Identifying requirements for Business Contract Language: a Monitoring Perspective

S. Neal., J. Cole, P. F. Linington, Z. Milosevic, S. Gibson, S. Kulkarni.

University of Kent,
Kent, CT2 7NF, UK.
{sn7, pfl}@kent.ac.uk

Distributed Systems Technology Centre,
University of Queensland,
Brisbane, QLD 4072, Australia.
{colej, sgibson, zoran}@dstc.edu.au.

Abstract

This paper compares two separately developed systems for monitoring activities related to business contracts, describes how we integrated them and exploits the lessons learned from this process to identify a core set of requirements for a Business Contract Language (BCL). Concepts in BCL needed for contract monitoring include: the expression of coordinated concurrent actions; obliged, permitted and prohibited actions; rich timeliness expressions such as sliding windows; delegations; policy violations; contract termination/renewal conditions and reference to external data/events such as change in interest rates. The aim of BCL is to provide sufficient expressive power to describe contracts, including conditions which specify real-time processing, yet be simple enough to retain a human-oriented style for expressing contracts.

1. Introduction

Commercial interactions are typically performed with respect to a prearranged agreement, or contract. Among other things, contracts are used to specify obligations imposed upon signatories and penalties that they will incur should these obligations not be met. Such obligations may be imposed on a periodic basis or in reaction to certain events between the signatories.

The use of computers to manage business interactions is standard practice for many organisations; however, the support within these systems for contractual semantics is implicit. We believe that, in order for contracts to be properly managed, their definitions should be made explicit and a framework should be provided to help support their specification, interpretation and general maintenance. Such a framework will enable real world contracts to be encoded and used to police the behaviour of the computer systems that are carrying out the duties pertaining to the contract.

This paper is based upon ongoing work carried out in a collaborative project between DSTC and UKC. Previous work by the authors had involved the implementation of contract management frameworks and it was noticed that there were many common features between these two prototypes. Our investigation, therefore, initially focussed on a comparative study of the approaches adopted and pinpointed the relative strengths and weaknesses of these. This comparison included integrating the prototype tools to monitor the same business system and experimenting to see which kinds of contract they were best suited to manage.

From these experiments, we identified a base set of requirements for a contracting framework, a major part of which was concerned with the specification of a language which can be used to express contracting semantics - the Business Contract Language (BCL).

We start this paper (in section 2) with a background discussion on currently available commercial applications, and related academic projects. We follow this (in section 2.3) with a discussion of requirements for a contract management framework and associated implementation issues. Section 3 presents a comparative overview of the two prototype implementations and how we decided to integrate them.

In section 4, we examine the motivation and key requirements for BCL; this is followed (in section 5) by a case study for a QoS contract which illustrates how BCL features can be used to describe complex contracts and how a supporting framework can be used to detect infringements of the obligations laid down in the contracts.

2. Background

The field of contract management is increasingly gaining the attention of both the commercial and academic communities. Commercial interest is driven by the opportunity to address one of the missing key ingredients

needed to support new forms of enterprise - variously coined as real-time extended enterprise, value chains, or virtual organisations. Academic interest is in part driven by this, but also by the capabilities of new Web Service technologies which provide an impetus for a shift in the research focus - solving problems of higher-levels of abstraction, increasingly of enterprise concerns. As a result, there is a renewed interest in the topic initially dealt with in [9] – namely studying semantics of contracts and their architectural implications for e-commerce systems.

2.1 Commercial Systems

At present, enterprise contract management functionality is mostly incorporated as part of ERP systems such as Oracle Contracts [12]. These are mostly database applications that store information about contracts, e.g. contract name, type, organisational roles involved and some contract significant dates, and provide certain notification capabilities.

In addition, some commercial offerings such as DiCarta [5], iMany [4] and UpsideContracts [6] have emerged providing specialised contract management software. These software products aim at supporting full contract life cycle management - ranging from collaborative contract drafting and negotiation (e.g. using Microsoft Word), storage of contracts, milestone-driven notifications, certain analytic features and some limited monitoring capabilities.

However, these products provide limited support for contract monitoring features, in particular in terms of event-based monitoring and rule-based checking of parties' actions as per agreed contract. In addition, they do not provide comprehensive support for seamless integration of contracts as part of the enterprise systems, including both internal and cross-organisational systems. This is perhaps because there is a lack of an overall model that expresses semantics of contracts as a governance mechanism for cross-organisational collaboration.

2.2 Academic projects

The following academic papers provide a good starting point for expressing such a contract model. [3] has proposed a logic model of contracts based on Petri Nets and the deontic logic formalism. [2] proposes the use of Genetic Software Engineering to specify and verify contracts and an event-based and policy-oriented model based on deontic concepts – for contract compliance monitoring.

This paper summarises our latest ideas in this direction and further extends the contract model presented in [2] to better deal with i) complex event patterns, in a similar way as the approach taken by [10] and [7] and ii) more powerful enterprise model expressing policy based on [1].

2.3 Contract management frameworks

To store and manage business contracts electronically, a framework must be implemented which provides generic contract management functions. In this section we detail what we believe are the fundamental services that such a framework will need to provide; we then discuss the implications of this with respect to providing a scalable implementation of such a framework which will be capable of operating in a heterogeneous environment.

