Dealing with contract violations: formalism and domain specific language*

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

This paper presents a formal system for reasoning about violations of obligations in contracts. 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. This language, called Business Contract Language (BCL) was previously developed to express contract conditions of relevance for run time contract monitoring. The aim of this mapping is to establish a formal underpinning for this key subset of BCL.

1. Introduction

The wide penetration of new broadband networks and new computing technologies such as XML, Web Services, Service Oriented Architectures and Even-Driven-Architectures have enabled better and more versatile collaborative arrangement between enterprises. Examples are virtual organisations, supply chains and extended enterprise.These cross-enterprise collaboration models bring tighter degree of integration between partners' business processes including more transparency of their data and processes than in the past. Such models also make it possible to accomplish 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 need to add or modify the existing architecture in which case they reflect an evolutionary character of occurrences.

These new collaboration models however give prominence to a number of problems some of which are new and some which may have been partially addressed in the past. One such problem is the analysis of business contracts as a governance mechanism for cross-organisational collaboration. At present, contracts are typically treated as legal documents and there is a weak link between them and the cross-organisational interactions that they govern. As a result, there is a renewed interest in contract architectures and contract languages as 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. Further, 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. 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. This language, called Business Contract Language (BCL) was previously developed to express contract conditions of relevance for run time contract monitoring [15, 19]. BCL can be regarded as a domain specific language, designed to support abstractions needed for the expressions of business contracts. It was developed by considering many contracting scenarios and taking into account the policy and community frameworks in [16, 13]. The initial research prototype for the BCL as part of Business Contract Architecture was developed and tested using several contract examples as presented in [15, 19, 18, 1]. The aim of this paper is to establish a formal underpinning for this key subset of BCL.

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. The main idea behind this formalism is to express contracts semantics in terms of deontic modalities (or normative constructs) such as obligations, permissions and prohibitions. In addition, this formalism supports the expressions of and reasoning about violations of such deontic modalities and the subsequent

*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 Cooperative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training).

actions that need to be taken to deal with violations. The formalism is based on the work by Governatori and Rotolo [7] who have proposed a formal system for reasoning about the Contrary To Duty (CTD) obligations. 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 logic (FCL).

This formalism is then used as a basis for checking the expressive power of relevant parts of Business Contract Language (BCL) previously developed and presented in [15, 19]. BCL is developed to support representation of key modelling concepts needed to express contracts as a governance mechanism for cross-organisational interactions and in particular to facilitate automated contract monitoring. BCL is briefly described in section 4 and BCL fragments for the example contract are presented in section 5.

We chose BCL as a mapping target for the FCL because BCL language is perhaps the most comprehensive language for the expression of contracts for the purpose of real-time contract management applications. The language is based on a precise policy framework proposed in [16], and further refined and tailored for business contact management domain [12], as well as on a rich expressive power of an event language for the specification of event-based behaviour as part of policy expressions. It is worth noting that although the language has its basis on these well-founded concepts, it was also developed in incremental manner, as we were considering increasingly complex contract scenarios and case studies from the contract management domain.[1] 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 section 6 we establish a correspondence between the semantics of FCL and the core concepts of BCL. Section 7 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

This paper is based on the analysis of the following sample contract, based on [18] and revised in [6].

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)

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

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

2.1 Price is a reference to the currency of the Australia unless otherwise stated.
2.2 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

3.1 The commencement date is scheduled as January 30, 2004.
3.2 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.com/catalog1.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 days.
5.3 If for any reason the conditions stated in 4.1 or 4.2 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 are used to specify the obligations, permissions and prohibitions that the signatories should be hold responsible to and to state the actions or penalties that may be taken in the case when any of the stated agreements are not being met.

We will focus on the monitoring of contract execution and performance: contract monitoring is a process whereby activities of the parties listed in the contract are governed legally, so that the correspondence of the activities listed in the contract can be monitored and violations acted upon. In order to monitor the execution and performance of a contract we have to have a precise representation of the 'content' of the contract to perform the required actions at the required time.

