A unified behavioural model and a contract language for extended enterprise [1]

P.F. Linington [a], Z. Milosevic [b], J. Cole [b], S. Gibson [b], S. Kulkarni [b], S. Neal [a]

[a] *University of Kent, Canterbury, Kent CT2 7NF UK*

[b] *Distributed Systems Technology Centre, The University of Queensland, Brisbane, QLD 4072, Australia*

Abstract

This paper presents a coordination model for expressing behaviour in an extended enterprise. Our model is unified because it enables the same style of expressions for describing behaviour/structure in a self-contained enterprise and for describing cross-enterprise behaviour/structure. This model can support a broad range of modelling activities but the specific focus of this paper is on deriving the key elements of a domain language primarily targeted at expressing and monitoring behavioural conditions stated in business contracts. We also show how business contracts serve as a unifying mechanism for describing interactions in the extended enterprise.

Key words: E-Commerce, Contracts, Enterprise Specification, Cooperative work, Web Applications/XML

Email addresses: pfl@kent.ac.uk (P.F. Linington), zoran@dstc.edu.au (Z. Milosevic), colej@dstc.edu.au (J. Cole), sgibson@dstc.edu.au (S. Gibson), sachink@dstc.edu.au (S. Kulkarni), sn7@kent.ac.uk (S. Neal).

[1] The work reported 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 Industry, Science & Resources). Collaboration was supported by the Innovation Access Programme-International Science and Technology, an initiative of the Government's Innovation Statement, Backing Australia's Ability.

1 Introduction

Web Services capabilities raise the potential of distributed computing to a new level. As with previous generations of middleware technologies they will support integration of applications - in this case over the Internet. In addition, they will increasingly enable closer and more direct involvement of autonomous systems such as agents, organizations and people to be linked together into the extended enterprise — to achieve collaborative benefits through underlying business processes and other automated arrangements. These linkages are increasingly established on a dynamic and on-demand basis and may have a limited life cycle. In creating such linkages, however, these systems will also tend to preserve a certain level of autonomy and will be unlikely to give others unlimited access to their resources and services. In addition, they will require expressions of mutual obligations in the context of their collaboration and expression of corrective measures in cases where parties fail to perform their promised contributions to such a collaboration.

These broader sets of requirements, directly resulting from the need for more explicit recognition of a social and enterprise context for the applications running over Web Services imply a need for richer *coordination models* than the previous generation of middleware technologies. Thus, although the area of coordination models and languages has been a subject of active research over the last decade or so [1] [15] [10], the Internet context and the autonomous nature of agents call for an extended way of specifying behavioural constraints in collaborative arrangements. In addition to describing *basic* behaviour using, for example, sequential, parallel, or temporal constraints and non-deterministic choice, there is a need to describe *modal* constraints on behaviour such as permissions, prohibitions, obligations and the holding of rights and authorities. We refer to these broader sets of constraints as *enterprise policies*. They have been studied as part of disciplines such as deontic logic, normative systems and multi-agent systems, to name but a few.

In the Internet context, most of the notable work dealing with the problem of coordination has been carried out as part of various Web Service standardization efforts. The focus here is on the expression of basic behaviour constraints as mentioned above. For example, the BPEL initiative [2] is defining coordination concepts that can be applied to specify coordination of internal business processes on one hand and for the specification of business protocols between autonomous entities i.e. cross-organizational behaviour aspects, on the other hand. Examples of basic behaviour constraints for the internal business processes in BPEL are: sequence (for sequential execution of individual activities in the business process), flow (for concurrent execution of activities), switch (allowing the selection of one branch of activities from a set of choices), wait (for a given period of time, or until some deadline) and pick (allowing the

thread to block and wait for a message or for a time-out alarm to go off). Examples of basic behaviour constraints for cross-organizational interactions in BPEL are the concept of partner links, which describe conversational relationships between the services of two different partners and the concept of business partner, representing capabilities of a trading partner in terms of a subset of partner links.

A step towards a somewhat richer support for coordination in terms of the support for enterprise policy expression is the recent work on the Web Service Level Agreement (WSLA) language. A WSLA is an agreement between a service provider and a service requester using Web Services and it defines the obligations of both parties with respect to the service parameters (e.g. availability, response time and throughput). WSLA is specifically developed for expressing Web Service parameters, including their measurement aspects (both basic metrics and aggregate metrics) and the operations required for monitoring and managing service [4].

Service Level Agreements (SLAs) can be regarded as a special kind of contract concerned with the obligations regarding *service level* fulfilment and the WSLA specification is concerned with non-functional aspects of service. The specification of service parameters in WSLAs complements the behavioural specification expressed in the Web Service Definition Language (WSDL).

While WSLA focuses on technical aspects of services there is a need to address the business aspects of services, more specifically *business level* agreements, dealing with a higher layer in the Web Service stack. The business aspects of services need to deal with broader issues related to the enterprise policy concerns and the integration with other enterprise systems at the application and not the middleware layer.