2.4 Framework requirements

Industrial distributed systems are typically built upon numerous middleware technologies (including CORBA, J2EE, .Net and Web Services) and utilise a wide range of software (e.g. workflow engines, billing systems). If a contract management framework is to be applied in such an environment, then it must be capable of adjudicating between all such system components. Integration of a contract management framework should involve minimal disruption to existing systems. It is unlikely that organisations will be prepared to shut down their systems to allow contract management code to be added.

For contracts to be managed in a heterogeneous environment, an independent process is required which can interface with all components to determine the key elements of their behaviour. One way of implementing such a process is as an event listener/monitor which receives notification of contract related behaviour from the components using a publish/subscribe mechanism. This is an approach widely adopted in industry as it provides a flexible mechanism for receiving details of events which can be encoded in a platform neutral format, such as XML.

2.4.1. General system requirements.

The purpose of a contract monitoring system is to determine what has, and is currently, happening in the enactment of activities associated with a contract. If the contract activity is reported using events, then we should consider how comprehensive and reliable the event generation and reporting mechanisms are expected to be. Amongst the factors to be considered are accuracy of the reported events, impact of the event generation process and organizational threats from it. We can look at these aspects in turn.

Accuracy of event reports - One aspect of the accuracy of event reporting is the timeliness and ordering of events. The difficulty of maintaining a consistent view of time in distributed systems is well-known. Steps may be taken to synchronize clocks, but residual skew means that distributed time-stamps can never be completely accurate. Events can be passed to a common point so that time-

stamps are originated by a common clock, however this risks delay and reordering during the transmission process that is often more severe than the skew of coordinated clocks would have been.

The practical implication of this is that the ordering of events reported close together in time, or the relative ordering of an event and a timeout for its receipt, are unreliable. This must be considered by the monitor when making decisions which are dependant upon the timing and ordering of events. There is, therefore, a need for an agreed latitude in timings, negotiated with knowledge of the properties of the infrastructure in use, and of the contract details.

Performance issues - Applying monitoring on a large scale requires the use of mechanisms that do not introduce performance bottlenecks. Solutions include the local processing of events to generate higher level and summary reports of activity, but this may imply that checks performed are weaker than might be possible on the raw data. In particular, operating on summary data may imply a need to relax timing checks still further.

Even at a smaller scale, fine grained timing constraints may imply a heavy load and optimised constraint checking is essential.

Security issues - The very fact that contracts are being monitored implies that there is not absolute trust between the participants, and so the trust assumptions made in designing the system need to be explicit. From the monitoring point of view we need to consider the trustworthiness of all the parts of the system, including the parties to the contract, the components of the infrastructure they use, the repositories holding contract information and the monitoring system itself.

A participant in a contract will need to have confidence in any event reporting mechanisms that have to be incorporated into their systems. This is more likely if the mechanisms are provided, or at least certified, by a trusted third party. Similarly, there will be a need to establish confidence in the event reporting mechanisms and there may well be business opportunities in the establishment of reporting services of established probity and reputation.

The monitoring component can also be a trusted third party, or it can be acting as an agent on behalf of one or more of the contracting parties. Depending on its status, it may give priority to the detection of particular kinds of violation, and even ignore violations that are in the interests of its principal. The level of trust placed in the monitoring system by the various parties will influence the way they respond to its reports of contract violations.

In cases where there is limited trust of the infrastructure it will be important for the party performing an action to ensure that the corresponding event carries a proof or authenticity and a guarantee of non-repudiation, such as a secure signature. The current prototypes have not investigated these aspects.

Finally, the information about a party's activities provided by the events reported may be considered confidential. A service provider, for example, may well commit to a service level agreement without wishing to disclose the actual achieved performance. Knowledge of the real performance has commercial value over and above the statement that the contract is fulfilled. Such a party must be able to trust the monitoring system to preserve confidentiality.

Having established the key low level requirements, we turn now to higher level issues; namely those which are solely related to contracts.

2.4.2. The contract lifecycle

There are several stages in the lifecycle of a contract: construction; negotiation; agreement; execution & management; and, finally, when all of the contractual obligations have been discharged, it expires. Note that some contracts may be ongoing and never expire.

A contract management framework should provide support for the contracts lifecycle; this may include repository based tools for contract templates as well as active contractual agreements (also referred to as contract instances). It must also provide support for a contract language capable of describing the required behaviour of the contract signatories. For example, the required, or permitted, sequences of events that the signatories are expected to exhibit in fulfilling the burdens placed upon them by the contract. This language will be interpreted by components within the framework and used to check the behaviour of the signatories to the contracts.

Our key requirements, then, may be summarised as follows:

- Platform neutral event monitoring mechanism
- Simple integration with existing systems
- Security/Confidentiality
- Contract lifecycle management
- Contract language

3. BCA and ECL Solutions

In this section we compare the respective contracting systems, starting with an overview of each and focusing on their capabilities for contract monitoring. We also discuss how we integrated these systems and compared their capabilities in monitoring business contracts.

3.1 BCA

The BCA contract management system is being developed by the Elemental project at the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC). It aims to support the entire contract lifecycle and to be configurable to the requirements of different contracting situations. The groundwork for the system was developed in [9], and the implementation is currently at a prototype stage.

