mutual normative positions such as delegations, empowerment, rights and so. Often these positions can be effectively represented in terms of complex combinations of directed obligations and permissions[6]. Hence violations of such complex positions result in violations of the consituent obligations.

4. Normalisation tools for FCL

In this section we examine how FCL can be used to analyse contracts and to reason about them so that ambiguities in a contract can be identified.

We introduce transformations of an FCL representation of a contract to produce a normal form of the same. A normal form is a representation of a contract

based on an FCL specification containing all contract conditions that can be generated/derived from the given FCL specification. The purpose of a normal form is to "clean up" the FCL representation of a contract, that is to identify formal loopholes, deadlocks and inconsistencies in it, and to make hidden conditions explicit.

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 violations 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 4.3 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 through the violation conditions. In what follows we will first examine some common patterns of this kind of construction and then we will show how to generalise them.

Let us 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 the violation of $O_s A$ is part of the premises of another policy, for example,

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

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 given two policies such that one is a conditional obligation ($\Gamma \vdash O_s A$) and the antecedent of second contains the negation of the propositional content of a the first ($\Delta, \neg A \vdash O_{s'} C$), then the latter is a reparational obligation of the former. Their reciprocal interplay makes them two related norms so that they cannot be viewed anymore as independent obligations. Therefore we can combine them to obtain an expression (i.e., $\Gamma, \Delta \vdash O_s A \otimes O_{s'} C$) that exhibit the *explicit reparational obligation* of the second norm with respect to the first. 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 the rules

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

From these we obtain

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

We 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 we can consider the situation described by Clause 5.1 of the contract. Given the rules

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

from which we derive the new rule

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

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

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

where $\mathbf{X}$ denotes either an obligation or a permission. In this last case, we will impose that D is a literal. Since the minor premise states that $\mathbf{X}_s D$ is a reparation of $O_s B_n$, 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. 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 examples and then we show how to give a formal definition of the notion of subsumption appropriate for FCL.

Let us consider the rules

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

10   *G. Governatori and Z. Milosevic*

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, if she violates such an obligation, then the violation of *QualityOfService* can be repaired by replacing the faulty goods within three days ($O_{Supplier}Replace3days$). In other words $O_{Supplier}Replace3days$ is a secondary obligation arising from the violation of the primary obligation $O_{Supplier}QualityOfService$. In addition r prescribes that the violation of the secondary obligation $O_{Supplier}Replace3days$ can be repaired by $O_{Supplier}Refund\&Penalty$, 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 as part of 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:

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

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 $O_{Purchaser}PayWithin7Days$, whose violation is repaired by the secondary obligation $O_{Purchaser}PayWithInterest$, while, according to the second rule, given the same set of circumstances *Invoice* and $\neg PayWithin7Days$ we have the primary obligation $O_{Purchaser}PayWithInterest$. 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, which then is more general than the second and we can discard r' from the contract.

The intuitions we have just exemplified is captured by the following definition.

Definition 4.1.   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 intuitions 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 often arises 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 is resolved. For example let us consider the following two rules:

$$r : PremiumCustomer \vdash O_s Discount$$
$$r' : SpecialOrder \vdash O_s \neg Discount$$

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:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash \neg A}{\Gamma, \Delta \vdash \bot} \tag{2}$$

where

(1) there is no rule $\Gamma' \vdash X$ such that either $\neg A \in \Gamma'$ or $X = A \otimes B$;
(2) there is no conditional rules $\Delta' \vdash X$ such that either $A \in \Delta'$ or $X = \neg A \otimes B$;
(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 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[7]. 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[1] to remove priority over rules, though the precise details could depend on the underlying 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*

We now apply the logical machinery presented to validate and transform business contracts into 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 clauses 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 above. 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[10], thus according to standard set theory results the fixed-point exists and it is unique. However, we have to be careful since merging first and doing subsumption afterwards produces different results from the opposite order (i.e., subsumption first and merging after), or by interleaving the two operations.

A domain expert can use the normal 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. In addition the domain expert can point out features included in the contract but missing in its formal representation.