This paper presents our solution for modelling coordination aspects of business level interactions over Web Services. The solution is based on our previously developed policy framework [12], influenced by the ODP community model [7] [9], and the results developed since then, as published in [13] [17] [19] [3]. The community model is aimed at describing cooperation between distributed and autonomous systems from a system-theoretic perspective. It provides a unified expression of basic behavioural constraints, which can be applied to model internal enterprise behaviour as well as cross-organizational interactions. It also provides a foundation to model the modal constraints and an underpinning model for describing these constraints is that of a contract. The definition of a contract is taken from the system-theoretic perspective, as "an agreement governing part of collective behaviour of a set of objects", and specifies obligations, permissions and prohibitions for the objects in the community.

This model can be used to express basic behaviour and policies within an en-

terprise, but it can equally be applied in the cross-enterprise context. While in the intra-enterprise context a contract sets out internal organizational policies, in the inter-enterprise context a contract reflects mutual agreement between organizations and it often has legally binding properties. Thus, the community model in the inter-enterprise context can address the concerns that are within the realm of legal or business contracts.

In cases where autonomous entities are filling roles in the community, and in particular in an inter-enterprise setting, it is possible that they may behave in a different way from that prescribed by their contract. Under these circumstances there is a need to implement monitoring mechanisms that will allow the detection of non-compliant enterprise behaviour. This is evidenced by the increasing demands from business for real-time monitoring of their activities internally and as part of the extended enterprise. Community models can also be exploited to support monitoring of activities in the extended enterprise; they serve as a specification of expected behaviour, which needs to be compared to the actual behaviour. In this paper we show the use of the community model in developing a contract language, which can be used to support event-based monitoring of activities prescribed by the business contract in the extended enterprise.

This paper is structured as follows. In the next section we provide a description of our underlying community model for enterprise object interaction and explain how it exploits the concepts of community, role, policies, modal control objects such as permits and burdens and their management, temporal constraints, and the composition and grouping of these concepts in relation to each other. The third section presents the use of the community model to describe interactions in the extended enterprise, as typically governed by a contract of some kind. The fourth section introduces the business contract language (BCL) which is based on the community model and which has been developed to support monitoring of the behaviour of trading partners and their services in the extended enterprise. The fifth section illustrates this by an example of a typical business contract such as a Service Level Agreement. The sixth section relates our work to similar efforts in the area. The paper concludes by indicating future directions.

2 Community model

One of the things that distinguishes the specification of business contracts from many traditional computer science problems is the richness of structure in business organizations and in the multi-party interactions found in such organizations. We can see this in two aspects of the structural descriptions: the rich set of entities in a business community and the rich set of participants

in the interactions they are involved in. In both these cases, there is a need to bring together a number of very different stake-holders and to identify recurring patterns in many different circumstances.

The ODP reference model [5] [6] [7] [8], on which many of our specification techniques are based, introduces the idea of a template for representing patterns of behaviour, and of roles in a template to provide a mechanism for its formal parameterization. A *role* is a placeholder in the template that is, in any particular pattern instance, filled by a specific object showing the behaviour expected from that aspect of the pattern. In communities, the main aim of introducing templates is to build larger structures, and so to express the correlation of actions performed by the different participants in the community. In actions, the focus is inward, looking at the way separate processes synchronise by playing a different sort of role, this time a role in the template for a shared action. These two perspectives are examined in turn.

2.1 Definition of community

The idea behind modelling an enterprise as a series of interrelated communities is to build up the constraints that govern what the enterprise must do from a series of separately defined pieces of behaviour, each with a well defined objective. Thus, in modelling terms, a *community* is a configuration of objects that interact to achieve some shared goal or objective. The community brings out the objective and the behaviour necessary to achieve it in a way that is independent of the details of the resources needed to achieve this behaviour. For example, the obligations involved in a commercial transaction in which goods are delivered and payment made can be described in terms of the roles of a general sales community that can be applied to transactions involving different goods, which take place between different kinds of organization.

The complete behaviour of the enterprise may eventually involve parts of the enterprise participating in a number of separately defined communities that have different objectives. The individual communities may describe, for example, the general framework for a trading process, application specific details of some parts of it (maybe as sub-communities) and possibly separate audit or security requirements. These separate communities may be nested or they may be linked by overlap through sharing one or more participants.

The key to the flexible combination and reuse of communities is to decouple the specification of the necessary behaviour from the statement of the objects involved by using a number of roles as formal parameters. The definition of a community type involves the declaration of a number of typed roles, and the community's behaviour is stated in terms of these community roles. When a

community instance is created, these community roles are filled by particular objects, whose types must match the type required, and these objects are thereby constrained to behave in a way consistent with the community's stated behaviour. As long as the observable behaviour of the community is consistent with its definition, the nature of the objects filling the various community roles can colour the actions of the community as a whole. The actual behaviour is selected from the alternative possibilities allowed for either explicitly or implicitly in the community design.