BCA currently implements support for various tasks within the contract lifecycle, including contract definition, access to contract data, monitoring, and providing notifications of contract significant occurrences to the signatories. The system is managed via a web-based user-interface.

BCA can be considered as an extensible contract management platform. The system was designed to ensure that new features could be easily integrated into the system in the future.

Events and states are the central motifs of this design, and help achieve this extensibility. The contracting platform is implemented by an infrastructure that contains components for generating events and updating the values of states. The infrastructure components along with those implementing the contract services communicate with one another via events. Events are typed, and components subscribe to events, specifying the types of the events they are interested in receiving. The infrastructure performs the event distribution.

The event-based inter-component communication decouples the infrastructure and service components from each other, which results in components only needing to be aware of the events they wish to consume.

Components perform processing in response to receiving events. During processing, they may access data both from the event itself, and from repositories which are available to BCA. As all event generation and state updates are maintained by the system, it is possible for all components to access information regarding them from such repositories; for example, to check causal relationships between previously observed events.

The components within BCA are configured with an XML language which defines events, states, contract conditions to monitor, and notifications to generate. These definitions make use of a syntax we have developed for specifying expressions and event patterns. For example, a state is updated in response to some event pattern, a policy is evaluated in response to an event pattern, and a relational expression might be used to specify the condition that a policy is checking. The event subscriptions for each component may be deduced from their configuration details.

States are implemented in BCA as components which define how and when their values are to be updated. An event pattern is used to specify when the state should be updated. In most systems, these things would be entangled within code that resides outside of the state definition. This helps decouple states from the rest of the contracting system.

By considering state as a distinct component in its own right, we can develop higher-level syntax for specifying them. An example of this is a special type of state called a recurring state. Recurring states provide a convenient means to specify states which pertain to a recurring period of time; for example, number of purchases made in each month. To define a recurring state, we specify the usual details of when and how to update the state, but we also specify when to create a new copy of it. There are also mechanisms for accessing both the current copy of the state as well as copies for previous periods.

Components within BCA may also generate notifications. These are similar to events but are intended for human consumption in the external environment, e.g. via e-mail or SMS.

3.2 ECL

The ECL (Enterprise Contract Language) system is being developed at the University of Kent. It aims to provide sophisticated support for the monitoring of contracts. ECL grew out of PhD work [8] on checking conformance of software to design patterns, and its current conception is a result of revising these concepts [1] in order to apply them to contracts. The implementation is currently at a prototype stage.

The PhD work was concerned with the specification of behavioural design patterns, and how such specifications may be used to dynamically verify the correct behaviour of a particular software implementation. Behavioural patterns describe the runtime dynamics of software rather than the more static aspects such as class hierarchies; a well-known example of a behavioural pattern is the Observer pattern [13]. The work carried out in this thesis resulted in the development of the Pattern Constraint Language (PCL) and the implementation of in interpreter to prove that the language could be used to effectively detect illegal software behaviour. Although PCL provides the conceptual basis for ECL, the code-base for ECL is entirely new.

An important finding from this thesis was that the interpretation of behavioural patterns was, in certain circumstances, ambiguous. Such ambiguity would occur as a result of behavioural specifications that could not

distinguish between behavioural branches at a point where some future event had not yet taken place. This problem manifests itself not because of poor specification, but because it is an inherent part of the nature of some behavioural patterns; this becomes particularly complex where a calculation is performed resulting in some part of the pattern's state being updated. Under such conditions, the PCL interpreter allows all branches to be simultaneously taken and then later resolved as future events eliminate the ambiguity. When the ambiguity is resolved, the correct values for the state of the pattern can be determined. Much of the work from this PhD involved the development of techniques which permitted this problem to be solved in a computationally efficient manner.

A similarity was noticed between the problems addressed by PCL and the problem of monitoring contracts, and this led to the development of ECL. In both areas there is some specification of required behaviour (of the pattern or the contract), and there is a requirement to determine if this is being adhered to. This similarity is deeper than it may initially seem, as both patterns and contracts specify certain obligations as well as certain permissions and prohibitions on the behaviour of entities in a system. It is intuitive that contracts consist of obligations, permissions and prohibitions, but not as immediately apparent that patterns do too. Proof of this is evident, however, in the Observer pattern, where there exists an obligation to notify all Observers of any changes to the Subject, and in the Singleton pattern where there is a prohibition to create more than one instance of the singleton class.

ECL is based upon a Community Model [1] which extends the concepts described by the ODP enterprise language [11]; with this model contracts are modelled as a community of parties bound to some contract. The model supports essential aspects of contracting, such as permissions, burdens and delegation. Under this model, a community comprises of a number of roles and describes some overall goal that it is trying to achieve. The permissions, prohibitions and obligations imposed on these roles will describe the bounds placed upon the communities members in order that they may achieve this goal. The roles of the communities also specify cardinality which may impose their being filled upon instantiation of a community instance. Communities may also be regarded as members of a community; this allows community nesting, with a sub-community filling a role within its parent community.

Architecturally, the ECL system consists of three main parts: a target system, a community model and an interpreter. The community model stores the definition of the community representing the contract, and this model is made use of by the interpreter, whose primary function is to receive input events from the target system and determine whether the contract has been violated.