5. Business Contract Language in brief

The purpose of BCL is to specify business contracts in a way suitable to enable monitoring of contract execution in an event-based manner. A contract execution period starts after contract terms are agreed and the contract is signed by signatories to the contract, and finishes at the specified point in time stated in the contract or as a result of various other termination conditions such as contract violation. Real-time monitoring of activities of the roles involved in business processes governed by contracts is a key aspect of enterprise contract management. The aim is to check whether these activities signify fulfilment policies agreed in the contract or their existing or possibly forthcoming violations. A special case of policy violation refers to situations in which a required activity of a role stated in a contract (directly or indirectly) has not been carried out. This means that the monitoring also needs to detect cases of the non-execution of activities emanating from contracts.

BCL incorporates relevant concepts from the Reference Model for Open Distributed Processing standards[14] and ODP Enterprise Language standard[15] and is developed by considering a number of scenarios from business contract manage-

ment domain. BCL also introduces the concept of event pattern as a specific style of expressing states of affairs of relevance for contract monitoring.

Event is the central concept in BCL and BCL can be regarded as an event-driven language. A single event can be used to signify:

- an action of a signatory to the contract, or any other party mentioned in the contract
- a temporal occurrence such as a deadline,
- change of state associated with a contract variable,
- contract violation and other conditions associated with contract execution.

In addition, multiple events can be combined and used to describe the execution of more complex activities. We introduced the concept of *event pattern* to specify relationships between events that are of relevance to business contracts. An event pattern is a means for describing a state of affairs. A state of affairs can range from the elementary, such as the occurrence of a particular action performed by a party or the passing of a deadline, to the more complex, such as "more than three sets of down time in any a one week period" and "one of the contract conditions has been violated". Examples of event relationships are logical relationships between events (AND, OR, NOT), temporal relationships (e.g., before and after), temporal constraints on event patterns (e.g., absolute and relative deadlines and sliding time windows[17]), event causality, and some special kinds of singleton event pattern (e.g., contract violation and state change events). We note that the event pattern concept has many similarities to the work by Luckham on complex event processing[19].

The main use of event patterns in BCL is to enable checking of policies related to a contract. Policies define behavioural constraints for the roles that carry out activities in a contract and these constraints are described in terms of event patterns. Policy checking consists of identifying event patterns in activities of parties filling a role and establishing whether they satisfy the policies. The policies take a form of modal constraints such as obligations, permissions and prohibitions. These modal constraints in a contract specification reflect their English-language meaning: obligations identify activities that must occur, permissions identify activities that may occur, and prohibitions identify activities that must not occur. In all cases, these constraints can be conditional, for example, if payment is not made then the supplier is permitted to charge interest on the outstanding amount. We note that there may be other business rules that state various constraints on a contract and do not have explicitly modal character, such as the start and end date of a contract and these are easier to incorporate as part of contract monitoring. Policies represent a key constituent of a business contract specifications. A contract is described as a set of policies that apply to the behaviour of signatories and various other parties filling roles involved in business processes governed by contracts.

As a result of policy checking procedures a policy violation may be detected. In BCL, we represent the occurrence of such violation using a special kind of event type, namely *PolicyViolation* event. If it occurs, this event can then be treated like

any other event and can be used as part of other event patterns for various purposes. One specific purpose is to use this event to link the violating policy with another policy that can take effect in response to this violation, referred to as CTD or reparation policy earlier. This is the mechanism used in BCL for the expression of (possibly a chain of) reparation policies. In this case, a BCL guard can be used as a precondition for the activation of this reparation policy. After this policy is activated the same monitoring machinery can be applied to check the fulfilment of this new policy. There is no limit of how many policies can be chained using this approach.

Contract related events can often change the state of various variables associated with the contract. To this end, BCL defines the concept of a *state* for that contract variable and the value of this state can be either determined explicitly in response to an event, or on request, when the state value is needed. Typically, a contract has many state variables changing in response to the corresponding events.

BCL introduces the concept of a *community*, an overarching concept for the specification of objects that collaborate to achieve a certain goal. These objects fill the roles of a community. Thus, a community is defined as a set of roles, policies that apply to the relevant roles, states, and related event patterns that apply to the community. We note that community is a general concept for describing collaboration and can be used to model structure within one organisation or cross-organisational structures. Business contracts are a specific kind of community.