If the details of the way a community's members interact are hidden, the result is itself an object. This is called the community's equivalent object, and, being an object, it is able to fill suitable roles in other communities. In this way, hierarchies of communities can be created, representing sets of rules of progressively narrowing scope. This process can also be extended to progressively higher levels of abstraction, introducing, for example, overarching communities to represent, where appropriate, relevant constraints from aspects of the applicable legal system.

Thus communities can be combined in a number of ways. Two communities may be:

- related by having a role from each of the communities filled by a single object; this object is therefore subject to two sets of overlapping constraints, so that the two communities effectively constrain each other via it;
- related by having one community fill a role in the other; the inner community can be seem as an object that fills a role in the outer community;
- related by links in a specification, and so requiring a role in one community always to be filled by the object that fills a linked role in the other community.

In general, one object can fill several roles in a single community instance; this one object is then constrained to perform pieces of behaviour that could have been given separately to a number of distinct objects. This is generally a useful flexibility to allow the efficient and balanced use of resources, but the combination may sometimes not be so desirable. The classic example is the requirement for separation of duties often found in security frameworks. This leads to the need for community specifications to be able to express a set of role-filling constraints, stating, for example, that a particular pair of community roles, such as proposing and approving a purchase, must not be filled by the same object instance. The exact form of constraint allowed depends on the nature of the community notation used, but navigation-based constraints expressed in a notation like OCL could well be applied.

Although creation of many simple communities can be expressed by a single process of instantiation, in which all the roles are filled and the bindings thus

formed are seen as static thereafter, something more dynamic is often needed. However, the creation of a new community role and the filling of a community role by a new object are both actions that can be controlled by the community behaviour. Thus in a community describing the procedures of a committee, for example, the behaviour may identify that there is a set of committee member roles and a procedure for creating an additional role in this set and filling it with an object that had not previously been involved. (Note here again that a *one member one vote* requirement would also lead to a requirement that no single object should fill more than one of these roles.) If there is a need for complete flexibility, reflective techniques could be applied to describe how a community might change its own behaviour — having a process to change its own rules of procedure!

The community concept is generic enough to describe not only organizational structures such as contracts and companies but also lighter-weight structures such as the agents involved in the collection of various items related to an aspect of contract execution such as the handling of purchase orders or invoices. The coordinated group of agents is a simple form of local community.

2.2 Basic behaviour

So, what form does the specification of behaviour take in community models? *Basic behaviour* is expressed as constraints on the set of *actions* that can occur, and on the permissible sequences in which they can occur. Each action in the defined behaviour is associated with at least one role; however, isolated actions are not as interesting or expressive as interactions involving more than one role, since interactions immediately involve communication between the objects filling the roles. It should be obvious that the parties involved in an interaction are not interchangeable; there is generally an asymmetry, with different flows of information between the various pairings of objects involved. If we focus on the action type, independent of where in the overall behaviour it occurs, we can again apply the role concept, but this time to distinguish between the different responsibilities relative to the action. These are *action-roles*; they allow the definition of an action type to express the action's semantics in terms of its requirements and consequences for the participants. Again, this provides formal parameters for the definitions, which can be filled by the actual objects involved in an instance of the action type. For example, an interaction that results in the transfer of a simple data item might have action roles indicating that there needs to be both a producer and a consumer of the data item. A slightly more complex role pattern can be seen in the four-step send and receive pattern of a typical client-server interaction.

However, interactions within or between organizations may involve a wider

range of roles. Consider, for example, a *publish call for proposals* interaction in a tendering scenario. This may be seen as a single interaction in describing the steps in tender management, with action roles for the:

- author, who prepares the call;
- controller, who approves the correctness and authorizes its release;
- set of suppliers who should receive the call;
- archivist, who makes and maintains a non-repudiable copy for audit purposes.

Here the action is a quite abstract representation of a key step in the tendering process that the specifier has chosen to make atomic; local procedures could refine it into a series of finer-grain steps (making up the behaviour of a community undertaking this step), but a top-level view of the tendering process does not need this level of detail.

The definition of an *action template* captures the semantics of the intended action in terms of pre- and post-conditions on the objects filling the action roles. It identifies a period of time — the duration of the action — within which the state changes required of these objects must take place. It therefore provides a weak temporal synchronization constraint on the behaviour of these objects.

2.3 Modal constraints

A community constrains the behaviour of the objects involved; it qualifies the wide range of possible behaviours, indicating which of the courses of action available to the participants are acceptable. It will commonly state what action a party is obliged to take in a given situation. This is generally a weaker concept of obligation than defined in, for example, the standard deontic logic, which regards conflicting obligations as an inconsistency that prevents any deductions from being made. In the real world, conflicts between contractual requirements can occur, and a community model for contracts has to allow analysis of different behaviours, even in the presence of conflict. Thus there is a need to express violation of the required behaviour and additional obligations arising from it. We expect a community member to be able to attach a cost to any sequence of actions, and so to be able to make rational decisions about what course of action to pursue in the presence of conflict. To assist in doing this, we attempt to make the changes in obligations resulting from any action directly apparent.