When an event is received by the system, the interpreter checks to see which communities have specified an interest in the event, it then checks to see what impact that event has on those communities. The most significant impact the event may have is violating the conditions of the putative community, but additionally it may trigger the removal of an obligation from a role, trigger the assignment of a party to a particular role, or perform any other manipulation of the community model. To check whether the event violated the community's conditions, the interpreter will check that the roles playing a part in the action have the necessary permission to perform that action. The interpreter also needs to interpret the policies within the community which will include valid patterns of events that the roles are expected/obliged to perform; these patters are described using operators which may be used to specify sequential and parallel sequences of events, in addition to complex timing predicates.

Because updates to the community are considered actions like any other, they may also be subject to the communities policies. This allows us to specify policies which determine the ways in which the community may legally evolve.

3.3 Comparison of existing solutions

Both systems are designed to facilitate electronic contracting, and while ECL focuses on generic monitoring using an underlying community model, the focus of BCA is to provide support across the entire contract life-cycle.

ECL allows the contract writer to add blocks of ordinary Java code into policy definitions. This feature effectively allows the interpreter to do anything that can be done with Java; this allows us to quickly experiment with new language features before deciding whether they will be fully implemented as high level ECL language operators.

ECL is based upon a formal model of behaviour and a community structure, whereas the BCA approach was more pragmatic and was driven by the need for a flexible, extendable architecture on which to build contract management features. The design of the language for specifying the semantic behaviour of the components within the BCA architecture is continually evolving to meet the needs as domain knowledge is gathered. Given the requirement for security additions to the BCL language, future versions of BCL will include a similar community model to that used by ECL.

Both of the systems interact with the systems they are observing in a similar manner, receiving their input as structured events. The main difference is that ECL takes into consideration a broader community model. BCA will be gradually incorporating a similar model which will

provide a more expressive means for the specification of event patterns.

Both systems have comparable expression operators, covering the usual range found in a typical programming language, e.g. relational operators (>, >=, =, etc) and mathematical operators (+, -, /, etc)

ECL's structure focuses on behaviour and so is akin to that found in a programming language, except it is tailored towards the specification of behavioural constraints for communities. It has more powerful support for event patterns, and has support for sequences of events, and parallel execution of events, whereas BCA will be incorporating this in near future. On the other hand, BCA's style is more declarative, and so consists more of distinct specifications of items such as events, states, policies and notifications.

BCA has more explicit expression of state variables and both events and states are used as building blocks to express policy and other contract-related constraints. Indeed, the link between the parts of a BCA specification is via the events that trigger them, referred to by the event's identifier, and the data that they use, referred to by its identifier. ECL's approach, on the other hand, is more behaviour-centric; the primary organizational element is the behaviour specification, and state is referenced wherever it is updated. In short, BCA's architecture is based around the data provided by its infrastructure, while ECL's is based more around the specification of the required behaviour of the system.

As a result of this, states are handled quite differently by the two systems. In BCA, states are distinct entities whose values and updating are managed locally by the state component. In ECL, states are stored within a community instance, and the code for updating a state may be contained within any of the policy definitions in that community.

Because of its behavioural orientation, it is simpler to express causal policy relations in ECL; for example, a policy which is triggered in response to a contract violation may be syntactically adjacent to a policy which detects the violation. In BCA, this is not the case as policies filter which events they react to according to their identifiers. Therefore a policy which reacts to a violation will detect that violation by receiving an event with a particular identifier; it is not possible to simply refer to a policy by name in the same community file.

Another area of comparison concerns the dynamic modification of systems. While BCA adopts a loosely coupled architecture, ECL is more tightly coupled. BCA's loosely coupled design enables straightforward dynamic modifications to contracts, e.g. adding new definitions, modifying existing ones and removing old ones. A drawback of this design, though, is that unchecked logical errors might occur.

Specific features - Time is handled differently by the two systems. In ECL, event patterns can include temporal constraints, allowing them to express things such as "after event A has occurred, we must receive event B within 5 seconds".

BCA currently does not have a special mechanism for expressing temporal constraints within event patterns. In BCA the points in time that constitute a temporal constraint are represented as events. Thus, to handle temporal constraints consistently they are specified as event patterns, with events being defined for the relevant time points in the pattern. So, to express the constraint given above, we must set up a temporal event to be generated 5 seconds after A occurs. If we receive this event before B, then the constraint has been violated. The aim is that with temporal matters handled in a consistent fashion with events, all the mechanisms for dealing with events and different types of event patterns can also be applied to temporal issues.

In summary we found that there were no fundamental conflicts between these two languages and that it is possible to align them as will be discussed in section 4 .

3.4 Integration

In the previous section we outlined the different approaches used in the design of the BCA and ECL toolsets. The aim of our research was to integrate these systems and carry out a comparative investigation of their capabilities. Architecturally, the main similarity between the systems was that they both received an incoming event stream which held details of the contract related events in the system under observation. In this section, we describe how we exploited this similarity in order to integrate the components of the respective systems, we highlight the difficulties that we encountered and we discuss what future enhancements could be made in order that the systems work more closely together.

Integrating the software components was fairly straightforward. As BCA had a more comprehensive suite of test case scenarios, we chose to use this as the main source of contract events. This meant that the ECL interpreter component had to be integrated with the BCA event stream. In order that the two tools could be used simultaneously, an extra event listener was added to the external event handling component in BCA; in reality this constituted an extra event sink listening to a JMS topic (a queue which guarantees that all subscribers will see all events).