6. BCL Fragments

We illustrate the use of BCL through the example of a contract for services from section 2 with fragments of language expressions introduced progressively and discussed alongside these fragments. We provide a substantial simplification of BCL syntax to illustrate the BCL concepts and their usage and use only those attribute of BCL concepts which are of relevance for the example. Note that BCL is XML-centric, exploiting relevant XML standards, in particular XPath, but these details are omitted from this paper. The BCL fragments are contract-oriented constraints over the purchasing process.

Role A BCL *Role* is used as a label for a party whose behaviour is constrained by policies stated in a contract. BCL roles are names, with the expected behaviour of parties filling roles defined in the containing *Policy* specification. Policy specification in turn includes *EventPattern* definitions associated with a specific *Role* name. The syntax for role identification is as follows:

    Role: Purchaser

Note that BCL roles have cardinality, that is, more than one party in a contract can fulfil a role.

Event Pattern As discussed previously in section 5, event patterns are a key component in checking policies related to a contract. Policy checking consists of identifying event patterns in activities of parties filling a role and ensuring that they satisfy the policies. Events are matched with an event pattern by event type.

> *Event typeId=PurchaseOrder*
>   *Defined by XMLSchema*
>   *for UBL Order*

This specifies that a purchase order event is signified by the existence of an XML document using the UBL Order XML schema. Note that event matching can exploit some further information such as event parameters from the event content, e.g. the amount specified in the Purchase Order document included as an event's payload.

Events in BCL can involve multiple *EventRoles*. The *EventRole* concept is a generic labelling mechanism for identifying roles in event execution that can be played by participants. An event with multiple roles is specified as follows:

> *Event typeId=PurchaseOrder*
>   *EventRole name=Buyer*
>   *EventRole name=Seller*

The BCL event roles are different from contract roles: by using generic event role names, the same event definition can be re-used in many contexts. The BCL event roles can then be bound to specific contract roles as shown by the mapping for our example as follows (to illustrate the point we only include the name attributes of the RoleType and EventRole elements):

> *Event typeId=PurchaseOrder*
>   *EventRole name=GenericBuyer*
>   *RoleType name=Purchaser*
>   *EventRole name=GenericSeller*
>   *RoleType name=Supplier*

This applies to all purchase orders in the community template, meaning that the event pattern will only be matched if the *Purchaser* fills the *GenericBuyer* event role and the *Supplier* fills the *GenericSeller* role.

State The BCL *State* construct is used to define data values shared by the participants in the *Community*. This is used to maintain running totals, counters and other state required to evaluate policy. Such a state defines a set of update actions and is introduced with the following syntax:

> *State: GoodsPurchasedAmount*
>   *CalculationExpression*
>   *UpdateOn: Payment*
>   *UpdateSpecification: return this + Payment.Amount*

This defines the amount spent by the *Purchaser*, which is updated whenever a payment is made. State changes are bound to event patterns and are deterministic.

That is, the value of a state can only be modified through the matching of visible event patterns. While such state is relatively easy to maintain consistently in an environment with centralised control, maintaining state in a distributed context is considerably more difficult[2].

Policy   A *Policy* is used to specify business-level constraints in a BCL *Community*[17]. It is explicitly associated with a *Role* and has a *Modality* indicating whether it is an obligation, permission or prohibition, described in detail below.

The behaviour associated with a policy is a conditional expression over events expressed as an event pattern. 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 another 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 behavioural condition expression, i.e., from the event pattern. We refer to that part of a behaviour expression as a *trigger*. So, in our example, the trigger 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*.

Obligation   A BCL *Policy* of an *Obligation* modality indicates that the event pattern defined in the policy must occur. An obligation is specified as follows:

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

The policy specifies the goods availability behaviour condition (clause 5.2 in the example contract) as an event pattern. In this case the event pattern is satisfied if the *GoodsAvailable* event generated by the Supplier (or their agent) 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. Notice that this policy specifies obligation on the supplier and it is silent about policies that may apply to other roles. For example, the policy does not say anything about the origins of the *PurchaseOrder* event and policies that might apply to the party that generates this event. Thus, the triggering condition for this policy is occurrence of *PurchaseOrder* event.