2.3.1 Permissions, prohibitions, obligations

The modal constraints to be considered are permissions, prohibitions and obligations. The first two deal with possible behaviours, while obligations deal with expected actual behaviours. This difference is reflected in the comparative difficulty in checking that the requirements are satisfied.

A *prohibition* on an object engaging in an action role is satisfied so long as no action is observed to involve the object acting in that role. Any counter-example is sufficient to show a violation.

A *permission* for an object to perform an action role is corroborated when the action is performed without error, but this may take indefinitely long to observe if the object does not choose to exercise its rights. Observation of an attempt to perform the action failing in a way that indicates the permission has been withheld is clear evidence of violation. However, the action may fail for many reasons, and failure without an indication of why the failure took place is not clear evidence of lack of a permission. At what point, for example, does declaring a resource busy every time an attempt is made to use it amount to a failure to honour the permission to do so?

Finally, an *obligation* requires an object to engage in some piece of behaviour, involving participating in particular action roles, and is discharged if the required behaviour is observed. However, there may be some difficulty in identifying a violation because there are many different levels of urgency that can be associated with an obligation, and so in practice the situation is only straightforward if there is a defined deadline or temporal ordering requirement on the performance of one or more steps in the required behaviour.

2.3.2 Permits and burdens

One of the complexities in identifying the obligations to be met at the different stages in the execution of a community's behaviour is the need to track the changes associated with each action performed. For example, placing an order results in an obligation to make a payment at a later stage, while accepting an order leads to an obligation to deliver the goods or services.

A distinctive feature of the model proposed here is the reification of the modal constraints, so that transfer of responsibility can be expressed directly as the passing of a corresponding token. To avoid confusion with unencapsulated constraints, we introduce new terminology, calling a reified permission a *permit*, and a reified obligation a *burden*. The concept of a permit is quite similar to the operating system concept of a capability, except that here it is a specification concept, rather than an implementation construct. Using these concepts, we can express the action of placing an order as communicating not just the

order details but also a burden representing the obligation to deliver the goods ordered. Initiating the order would also establish a burden for payment locally on the initiator.

Once a burden has been created, it can be discharged by performing the piece of behaviour that it requires. The burden may then cease to exist (although, for periodic or continuing obligations it may continue, but with modified deadlines). Alternatively, the burden can, subject to there being suitable permissions, be passed on to another party, thereby transferring the responsibility it represents. Note therefore that it is not unreasonable for a participant to hold a burden that requires an action to be carried out when it does not have a permit to perform the action, as long as it does have a permit to transfer the burden and an expectation of finding a willing recipient to take it on. Alternatively, it may have an expectation of receiving the necessary permit before the burden becomes due.

As an example of this kind of behaviour, consider the placement of a purchase order where the supplier fulfils the order by using a shipping agent to deliver the goods. The supplier receives the order together with a burden to deliver the goods. As a matter of its internal organization, it decides to use a shipping agent who is responsible for organizing the services of and monitoring the performance of a suitable carrier. In the high level specification, there is one action in which the supplier receives the order with a burden to deliver within a certain time, which it discharges by another action of making the delivery. This behaviour is refined into a sub-community concerned with delivery, whose behaviour is made up of a series of steps, in which:

- the supplier transfers the delivery burden to the shipping agent, together with permits to access the goods and make the delivery;
- the shipping agent selects a carrier and transfers the delivery burden and associated permits to it, together with a burden to report on successful delivery. In doing so it also acquires a burden to take corrective action if it does not receive notification that delivery has taken place within the specified time. Note that because the selection of the carrier implies the acceptance of the burden, this action may fail.
- the carrier makes the delivery, and reports the fact, thereby discharging its burdens.
- receipt of the notification cancels the shipping agent's monitoring burden. The process is then complete.