Figure 1 - High level integration architecture

The above architecture ensured that both BCA and ECL event interpreters were guaranteed the same views of the system under observation.

An immediate problem that we encountered was that BCA events would only detail the type of the event, a timestamp and a contract id. In order that the ECL toolset could work with the events, extra information was required detailing which signatories were responsible for which role in the events that had been reported. For example, in a 'goods-paid-for' event, ECL required information regarding which signatory had been the payer and which had been the payee. The BCA toolset comprises a number of components, one of which stores contract signatory information (the Notary). By intercepting messages as they were added to the Queue, we could adjust their formatting, contact the Notary, and add the extra information that was required to the events.

There are further ways in which these toolsets may be integrated, including: exposure of internally generated events; data sharing, possibly via a common data gateway component; and also the inclusion of a common community model. The current state of integration allows us to achieve our key objective for this exercise, namely the capability to perform black box comparison of the toolsets.

4. A business contracts language

4.1 Motivation

The collaborative work carried out not only highlighted the differences in approaches that had been taken towards implementing a contract framework, but also the difference in expressive capabilities of our respective contract languages. It was noted that each of these languages had particular strengths and weaknesses and were each better suited to expressing specific types of requirement in contracts. In response to this, we formulated a common set of requirements for contract specification languages which we used to drive the development of a new language for contract specification: the Business Contract Language, or BCL.

The remainder of this section will discuss these requirements, the following section presents a case study which illustrates how BCL syntax deals with them.

4.2 Requirements

Contracts cover a wide range of subject areas but most will share many common features; for example, deadlines pertaining to actions that the signatories to the contract are obliged to perform are a feature found in all but the most trivial of contracts. By identifying these features, we form the core requirements for BCL.

Time **–** There are different types of time constraint and different ways of expressing them. A simple use of time might be to mark start and end dates between which the contract is deemed to be in effect; such markers may be static in nature or dynamic. For example, a tenancy agreement may specify a static start marker to indicate that the contract start date may not be moved, but could allow the end marker to be dynamic to permit the contract to be extended at the end of its regular period. Dynamic time markers may be the subject of other types of constraint, perhaps which apply penalties to signatories which alter them – e.g. a late delivery of goods penalty.

If our goal is to create high-level language to specify contracts, then we should try and find natural and succinct ways to express more complex temporal constraints; for example durations.

We propose a syntax which allows durations to be specified in a variety of intuitive ways, such as: 1 year, 2 months, and 7 days. Durations must be specified relative to some point on the time line; for example, we can tie the duration "7 days" down to a specific point by expressing something along the lines of: 7 days after goods have been received. Symbolic names may also be used to represent durations, such as: except on *Public Holidays*.

These basic temporal elements permeate the conditions of most contracts, but their occurrences may be specified in a number of different ways. The following example demonstrates a sliding time window operator which will allow constraints over a moving duration of time: in any 3 day period the total value of orders placed must *not exceed* a thousand dollars.

A sliding window can be considered conceptually as a simple polling construct which requires re-evaluation every time there is a clock tick. In practice though, implementing a sliding window, for all but the largest of clock granularities, will require a different approach in order that polling may be avoided.

One possible solution to implementing this might be to explicitly specify the times at which the window's predicates should be evaluated. If we consider the above example which restricts the total value of orders in any 3 day period, then we might decide that this should be evaluated every 24 hours at midnight.

Mon	Tue	Wed	Thu	Fri	Sat	Sun
200	600	700	-	-	-	-

Figure 2 - *Value* of orders received each day

If we assume that each order event is for 100 dollars, then the above figure illustrates how periodic checking of a sliding window's predicate (every midnight in this case) will detect that the limit of order value has been exceeded. However, the limit of 1000 dollars was reached when the orders for Wednesday totalled 200, but the violation of the predicate was only trapped after a further 500 had been spent. A simple solution to this shortcoming would be to ensure that the predicate is evaluated every time there is an event of interest received. This would allow the window's predicate to trap the exact event that caused the limit to be exceeded and move the contract into a state whereby subsequent order events should be denied or cause a contract violation.

The problem with evaluating in response to events is that sometimes it is the lack of an event that is of interest. To illustrate this, we consider another example which specifies that within any three day period there must be *at least* ten orders placed. If we evaluate this in response to order events being detected, then there is a risk that too much time may pass before a relevant event is generated. The figure below shows a count of order events received on consecutive days. If we rely upon events to trigger an evaluation, then the predicate will not be checked until Sunday; however, the predicate was clearly violated at the end of Thursday.

Mon	Tue	Wed	Thu	Fri	Sat	Sun
10	0	0	0	0	0	1

Figure 3 – *Number* of orders received each day

It should be possible to determine, given the state of a contract, exactly when a sliding window's predicate will need to be checked. So, in the above example, when the ten orders are placed on Monday we should schedule a timer to re-evaluate the predicate at the earliest time that it could be violated; i.e. on Thursday.

When timers are used to detect the absence of events, it will be prudent to re-evaluate their necessity in response to events that affect the sliding window's predicate. If, in the above figure, ten order events were received on the Wednesday, then the Thursday timer could be cancelled and a new timer set to ensure that the predicate is checked by Saturday at the latest. To summarise, once a predicate of this nature is satisfied, we can safely ignore any further checking of the window's predicate until such a time that any of the events which caused the predicate to be satisfied have left the window.