Note that a *GoodsAvailable* event signifies availability of goods which may be manually or automatically entered into the system and a *PurchaseOrder* event sig-

nifies, for example, that a message carrying a *PurchaseOrder* document has arrived. These events can be generated using any delivery mechanism such as email, SMS message etc.

The definition of monitoring for this obligation is made easier by the explicit specification of a time period in the policy. If the obligation is not satisfied in the time period, then a violation event will be generated.

Permission A BCL *Policy* of a *Permission* modality indicates that the behaviour defined in the policy is allowed to occur. For example:

```
Policy: ChargingPolicy
  Role: Supplier
  Modality: Permission
  Trigger: SystemStart
  Behaviour: InvoiceSend after GoodsAvailable
```

The policy specifies that the Supplier is permitted to send an invoice after it made goods available. Note that the example contract does not explicitly state this policy, but we imply it from the natural language interpretation of the contract.

Prohibition A BCL *Policy* of a *Prohibition* modality indicates that the behaviour defined in the policy must not occur, for example:

```
Policy: PurchaserSpecialOrderCondition
  Role: Purchaser
  Modality: Prohibition
  Trigger: PurchaseOrder
  Behaviour: PurchaseOrder.PurchaserAge less LegalAge
```

This policy specifies that Purchasers below legal age are prohibited from purchasing "special orders".

Violations BCL supports expression of guarded conditions that can be applied 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 the precondition for the evaluation of the corresponding element. For example, the 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.

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

One specific use of guards can be to specify conditions for the activation of reparations of CTD policies as discussed in previous section. For example, the following policy expresses the condition in the first sentence of Clause 5.1 of the example contract. Note that for simplicity we do not elaborate on the exact meaning of the *QualityOfServiceAgreement* condition below.

> *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*) will be generated indicating that this obligation is violated. This event 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: Replace.date before 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 the *HasOcurred* event pattern expression where the *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 CTD obligation, whether the Replace event ocurred within 3 days of obligation violation.

7. Mapping FCL to BCL

In this section we present a mapping from the FCL to BCL, with the aim of determining whether BCL supports key concepts of FCL. First we will extend the language of FCL with a set of rule labels. Those labels will be used to uniquely identify the clauses of a contract.

The mapping of a formal contract $\mathcal{C}$ from FCL to BCL is determined by a function *map* that parses each rule r_i in $\mathcal{C}$ and return an expression in BCL, according to the format of the elements in r_i. Given a rule

$$r_i : A_1^i, \ldots A_n^i \vdash B^i$$

where, r_i is the id of the rule, A_j^is are either modal literals or literals and B^i is an $\otimes$-expression, we use $Ant(r)$ to denote the set of literals in the antecedent of the

rule and $Con(r)$ to denote the consequent of the rule. Thus given rule r_i, we have

$$Ant(r_i) = \{A_1^i, \ldots, A_n^i\}, \quad Con(r_i) = B^i$$

A modal literal carries three types of information: the modality (obligation, permission, prohibition), the subject or bearer of the modality, and the expected behaviour. For example the modal literal $O_{buyer}PayWithin7days$ indicates that the buyer, has the obligation to pay for a service within seven days. Here the modality is O (an obligation), the buyer is the subject of the obligation, and $PayWithin7days$ is the expected behaviour of the subject of the obligation. To map a modal literal into BCL, we have to define functions to extract these pieces of information from FCL. We assume fixed but arbitrary bijections from the set of events symbols and propositional letters in FCL to event patterns and states in BCL, and from subjects of modalities in FCL to roles in the community of BCL corresponding to the contract $\mathcal{C}$. Thus we have that given a modal literal M_sA, $role(M_sA)$ returns the role in BCL corresponding to the subject s, $behaviour(m_sA)$ returns the event or state corresponding to the literal A, and $modality(M_sA)$ returns *Obligation* if $M = O$, *Permission* if $M = P$ and *Prohibition* if $M = F$.