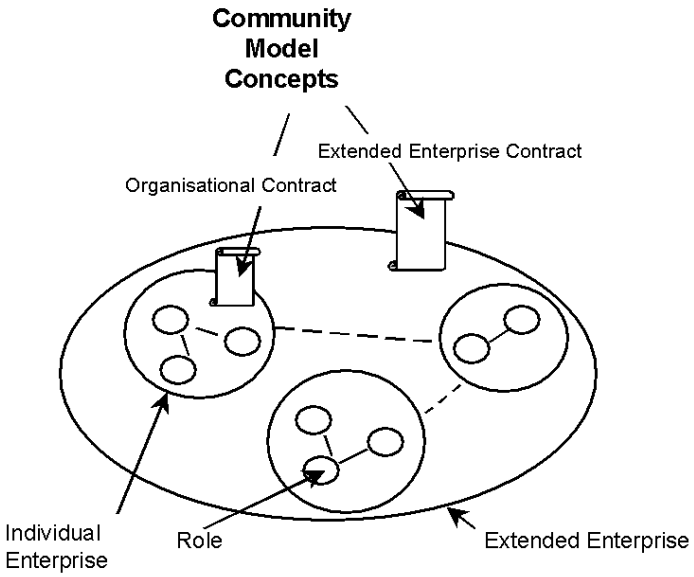

Fig. 1. Inter-organizational and Intra-organizational communities

2.4 Intra-enterprise and inter-enterprise applications of communities

It should be clear from the examples in the previous sub-section that communities are well suited to the hierarchical specification of constraints to match the different management domains involved (see figure 1). Thus the necessary constraints on the large scale that govern the cooperation between organizations engaged in commercial ventures can be expressed as communities in which the community roles are to be filled by objects representing the organizations concerned in their entirety. These are the minimum necessary constraints needed to represent the commercial agreements, and are likely to evolve rather slowly.

On a finer scale are the internal divisions of responsibility, representing the operational structuring into units within the separate organizations, be they divisions or subsidiaries. These are likely to evolve on a shorter time-scale, and can do so as long as they continue to be consistent with the inter-organizational behaviour required. This process can be used repeatedly to represent progressively finer-grained organizational units.

Although the largest-scale currently practical application of these modelling techniques is to describe relations between enterprises, they could also be used in the future to represent even larger scale social structures that influence commercial agreements. Future work might eventually lead to the ability to represent key aspects of legal structures, particularly those with a quite well defined scope and ontology, such as tax law or the core rules of property

conveyancing.

3 Expressing context for business contracts using communities

3.1 A model for contracts

A specific application of the community model introduced above can be to express the required behaviour of trading partners in a business contract — in terms of their basic behaviour constraints and policy constraints as part of their collaborative arrangement. This behaviour specification can then be used to support run-time checking of actual behaviour of trading partners against the required behaviour specified in a contract. We demonstrate how this community model is used to describe surrounding enterprise models of the contract management architecture, i.e. a Business Contract Architecture (BCA), as originally proposed in [16].

In applying the community concept to the expression of contracts, we use a community to represent each contract. The roles in the community represent placeholders in a contract proforma, and the objects filling community roles represent the signatories to the contract. Thus, in modelling terms, the concept of a contract is a specialization of a community, but other forms of community may still be used to represent, for example, the internal structures of organizations without these being formal contracts.

3.2 Use of the community model for contract monitoring

In order for contracts to be monitored automatically, they must be available in a standard form, such as is provided by the language described in this paper, and this form must be stored in a repository accessible to the contracting parties. A monitoring component can then access this definition, which consists of the constituents parts of the specification of communities, such as policies, roles, event patterns, temporal constraints and states (described in the next sections). The monitor can then collect events significant to the contract from the participants or the environment and interpret these in order to determine whether the contract is being followed. If problems are detected, these can be reported to an enforcement component that takes suitable corrective actions. The interactions of these components are shown in figure 2.

Often, incorrect behaviour can be detected immediately, but in more complex cases, particularly when a party is involved in several contracts at once, it may

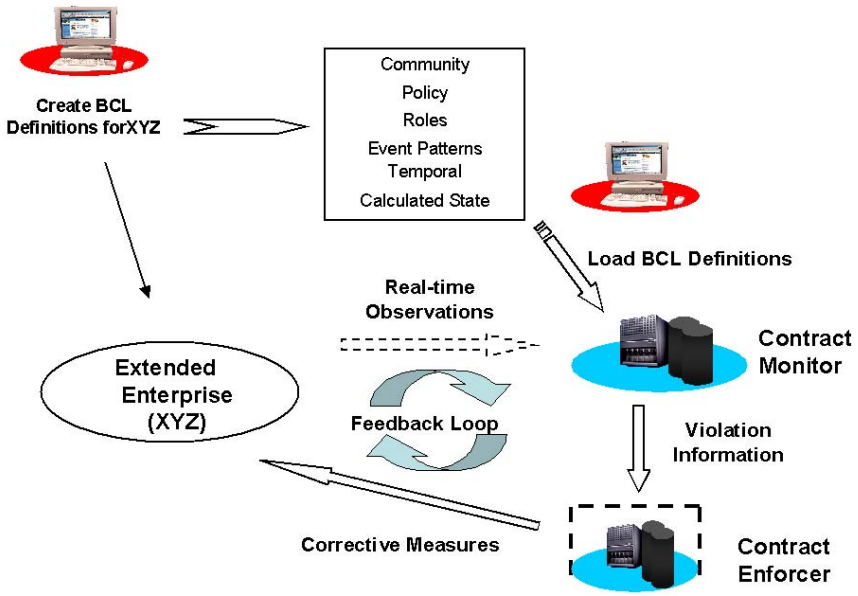

Fig. 2. The main components involved in contract monitoring

be necessary to observe a number of events before ambiguities of interpretation can be resolved. These problems of interpretation are discussed in [18] and [20].