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 own 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. This, following the *Conceptualization Principle* of [8], effectively means that the language should allow their users to focus only and exclusively on aspects related to the content of the contract, without having to deal with any aspects related to their implementation. As stated in [8], 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 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 [14] 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 modal operators is not crucial in this paper. All we need is that the modal operators obey the usual mutual relationships

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

are closed under logical equivalence, i.e., if $A \equiv B$ then $OA \equiv OB$, and satisfy the axiom $OA \rightarrow PA$ (if A is obligatory, then A is permitted) implying the internal coherency of the obligations in a contracts: 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, are specific to some of the signatories of the contract. To capture this we will consider directed deontic operators [10]; 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.

Very often contracts make provisions about unfulfilled clauses: those provisions describe what some of the subjects of a contract have to do in case they breach the contract (or part of it), and 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 disattended, is know in the deontic litera-

ture as contrary-to-duty obligations (CTD) or reparational obligations. These are in force only 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.

Contrary-to-duties are one of the most debated fields of deontic logic, and, at the same time, they are subject to contrary-to-duty paradoxes. In response to the paradoxes many systems often with different intuitions and motivations have been proposed. The question whether an ultimate solution for all CTD paradoxes exists is still open. In this paper we do not touch upon this issue and we focus on a simple logic of violation that seems to avoid most of the well-known 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 the them is one the essential requirements for reasoning about and monitoring the implementation and performance of business contracts.

The idea behind the logic of violation [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, very 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 [7], and several authors have used exceptions to raise conditions to repair a violation in the context of contract monitoring [19, 9].

In the next section we lay down the foundations for our Formal Contract Logic (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.

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 last equivalence 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 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. Often these notions can be effectively represented in terms of complex combinations of directed obligations and permissions [5]. Hence violations to such complex notions result in violations to the obligations describing such notions.

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. Let examine some common patterns for this kind of construction (the general rule for merging clauses is given in (1).

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

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

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 the rules

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

From these we obtain

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

The schema in (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 we can consider the situation described by Clause 5.1 of the contract. Given the rules

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

and

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

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

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

where $\mathbf{X}$ denotes 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.

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

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

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 ($O_{Supplier}Replace3days$). 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 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:

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

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, 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 $r_2 : \Delta \Rightarrow D$ 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. In the examples above, we can drop rule r, whose normative content is included in r''.

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 two steps:

1) Starting from a formal representation of the explicit clause of a contract we generate all the all 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.

4. Business Contract Language (BCL) in brief

The purpose of Business Contract Language (BCL) is to specify business contracts in a way suitable to enable monitoring of contract execution in an event-based manner. Contract execution period starts after contract terms are agreed and 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 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 [12] and Enterprise Language standard [13] and is developed by considering a number of scenarios from business contract management domain. BCL also introduces the concept of

event pattern as a specific style of expressing state 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 many 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 [15]), 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 [17].

The main purpose of event patterns in BCL is to enable checking policies related to a contract. Policies define behavioural constraints for the roles that carry out activities in 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 contract and do not have explicitly modal character, such as start and end date of contract and these are easier to incorporate as part of contract monitoring. Considering policies, they 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 roles involved in business processes governed by contracts.

As a result of policy checking procedure 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 type. If occurred, this event can then be treated as 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 contrary-to-duty or reparation policy before. 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 was activated the same monitoring machinery can apply 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*, which establishes linkage of contract with organisational structures. Community is 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, 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 is a specific kind of community.

5. BCL Fragments

We illustrate the use of BCL through the example of contract service from section II with fragments of language expressions introduced progressively and discussed alongside these fragments. 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 4, 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.

```
EventType Id=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 to be distinguished 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. In our example this can be achieved as follows:

```
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 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 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, as discussed in more detail in [1].

Policy A *Policy* is used to specify business-level constraints in a BCL *Community* [15]. 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 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 behavioural condition expression, i.e., from the event pattern. We refer to that part of behaviour expression as trigger. So, in our 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*.