The antecedent of a rule is a set of literals and modal literals, and in FCL a literal can be either an event symbol or a propositional letter. In FCL both propositional letters and event symbols have the same logical status. However, this distinction is important for contract monitoring (see the discussion in the section where we present the elements of BCL). Therefore when we map them from FCL to BCL we must be able to distinguish these and to use them in the appropriate ways. To this end we introduce two functions, *parseEvents* and *parseStates* that take as input a set S of literals and return the set of events corresponding to the event symbols in S and the states corresponding to the propositional letters in S. If S does not contain any event symbols *parseEvents* returns the special event *SystemStart*.

The mapping from FCL to BCL is done by a function *map* that takes as input a rule in FCL and it returns a policy in BCL. When a rule specifies a CTD (i.e., the consequent of the rule is an $\otimes$-expression) then *map*, besides returning the policy corresponding to the primary obligation, will additionally call an auxiliary function *vmap* (for violation map).

The function $map(r_i)$ is thus defined as:

If $B^i = M^sC^i$ for some modality M^s and (modal) literal C^i (i.e., B^i is a modal literal) then $map(r_i)$ generates the following policy:

> Policy: id=r_i
>> Role: $role(Con(r_i))$
>> Modality: $modality(Con(r_i))$
>>> Trigger: $parseEvents(Ant(r_i))$
>>> Guard: $parseStates(Ant(r_i))$
>>> Behaviour: $behaviour(Con(r_i))$

otherwise, when there is a reparation of an obligation involved, namely when

$B^i = O^r C^i \otimes D^i$ (i.e., B^i is an $\otimes$-expression), $map(r_i)$ generates the following BCL policies:

> Policy: id=r_i
> > Role: $role(Con(r_i))$
> > Modality: Obligation
> > > Trigger: $parseEvents(Ant(r_i))$
> > > Guard: $parseStates(Ant(r_i))$
> > > Behaviour: $behaviour(Con(r_i))$
> $vmap(D^i, r_i, 0)$

The function *vmap* generates a policy corresponding to the violation of the policy where the call to the function occurs.

The function $vmap(B^i, r_i, n)$ takes as input a deontic formula B^i, a rule r_i, and an integer n. n essentially gives the level at which the violation occurs, and it will be used to determine the id of the policy corresponding to the reparation of the violation. As for the function *map*, the definition of *vmap* depends on the format of its first parameter. If $B^i = M^r C^i$ then, the BCL policy corresponding to $vmap(B^i, r_i, n)$ is

> Policy: id=$r_i.n$
> > Role: $role(Con(r_i))$
> > Modality: $modality(Con(r_i))$
> > > Trigger: SystemStart
> > > Guard: HasOccured r_iViolated
> > > Behaviour: $behaviour(Con(r_i))$

otherwise (i.e., $B^i = O^r C^i \otimes D^i$) *vmap* produces the following BCL expression

> Policy: id=$r_i.n$
> > Role: $role(Con(r_i))$
> > Modality: Obligation
> > > Trigger: SystemStart
> > > Guard: HasOccured r_iViolated
> > > Behaviour: $behaviour(Con(r_i))$
> $vmap(D^i, r_i, n+1)$

We illustrate the mapping with the help of some examples. Let us consider the rule corresponding to Clause 7.1 of the contract ("The supplier can terminate the contract after 3 delayed payments").

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

Where *2Delays* is a propositional letter and *PayWithin7days* is an event symbol.

The element of the rule are:

$$Ant(7.1) = \{\mathit{2Delays}, \neg PayWithin7Days\}$$
$$Con(7.1) = P_{Supplier} Terminate$$

Here $Con(7.1)$ is a modal literal, therefore we can use the first part of the the definition of *map*. Moreover

$$role(P_{Supplier}\,Terminate) = \text{Supplier}$$
$$modality(P_{Supplier}\,Terminate) = \text{Permission}$$
$$behaviour(P_{Supplier}\,Terminate) = \text{Terminate}.$$

For the antecedent of the rule we have

$$parseEvents(Ant(7.1)) = \neg PayWithin7Days$$
$$parseStates(Ant(7.1)) = 2Delays.$$