Community behaviour is specified in terms of actions the participating objects can perform. However, for monitoring purposes, we are primarily concerned with observable events; these events may be the visible result of performing actions, or they may be other observations of the environment. The collection and distribution of events can take place in a number of ways, depending on the level of automation. Information about the events could come from programs within the enterprises, from people entering data within the system, from hardware, or from the communications infrastructure. In many cases the software or hardware will need to be instrumented in order to generate this information. Events about the creation of new contracts or the binding or rebinding of signatories to roles are of particular significance to a contract monitor. Events might be distributed to a monitoring component, or to other interested parties, by using a publish and subscribe style of event notification infrastructure.

3.3 Contract Monitoring as a control mechanism

The signatories to the contract act as sources of events that are passed to the contract monitoring system. Whilst the monitor may be just an observer, it may well itself generate events in turn. These can be distributed to other parts

of the system, or to the monitors of other contracts, possibly at different levels of abstraction, or may be returned to their originator for further processing.

The events a monitor generates will generally be progress signals, indicating that some stage of the contract has been completed, or that a violation has occurred. There are a number of design choices to be made about how tightly coupled the monitoring component is to the enterprise. At one extreme, it may be essentially passive, reporting problems to some management authority for corrective action as necessary. Alternatively, it may respond immediately to any violation, signalling exceptional events on so short a time-scale that incorrect events can be aborted. The time-scale that is feasible will depend to a large extent on the timeliness of the event reports made; inclusion of manual processes are likely to make tight coupling impractical.

Another issue, as soon as feedback paths are introduced, is to determine the obligations placed on the monitor. It is possible to imagine an independent and impartial monitor run by, for example, a regulatory authority, but it is also possible that the monitoring process is owned and operated by a signatory to the contract, and that it will give greater priority to the interests of that party. There might even be multiple monitors, each protecting the interests of one of the signatories, giving rise to issues of arbitration between monitoring components.

3.4 The limits of the monitoring system and executive override

The practical realities of contract management require some *system override* feature to handle situations outside those foreseen in the contract. In brief, the reality is that contracts are rarely devoid of ambiguity and rarely specify what should happen in every possible situation — there is always an element of interpretation of the contract conditions required. This leaves space for the signatories to come to some sort of agreement as to how a particular situation should be interpreted, and to override the system's coded behaviour with these details. Another reality is that even if the contract unambiguously specifies a condition, the parties may choose expediency and opt to ignore a violation of that condition. For example, the supplier may ignore a minor violation for late payment because it is not worth the time and effort to follow it up, or to avoid the reduction of good will that such a penalty may incur.

Once such an override feature is introduced, there are likely to be additional requirements on contract specification to indicate what the consequences of common forms of override might be, and there are clear similarities here with the requirement to express compensating actions found in many workflow systems.

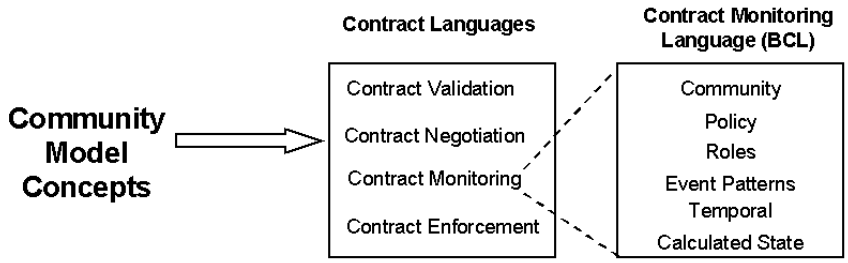

Fig. 3. The contract monitoring domain language

4 Business contract language (BCL)

The community-based approach can support a broad range of modelling activities, such as deriving key elements of various contract languages (see figure 3). In this paper we concentrate on a contract monitoring language, developed primarily for the purpose of expressing and monitoring behavioural conditions stated in business contracts. This language, called the Business Contract Language (BCL) is based on two previously separate sets of ideas. One of these comes from the work at DSTC over the last couple of years on further refining the original BCA design [16], including the development of a family of languages to express contract semantics (e.g. enterprise policy, event, state and notification languages [19]) and the application of XML and Web-based standards for BCL and BCA designs. The other set of ideas comes from separate work at the University of Kent on enterprise policy specification and on dynamic verification of design patterns in distributed systems [20], more recently extended to focus on policy-based checking of B2B contracts [13]. The two groups collaborated earlier on a framework for describing enterprise policies [12], which is based on the RM-ODP concepts of community and which has influenced later ODP standardization activities for ODP Enterprise Language [9].