As we have seen, the necessity to re-evaluate the sliding window's predicate is influenced by events entering and leaving the time window. If the window moves every millisecond, then there is a possibility that the events may enter and leave the window at the same rate; this could lead to a situation where there are a vast number of re-evaluations of the window's predicate required. By limiting the movement of the window, we can also limit the required number of re-evaluations required; this allows us to greatly optimise the implementation of a sliding window operator.

Fortunately, business contracts seem rarely to require very fine grained sliding windows. A term such as "a one week period" in a business contract, may be interpreted in a number of ways. It could refer to mutually exclusive, consecutive slots of time (e.g. consecutive series of seven days starting on Monday) or it could mean a number of overlapping time periods of a shorter duration (e.g. within the last seven days). In the latter of these cases it is the length of the 'shorter duration' that will determine the granularity of the checking required. Typically, a business contract will be interested in complete days which allows the granularity of the windows steps to be set at 24 hours.

Once a predicate for a sliding window is satisfied, whether it will be re-evaluated or not is dependent upon the behaviour in which it is embedded.

Behavioural patterns – The behaviour of a signatory is defined by the actions that they perform and will be subjected to the restrictions that the contract imposes upon them. In order to define behavioural constraints upon signatories, the contract syntax must be capable of specifying complex patterns of actions.

In a simple case, such a pattern may just be a linear sequence of actions; for example: order, deliver, and pay. Many contracts, though, are more complex than this and may involve parallel threads of behaviour. In the case of a home purchase contract, for example, there will be a number of preliminary actions that need to be completed before the final purchase contracts are signed. These preliminary actions will include a successful mortgage application, credit checks, land registry checks, etc. These actions could complete in any order, but only once they have all completed may the final contract be approved.

In practice, we would expect behavioural patterns to have an equivalent expressive power to that normally associated with process algebra.

Authorisation & Accountability model – Concepts similar to those used in community models are already used to model authorisation. Role Based Access Control (RBAC) is a widely accepted means for controlling access to restricted resources; for example, in operating systems. We propose an extended model which expresses not only

the permissions associated with contracts, but also the obligations.

In order that penalties may be applied to signatories, contracts will need to specify accountability. For example, if a payment for goods is late, then the obligation to pay has not been met and there may be a financial penalty - perhaps interest on the amount due. The contract must be able to identify who is responsible for the late payment and also who is liable to pay the penalty (these may not be the same signatory).

A further requirement of the accountability model is delegation. In many situations, the responsibility for completing a task may be delegated. Under these circumstances, the contract must still be capable of identifying the accountable party. Delegation of a task does not necessarily indicate delegation of accountability. For example; a contractor may agree that work will be completed by a certain date but will delegate the actions required to complete the work to a sub-contractor. If the work is not completed as agreed, we must be able to identify whether the contractor delegated the accountability along with the responsibility to perform the actions to the sub-contractors. In addition to this, we will also need to be able to express whether the contractor is entitled to delegate either the actions or the accountability.

4.3 Our approach

The fundamental concept in BCL is the community. A community represents a collection of enterprise objects which share a common goal. An enterprise object is an object which can be used to represent anything of interest in the system we are modelling; this could be a human user of the system, a computational object or even an element of data. Importantly, an enterprise object can be used to represent a community; therefore, communities may be composed, at a high level, from other communities. The remainder of this section will use the term object generically to refer to all types of enterprise objects.

Within a community there will be roles defined. A role is a group within a community to which objects may belong that identifies the position of the object within that community. All members of a community must belong to at least one role in that community and, subject to the restrictions specified by the community, may belong to multiple roles at any time. Further to this the community may specify constraints on the cardinality of the roles to indicate minimum and maximum permissible number of members.

A community will also specify policies that its members must adhere to. The policies will not refer to the members directly, but will instead refer to the roles. This separation of policy from a particular object allows the model to evolve dynamically and for objects to move between roles over time.

In our model, a contract is simply a special case of a community where the members of the community are legally bound by the policies of that community. We can therefore use our model to represent models of individual enterprises as well as of the collaborations between them.

Permissions and obligations are expressed in a novel way as transferable objects that place constraints via the action semantics on the objects holding them. Following the usage in [1], these are called permits and burdens.

In addition, we unify temporal and other constraints such as those that relate to state and the actions of parties as defined in the community model - so that they can be treated as an integral part of policy and other contract constraints.

5. Case study

The implementation of BCL is work in progress. To illustrate some of the features that the BCL currently provides, we use an example that reflects a realistic quality of service contract. For reasons of clarity and space, we only show the elements of this contract that are of relevance to this discussion.

5.1 QoS example contract

Our case study will examine the specification of a contract which defines an agreement between a service provider and a client. The service provider provides web servers which must adhere to uptime guarantees, and the client has purchased space on these servers. The agreement will conform to the following criteria:

- The contract will be for a fixed period of twelve months from an agreed start date.
- The maximum permitted downtime for the server will be twenty minutes in any one week period.
- Downtime is defined by there not being HTTP access to the server.
- In calculating downtime, the contract will exclude any times where:
  - 48hrs maintenance notice has been given to the client,
  - emergency maintenance is required,
  - the client has not paid outstanding invoices by the agreed payment deadline,

- service has been made unavailable by events of Force Majeure.
- In the event that the agreed downtime limit has been exceeded, the service provider will, upon request of the client, credit the clients account with a pro-rated charge for one weeks service.
- All invoices are to be paid within 28 days.