Therefore the mapping of rule 7.1 gives us the following policy in BCL

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

In the second example we have a case where we have to use *vmap*. Consider the rule corresponding to the first part of Clause 6.1 of the contract.

$$6.1 : Invoice \vdash O_{Purchaser}\,PayWithin7Days \otimes O_{Purchaser}\,PayWithInterest.$$

The elements of the rule are

$$Ant(6.1) = \{Invoice\}$$
$$Con(6.1) = O_{Purchaser}\,PayWithin7Days \otimes O_{Purchaser}\,PayWithInterest$$

Since $Con(6.1)$ is an $\otimes$-expression we have to use the second part of the definition of *map*, from which we obtain

Policy: id=6.1
    Role: Purchaser
    Modality: Obligation
        Trigger: Invoice
        Behaviour: PayWithin7Days
$vmap(O_{Purchaser}\,PayWithInterest, 6.1, 0)$

At this stage we have to evaluate

$$vmap(O_{Purchaser}\,PayWithInterest, 6.1, 0).$$

Since the first argument of the *vmap* is a modal literal we can use the first part of the definition. This yields the following BCL policy

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

The analysis above demonstrates that all the current FCL concepts can be mapped to corresponding parts of BCL, in particular the policy aspects of BCL. The FCL connective operator $\otimes$ introduced in Section 3.1 provides a formal foundation for expressing primary obligations and violation conditions, and their ensuing policies in a recursive way.

In terms of BCL, we have shown that a combination of BCL guards and a special kind of event, namely *PolicyViolation* event, can be used to implement the semantics of the FCL connective operator. The occurrence of *PolicyViolation* can be used to set to true the guard condition that applies to the reparation policy. Similarly to FCL, it is possible to specify a chain of reparation policies. This capability further illustrate the expressive power of BCL.

8. 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 given 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 the previous section to analyse a fragment of a BCL program using formal methods and to return a normalised BCL program for a contract.

We will assume a mapping that extracts the elements of a BCL policy fragment and maps 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 functions *name*, *trigger*, *role* and *state*.

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

Policy: *pId*
  Role: *roleId*
  Modality: **Obligation|Permission|Prohibition**
  Trigger: *eventPatterns*
  Guard: [*state*]|[**HasOccured** *pId′* **Violated**]
  Behaviour: *eventPattern*

to a FCL rule is defined as follows:

---

[b]The following policy is a schema of policy. The guard can be either a set of states *state* or a state signalling that policy *pId′* has been violated, **HasOccured** *pId′* **Violated**.

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

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

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 FCL rule[c]

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

The above two transformations allow us to map a BCL version of the contract to a corresponding FCL version and then to obtain the normal form of the contract. After this step we can use the FCL to BCL mapping to transform back the FCL normal form into a consolidated BCL version ready to be used for contract monitoring.

9. Related Work

Other contract languages have been proposed recently, most notably the Contract Expression Language[4], Web Services Level Agreements[13] and ecXML[5]. BCL has a number of similarities with these. For example, regarding the event-oriented style

---

[c]The *SystemStart* event is mapped to an empty literal in FCL.

of the specification, it has similarities with ecXML and regarding its deontic foundation, it has similarities with ecXML, CEL and WSLA. However, BCL covers broader aspects, including the organisational context for the definition of policies, behaviour and structure and relationships between these concepts. In terms of the logical approach of the FCL presented in this paper, this work has similarity with the early work of Lee[16], who proposed the use of deontic formalism for the specification of contracts. However, to the best of our knowledge the work presented in this paper is unique in that we apply a recently developed logic of violation[10] to specify aspects of contracts that deal with violations. Grosof[11] considers the monitoring of contracts and includes the treatment of violations, but it does not use deontic modalities. Thus there is not a full correspondence between the proposed logic and the domain to be modelled by it, thus the treatment of violations must be hard-coded in the definitions of the rules and policies instead of in the logic to reason about them.

10. Discussion and Future Work

In this paper we presented a formal system for the representation of contracts including the representation and reasoning about violations of obligations in contracts. We use this system to provide a logic-based foundation for the policy aspects of the domain specific language, BCL, which was developed to support business contract specification for contract monitoring purposes. The paper also shows how to transform BCL into FCL to provide formal validation and verification of BCL specifications.