Obligation A BCL *Policy* can have an *Obligation* modality, indicating 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 *PurchasOrder* 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 *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 *GoodsAvailable* event signifies availability of goods which may be manually or automatically entered into the system and *PurchaseOrder* event signifies for example that a message carrying *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 the time period, then a violation event will be generated.

Permission A BCL *Policy* can have a *Permission* modality, indicating 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* can have a *Prohibition* modality, indicating 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 only Purchasers below legal age are prohibited of 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 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).

One specific use of guards can be to specify conditions for the activation of reparation or contrary-to-duty 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 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: HasOcurred QualityOfServicePolicyViolated
Behaviour: Replace.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 *HasOcurred* 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.

6. FCL and BCL

In previous sections we have shown how BCL can be used to describe basic deontic modalities of obligations, permissions and prohibitions. We have also shown how it can be used to describe violation conditions and reparation policies. Here we give a mapping from FCL to BCL.

The $\otimes$ operator introduced in Section 3.1 provides a formal foundation for expressing primary obligations and violation conditions. This violation condition in turn can express subsequent policies that come in effect when this

condition is true. Further, the logic supports recursive expression of such violation conditions.

In terms of BCL, we have shown in the previous BCL fragments 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. Here the occurrence of *PolicyViolation* can be used to set to true the guard condition that applies to the reparation policy. Similarly as in FCL, it is possible to specify a chain of reparation policies. This capability illustrate further expressive power of BCL.

6.1. Mapping Contract to BCL

In this section we present a mapping from the FCL to BCL. 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 $\mathscr{C}$ from FCL to BCL is determined by a function *map* that parses each rule r_i in $\mathscr{C}$ and return an expression in BCL, according to the format of the elements in r_i.

Given a rule

$$r_i : A^i_1, \ldots A^i_n \vdash B^i$$

where, r_i is the id of the rule, A^i_js are either modal literals or literals and B^i is an $\otimes$-expression, we use $Ant(r)$ to denote the set of literal in the antecedent of the rule and $Con(r)$ to denote the consequent of the rule. Thus given the above rule r_i, we have

$$Ant(r_i) = \{A^i_1, \ldots, A^i_n\}, \quad Con(r_i) = B^i$$

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. To define this mapping we adopt the fact that there are fixed but arbitrary bijections from the set of events symbols and propositional letters in FCL to events and states in BCL, and from subjects of modalities in FCL to roles in the community of BCL corresponding to the contract $\mathscr{C}$. Thus we have that given a modal literal X_sA, $role(X_sA)$ returns the role in BCL corresponding to the subject s, $behaviour(X_sA)$ returns the event or state corresponding to the literal A, and $modality(X_sA)$ returns *Obligation* if $X = O$, *Permission* if $X = P$ and *Prohibition* if $X = F$.

The antecedent of a rule is a set of 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 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. In case 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. In case a rule specify a CTD (i.e., the consequent of the rule is an $\otimes$-expression) then *map* beside returning the policy corresponding to the primary obligation will 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 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: $parseState(Ant(r_i))$
    Behaviour: $behaviour(Con(r_i))$

otherwise, when there a reparation obligation is involved, namely when $B^i = O^rC^i \otimes D^i$ (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: $behaviour(Con(r_i))$
  $vmap(D^i, r_i, 0)$

Here, the second function *vmap*, is referred to violations and, in a similar way we have $vmap(B^i, r_i, n)$, where B^i is a deontic formula, r_i is a rule, and n is an integer, depends on the format of its first parameter. If $B^i = M^rC^i$ then, the expression 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., $B^i = O^rC^i \otimes D^i$) it produces

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").