Many different sequences of events will be possible, even in this simple contract. In order for a contracting framework to provide automated support for such a contract, it is important to define a set of events that the framework must generate and handle in response to key events between the contracting parties. For example, fundamental events might include details of when HTTP access to the servers is possible, and may also be used to signal when invoice payments have been made.

It will be a requirement of the framework that events are reported; this may imply a need for insertion of event generation code which will allow the reporting of the key events as they take place. Such code could be implemented within applications, but is more likely to be useful if intelligent stubs can be created which analyse the use of the components in the system under observation and send events reporting its behaviour to the monitor at appropriate times. Implementing event reporting code at this low level would allow it to be managed, and automatically inserted and removed from the system, without affecting the business logic or deployment of the application being monitored.

5.2 Specifying the contract with BCL

In order to specify a contract to maintain the above requirements, we must first determine which actions in the system under observation we are interested in and specify an appropriate type hierarchy for these.

In our prototype implementation, BCL is an XML based language. The decision to use XML was made in light of the excellent range of tool support available, including parsers, transformation engines and document validity checkers.

Actions – Actions represent observable behaviour in the system under observation. All actions in BCL are part of a single type hierarchy with a single root type known simply as 'action'; this hierarchy enables the generalisation of actions. Contracts will refer to the actions from this hierarchy in order to define bounds for the behaviour of their signatories. The following example illustrates the syntax required to specify a new action type:

```
<action-defn name="service" type-of="action">
  <action-role name="provider/>
</action-defn>
```

Note that the super-type for the new action is specified as 'action', the root of the action hierarchy. The definition for an action will contain action roles which allow the actions to be parameterised with the enterprise objects involved in the action; therefore, in this example, all reported service actions will contain a reference to the enterprise object that played the provider role in the action when it was performed. A role defined within an action is also present in all subtypes of that action. Action definitions which represent interactions must specify a role for each party involved in the interaction.

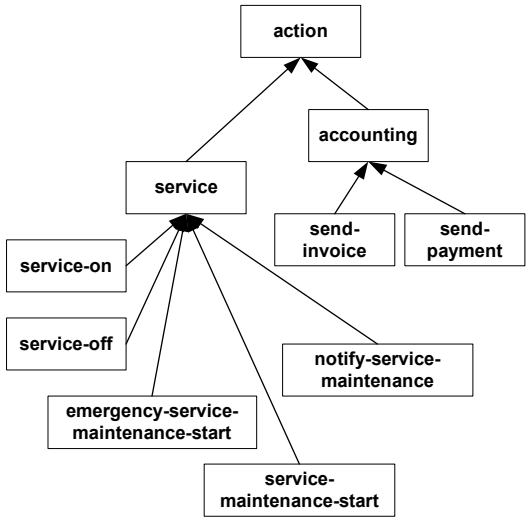

Figure 4 - An action type hierarchy

In addition to this, action types may also specify data that should be contained in the events that report them; for example, the accounting action in Figure 4 might specify an invoice number field that all subtypes will contain. We will use the above action hierarchy to specify the QoS contract.

Contract lifecycle – All contracts will follow the same basic lifecycle stages: specification, negotiation, active, expired. All contract instances must therefore identify the period for which they are *active* in order that an interpreter/checker may monitor them. All contracts must therefore contain two expressions indicating the start and end dates of the contract; these expressions can be evaluated dynamically, allowing a contract to specify non-static start or termination time.

The following syntax illustrates how start and end times are specified on our case study contract (note that a contract is a community, hence the community tag where we might have expected to see contract; for brevity, all subsequent code snippets are assumed to be declared within these tags):

```
<community name=”QOS Agreement”>
  <value name="contractStartTime" type="date"
    expr="?"/>

  <value name="contractEndTime" type="date"
    expr="?"/>

  <!– remaining contract details go here -->

</community>
```

The question marks must be substituted at the negotiation stage of the contracts lifecycle with either a date formatted string for static date values, or an expression which can used to evaluate the time point dynamically.

Contract roles **–** Our case study contract will need to be signed by a service provider and a client. Appropriate roles are therefore specified within the contract:

```
<community-role name="service-provider"/>
<community-role name="client”/>
```

The default cardinality for roles is one; so in this example, there must be one service provider and one client for this contract at all times.

Contract state **–** Contracts will need to maintain contract relevant state. This could be negotiated state that has been set prior to the contract becoming active, a constant value shared by all contract instances, or a value specific to a particular contract instance.

For our case study we might declare the following within our contract specification:

```
<constant name="paymentTimeLimit"
  type="time" value="28 days"/>

<variable name="numOverdueInvoices"
  type="int" init="0"/>
```

The constant definition introduces an immutable value called ‘paymentTimeLimit’ into the contracts scope which can be used to calculate whether a contract has been paid on time. We also declare a variable which will be used by contract instances to keep a track of the number of overdue invoices the client has with the service provider.