Our investigation of BCL has found high level of its expressiveness for this purpose. The BCL expressions of obligations, permissions and prohibitions are sufficient to express most of the deontic concepts addressed by the FCL. In addition, BCL provides a good solution for the expression of violations and the corresponding reparations or CTD obligations. This solution is based on the use of the concept of a guard as a predicate for determining when the dependent, e.g., reparation policies should be activated. This predicate in turn is expressed as a special kind of event pattern expression that allows detection of a specific kind of event type, the policy violation event type. Note that in our implementation of a contract management system this event type is generated by a business policy monitoring component at a point in time when the policy's enclosing event pattern has been found to be violated.

One can perhaps attribute this expressiveness of BCL to the precise enterprise modelling framework used as a starting for formulating BCL concepts. This was then augmented with the incremental development of the language, based on increasingly complex contract scenarios collected from various interaction patterns such as e-procurement and industry domains such as finance and insurance. This in turn suggested a need for a well structured language to reflect the separation of main concerns such as separate structuring into community, policy, state and

event pattern sub-models. The language is augmented with the use of events as central point for integrating these sub-models. This design solution enables further evolution of the language as more scenarios are gathered.

We have also identified several aspects of BCL that need further consideration and which we plan to study in future. One particular issue is whether, and if so how, the current BCL expression of policy should be structured to better support policy specifiers in distinguishing the triggering conditions from the policy behaviour conditions. In other words, we need to investigate whether the current compact policy expression of BCL, which consists of both the triggering events for the activation of policy and the events that directly refer to the actions of roles to which the policy applies, needs to be separated into the respective components.

Another issue that needs further investigation is whether there needs to be a better separation between subject and target roles in a policy expression. BCL's construct of event role parameters in the event specification provides a good starting point, but this needs more detailed exploration.

We also plan to study how policy conflicts and priorities could be supported in both FCL and BCL. We believe that an approach where BCL is combined with an efficient non-monotonic formalism (Defeasible Logic) specifically designed to reason in presence conflict via priorities[7] can prove beneficial for the monitoring of contracts and can lead to further development of BCL.

Finally, there are other normative concepts such as the notions of right, delegation, and authorisation against which it would be useful to test the expressiveness of BCL and we also plan to investigate this in future.

Acknowledgements

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. Andrew Berry and Zoran Milosevic. Extending choreography with business contract constraints. *International Journal of Cooperative Information Systems*, 14(2-3):131–179, 2005.
3. José Carmo and Andrew J.I. Jones. Deontic logic and contrary to duties. In D.M. Gab-

26 *G. Governatori and Z. Milosevic*

bay and F. Guenther, editors, *Handbook of Philosophical Logic. 2nd Edition*, volume 8, pages 265–343. Kluwer, Dordrecht, 2002.

4. Content Reference Forum. Contract Expression Language (CEL) –an UN/CEFACT BCF compliant technology, January 21 2004.

5. Andrew D.H. Farrell, Marek J. Sergot, Mathias Sallé, and Claudio Bartolini. Performance monitoring of service-level agreements for utility computing using the event calculus. In *1st IEEE Workshop on Econtracting (WEC04)*, pages 17–24, July 2004.

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

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

8. Guido Governatori and Zoran Milosevic. An approach for validating bcl contract specifications. In Claudio Bartolini, Guido Governatori, and Zoran Milosevic, editors, *2nd EDOC Workshop on Contract Architectures and Languages (CoALA 2005)*, 2005.

9. 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.

10. Guido Governatori and Antonino Rotolo. Logic of violations: A Gentzen system for reasoning with contrary-to-duty obligations. *Australasian Journal of Logic*, 4:193–215, 2006.

11. 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.

12. 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.

13. IBM. Web service level agreements, Accessed 31 March 2004.

14. ISO/IEC 10746-1 10756-2 10746-3 10746-4. Basic reference model for open distributed processing.

15. ISO/IEC IS-15415. Open distributed processing-enterprise language, 2002.

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

17. 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.

18. 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.

19. David Luckham. *The Power of Events.* Addison-Wesley, 2002.

20. 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.

21. 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.