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

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

The element of the rule are:

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

Here $Con(7.1)$ is a modal literal thus 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 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

7. Related Work

Other contract languages have been proposed recently, most notably the Contract Expression Language [4], Web Services Level Agreements [11] and ecXML [3]. BCL has a number of similarities with these. For example, regarding the the event-oriented style 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. In terms of the logical approach of the FCL presented in this paper, this work has similarity with the early work of Lee [14], who proposed the use of deontic formalism for the specification of contracts. However, to the best of our knowledge our work is unique in that we apply recently developed logic of violation to specify aspects of contracts that deal with violations. [9] 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.

8. Discussion and Future Work

In this paper we have presented a formal system for the representation of contracts including the representation and reasoning about violations of obligations in contracts. The main aim of the paper is to use this system to provide logic-based formal foundation for the aspects of a domain specific language, BCL, developed to support business contract specification for contract monitoring purposes. Our

investigation of current features of BCL has found high a level of expressiveness of the BCL for this purpose. In particular we have found that the BCL expression of obligations, permissions and prohibitions is sufficient to express most of these deontic concepts. In addition, we have found that BCL provides a good solution for the expression of violations and the corresponding reparation or contrary-to-duty obligations. This solution is based on the use of the concept of 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 searching for 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 incremental development of this language. Coupled with a precise enterprise modelling framework as a starting point, in this development we have considered 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 decoupled language to reflect separation of main concerns such as separate structuring into community, policy, state and event pattern sub-models and 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 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 role to which the policy applies, needs to be separated in 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 the FCL and BCL. We believe that the approach of [6] where BCL is combined with an efficient non-monotonic formalism (Defeasible Logic) specifically designed to reason in presence of conflict via priorities can prove beneficial for the monitoring of contract and can lead to further development of BCL. This work also shows how RuleML, a general markup language for rules intended as tool to exchange rules over different systems and architecture, can be extended to represent contracts, and the logic based on the combination of BCL and Defeasible Logic can be used to reason about contracts.

References

[1] A. Berry and Z. Milosevic. Extending choreography with contract constraints. *Int. J. of Cooperative Inf. Syst.*, 14, 2005.

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

[3] A.D.H. Farrell, M.J. Sergot, M. Sallé, and C. Bartolini. Performance monitoring of service-level agreements for utility computing using the event calculus. In *1st IEEE WEC04*: 17–24, 2004.

[4] Content Reference Forum. Contract Expression Language (CEL) –An UN/CEFACT BCF compliant technology, 21/12/2004.

[5] J. Gelati, G. Governatori, A. Rotolo, and G. Sartor. Normative autonomy and normative co-ordination: Declarative power, representation, and mandate. *AI& Law*, 12(1-2):53–81, 2004.

[6] G. Governatori. Representing business contracts in RuleML. *Int. J. of Cooperative Inf. Syst.*, 14, 181–216, 2005.

[7] G. Governatori and A. Rotolo. A Gentzen system for reasoning with contrary-to-duty obligations. A preliminary study. In Δ*eon'02*: 97–116, 2002.

[8] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5/WG3-N695, ANSI, 1982.

[9] B.N. Grosof and T.C. Poon. SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *12th WWW*: 340–349. ACM, 2003.

[10] H. Herrestad and C. Krogh. Obligations directed from bearers to counterparts. In *5th ICAIL*: 210–218. ACM, 1995.

[11] IBM. Web service level agreements, Accessed 31/06/2004.

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

[13] ISO/IEC IS-15415. Open distributed processing-enterprise language, 2002.

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

[15] P. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract for extended enterprise. *Data & Knowledge Engineering*, 51:5–29, 2004.

[16] P. Linington, Z. Milosevic, and K. Raymond. Policies in communities: Extending the odp enterprise viewpoint. In *EDOC98*, 1998.

[17] D. Luckham. *The Power of Events*. Addison-Wesley, 2002.

[18] Z. Milosevic and G. Dromey. On expressing and monitoring behaviour in contracts. In *EDOC2002*, 2002.

[19] Z. Milosevic, S. Gibson, P. Linington, J. Cole, and S. Kulkarni. On design and implementation of a contract monitoring facility. In *1st IEEE WEC04*: 62–70. 2004.