Accountability **–** To illustrate how accountability can be expressed we will examine the assignment of a burden. Our contract specifies that invoices must be paid within 28 days. This means that the client attracts a timed burden as a consequence of the invoice delivery action to pay the outstanding invoice.

We have already introduced the variable which keeps a count of outstanding invoices; we will now examine how burdens can be used to update this variable in reaction to payment violations. The code for the policy that monitors the invoice payments is verbose, so we use pseudo code to illustrate this instead:

```
upon receipt of an invoice action:
  record the invoice’s number and total amount
  apply a burden to the client to pay the invoice
  set a timer to violate this burden in 28 days
  loop:
    upon receipt of a payment for this invoice:
      record the amount of the payment
      if the invoice has been fully paid:
        discharge the burden
        if the burden was violated:
          decrement the overdue invoice counter
      else:
        repeat the loop

upon burden violation:
  increment the overdue invoice counter
```

The above algorithm uses a loop and a record of the received invoice’s number to allow a number of payments to be made for a single invoice. Note also in this example that we set a timer to violate the burden in 28 days. If the burden is discharged before this period, then the timer will have no effect.

Event filtering and time windows **–** The contract specifies that if there is more than 20 minutes of downtime in a week then the client is eligible for a refund. There are a number of influences that can affect this eligibility too, including: prompt payment of invoices by the client, fair notification of service interruptions for maintenance purposes, emergency maintenance, and interruptions due to events of Force Majeure.

In order to reduce the complexity of the events that the sliding window must deal with, we can add filters to the contract. Filters can receive the action notifications (system events) and listen for variable updates before producing higher-level events which delimit the start and end of chargeable downtime periods. For instance, if the number of overdue invoices variable has a value of one when a server goes down, then a ‘downtime’ event will not get forwarded to the sliding window until the invoice has been paid and the client is eligible, once more, to claim for the down time.

Filters for this contract will need to listen for changes to the unpaid invoices variable, as well as all types of service actions. Logic within the filters will determine when the accountable periods of downtime start and end, before creating new events to represent the relevant start and end points. It is these new events that allow the following time-window to calculate the total accountable downtime:

```
<policy name="monitor-service-events">
 <variable name="totalDownTime" type="time"/>
 <assign-burden id="maintain-service"
   community-role="service-provider">

   <sliding-window
      from="${contractStartTime}"
      until="${contractEndTime}"
      width="1 week"
      step="1 day">

    <evaluate>
      <for-each
       select="${window.events}" name="e">

        <!--examine all events and -->
        <!--calc the total downtime-->
      </for-each>

      <!-- if downtime is over 20 mins -->
      <violate-burden/>

    </evaluate>
   </sliding-window>
  <burden-violated>
   <email to="${clientEmailAddr}">
    WARNING: In the past week there has
    been ${totalDownTime} mins of downtime.
   </email>
  </burden-violated>
 </assign-burden>
</policy>
```

The above policy specification has been taken from a working example of this contract in our prototype interpreter and illustrates how the sliding window and burden operators can be used in conjunction.

Within the burden to maintain the service levels, a sliding window is used to periodically calculate the amount of downtime detected. The evaluate element of the window will be interpreted according to the 'step' attribute (in this case every 24 hours) and will explicitly violate the burden should the downtime limit be exceeded; this will result in an email, detailing the total downtime value, being sent to the address specified by the 'clientEmailAddr' variable.

ECL has been designed as a prototyping language. This means that operators are continually being added and removed in order that we may experiment with different ways of expressing our contracting scenarios. At the time of writing, operators for calculating summations of time bounds between events are under development. As a result of this, we have replaced the longhand evaluation of the total-downtime value with comments in order to aid readability.

6. Conclusions

This paper has presented the preliminary findings of an attempt to combine experience with two different contract monitoring systems. The result is an outline for an enhanced contract definition language, BCL, which the collaborators hope to complete in the near future and then to test with a variety of contract types.

7. Acknowledgements

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

This project was supported by the Innovation Access Programme-International Science and Technology, an initiative of the Government's Innovation Statement, Backing Australia's Ability.

8. References

[1] P. Linington, S. Neal, *Using Policies in the Checking of Business to Business Contracts*, Policy 2003 Workshop.

[2] Z. Milosevic, G. Dromey, *On Expressing and Monitoring Behaviour in Contracts*, EDOC2002 Conference, Lausanne, Switzerland

[3] R. Lee, A Logic Model for Electronic Contracting, *Decision Support Systems*, 4, 27-44.

[4] iMany, www.imany.com

[5] DiCarta, www.dicarta.com

[6] UpsideContracts, www.upsidecontract.com

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

[8] S. Neal, *A Language for the Dynamic Verification of Design Patterns in Distributed Computing*, PhD Thesis, University of Kent, 2001.

[9] Z. Milosevic. *Enterprise Aspects of Open Distributed Systems*. PhD thesis, Computer Science Dept. The University of Queensland, October 1995.

[10] S. Neal and P.F. Linington., "Tool Support for Development using Patterns", in Proc. 5th International Enterprise Distributed Object Computing Conference, Seattle, USA, September 2001.

[11] ISO\IEC IS 15414, Open Distributed Processing-Enterprise Language, 2002.

[12] Oracle Contracts, http://www.oracle.com/appsnet/products/contracts/content.html.

[13] Gamma *et al*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.