The expressive power of BCL is in its capability to describe richness of structure and interactions in business organizations, including the enterprise policy concepts that are needed for the interpretation of behaviour governed by business contracts. Thus, BCL expresses true enterprise concepts, as opposed to other languages such as BPEL, WSLA and ebXML, that address computation aspects of behaviour.

BCL uses a number of key modelling elements to express business contracts. The process normally starts with the definition of a set of communities, which may be parameterized by the identification of policies. Details of the behaviour required are expressed using temporal and event-related constraints together with state conditions. Thus BCL is made up of:

- community expressions defining the domain for the definition of roles, role relationships, and policies, including delegation and accountability policies; these expressions also state relationships between communities (and thus contracts, e.g. sub-contracts and other dependencies between contracts);
- policies in contracts that express constraints on behaviour of the contract signatories, typically obligations, permissions and prohibitions; these constraints can take various forms and currently BCL provides support for expressing temporal, event relationship and state constraints, which can be superimposed on the basic behaviour expressions;
- temporal constraints that can take a simple form as in the expression of absolute and relative time points or a more complex form as in sliding time windows;
- event matching constraints and the relationships for describing compositions of such constraints, possibly further temporally constrained, are expressed in terms of event patterns; BCL currently supports the following relationship expressions: causal relationship, temporally ordered sequences and parallelism;
- state conditions, related to those contract variables whose values can change in response to conditions such as external triggers and temporal conditions (e.g. deadlines)

BCL exists in two syntactic forms. In the running system, a portable XML-based notation is used, and this is the form stored in repository and used for the encoding of messages describing events and for distributing dynamic changes to contracts; however, this form is somewhat cumbersome for development and, for design purposes, a human-readable language with more conventional concrete syntax is used. A tool has been implemented to parse this form and generate the XML representation automatically from it; additional visualization and model-checking facilities are planned for this tool.

The BCL language can be used in a number of ways at different stages in the contract's life-cycle. It can be used to define templates, to describe the state of a contract during execution, or to record elements in a historical trace. These differ in the strength of value bindings, since a template expresses the range of possible values but a trace reflects actual observation. The role any particular language utterance plays cannot be determined from the syntax, since this is uniform, but is determined by the context in which it occurs.

The power of the BCL language is that it allows the integration of complex monitoring conditions that include event patterns, temporal conditions and state conditions into a compact behavioural expression whose evaluation can ultimately determine whether a policy violation has occurred or not.

The definition of a community specifies a set of roles, policies, events, and states; it is hierarchical in form, so that any given community can be related to its parent and child communities. The community model expressions in BCL enable the creation and modification of community specifications as well as dynamically updating the status of a community, including assignments or re-assignments of roles and policies. The latter can occur during the lifetime of a community (and thus during business contract execution).

From the observational perspective a community model is involved in the checking of the validity of events that either change the specification of the community or signify some community operation (such as policy assignment or re-assignment during authorization or delegation operations).

BCL includes the concept of a community template and an automated means for instantiating communities from this template. Instantiation rules associated with a template define when a new community should be created from the template and what identifier should be used to identify this new community, e.g. it can be specified that the identifier obtained from an event that triggered creation of this community should be used. The instantiation rules also include the specification of *correlation information* which links this community instance and the separately created items related to this community, such as the events received by the community and the events generated by this community. For example, a shipper may be involved in many instances of the same shipping contract type.

The correlation concept is similar to the BPEL correlation sets concept — which is used to associate instances of business messages to the instances of business processes. This similarity will facilitate our planned integration of BCL and BCA with BPEL technologies.

4.2 Enterprise policy

A significant part of any business contract is made up of the statements that express deontic constraints of the form elaborated in the section 2.3 and thus policy expressions play a key role in BCL. A policy expression specifies the role to which a deontic modality applies, the deontic modality in question (typically permission, prohibition or obligation) and the behaviour to which this deontic modality refers. In essence it expresses fragments of behaviour that need to be monitored to determine compliance of the parties to the contract. These behaviour fragments range from the specification of primitive behaviour, i.e. events that directly reflect the actions of parties, to more complicated

behaviours that consist of complex event expressions. The latter include:

- event patterns that express some combination of actions of significance to the contract (e.g. a Purchase Order (PO) received, should be followed by a PO acknowledgement);
- the expression of state conditions of relevance to the contract (e.g. the total amount of payment in one month); and
- a range of temporal conditions (e.g. within 5 days from the receipt of the purchase order).

The BCL semantics specify that the evaluation of a policy is signified where appropriate by an event which contains the evaluation result. This allows the user to specify how contract violations are to be handled, for example, simply by sending notifications to various interested parties or by applying other policies to internally processed violation events. An example of such a policy would be to prohibit the supplier from proceeding with further actions until their violation has been remedied.

4.3 Event-patterns

BCL is an event driven language. A BCL event signifies some important occurrence with implications for contract condition fulfilment and this is the central mechanism to support real-time contract monitoring. A BCL event falls into one of two basic categories, either atomic or complex. A complex event is described using the concept of an event pattern. The BCL event pattern language is similar to the language described in [14] and supports the expression of event matching constraints and relationships. The concept of a complex event also allows us to group events and treat the group as an event at another level of abstraction.

The monitoring engine will subscribe to the relevant event types so that when an event of that type occurs the engine will begin the process of contract condition checking, performing, for example, the detection of event patterns or the evaluation of policy constraints.

4.4 Calculated States

The monitoring of business activities governed by a contract often requires checking of certain state information associated with the contract and calculating its new value in response to the occurrence of some events. BCL provides two options for calculating values of state variables. One way is calculating the value on request. For example, if one needs to calculate the amount of a

fine based on the time interval since some deadline has passed, it may be more efficient to perform lazy evaluation, calculating this value only when needed, rather than via periodic updates. Another option is to calculate the new state and update it in response to an occurrence of a particular event. Whenever this event happens, the appropriate calculation is carried out.

Calculated states are self-contained in the sense that they encapsulate the expression for determining their current value and manage their own updating. This allows a clean separation of the code handling state information from the rest of the system.

4.5 Temporal aspects

Temporal constraints are essential to many contracts. They are used to express conditions related to the contract as a whole, e.g. start and end dates of the contract or conditions that relate to deontic modalities. Examples of the latter are deadlines for discharging obligations (e.g. payment should be made one week after receipt of an invoice) or intervals permitting certain behaviour (e.g. the times of day to which different bandwidth prices apply). These temporal elements can be specified in a number of different ways. For example, a contract may specify recurring intervals, e.g. "goods worth $500 must be purchased each month", or specify intervals that are combined with other intervals, as in "3PM to 6PM during Weekdays".

BCL provides various options for expressing temporal conditions ranging from basic expressions such as points in time, durations and deadlines to complex expressions, such as the sliding time windows.

The most basic specification of time is as an absolute time point; this may be a literal value, such as "12:00PM, 31 January 2003". Such a time point may be used to specify the contract start and end dates, or a deadline for obligations, such as "the goods must be delivered before the 1st of March". Time points may also be specified relative to some other time point, as in "the deadline for payment is seven days after the goods have been received".

Some contracts contain expressions of a more complex temporal form. Take for example a SLA contract specifying that: "there must not be more than three occurrences of downtime within any one-week period". There are many contract conditions that have this basic form, and we refer to this temporal pattern as a sliding time-window. From the monitoring perspective, the problem is to detect the occurrence of some condition in the course of a time window that 'slides' over the time line. The time window is characterized by the window's width, the specific condition that needs to be checked within the window, the expressions stating what to do when a condition is found or is

not found, and if, appropriate, how to move the window forward.

5 Application to a Service Level Agreement

This section illustrates the use of BCL by considering a simplified example of a special kind of business contract, namely a Service Level Agreement (SLA), such as the SLA discussed in [19]. Although this specific example is based on an industrial SLA we focus only on those aspects that illustrate the expressive power of BCL and leave out other, more trivial, details. To make the examples easier to follow, a simple pseudo-code notation derived from the human-readable notation for BCL is used in the examples; the real implementation stores a representation of the contracts in its repository in the less readable XML-based syntax of BCL.

In the example, the user enters into a framework contract with the service supplier, under which more specific sub-contracts are agreed to cover the use of individual servers. The individual servers provide web servers that must comply with uptime guarantees, and the client purchases space on these servers. The contracts will conform to the following criteria:

- The framework contract will be for a fixed period of twelve months from an agreed start date.
- The maximum permitted downtime for any server will be twenty minutes in any rolling seven-day period, starting at midnight.
- Downtime is defined by there not being HTTP access to the server.
- In calculating downtime, the contract will exclude any times where:
  - 48 hours notice of maintenance has been given to the client,
  - emergency maintenance is required,
  - the client has not paid outstanding invoices by the agreed payment deadline, or
  - the service has been made unavailable by Force Majeure.
- If the agreed downtime limit has been exceeded, the service provider will, upon request from the client, credit the client's account with a pro-rated charge for one week's service.
- Invoices are issued at the end of each month; all invoices are to be paid within 28 days of issue.
- The client is allowed to purchase additional space on other servers, there by creating additional sub-contracts of framework agreement.

The description of the implementation of this SLA given below follows the structure of section 4.

The executable version of the contract is modelled as a community. The community will define the client and supplier roles, basic behaviour required of the roles and the policies applied to those roles, including the maximum permitted downtime limit placed upon the supplier, and the obligation for the client to pay for the service within 28 days of the service starting. The community also contains definitions of the states and events required by these policies, such as events representing downtime or the deadline for payment.

The community will declare variables (which we call *value containers*) used by any of the polices, states, events and other definitions associated with the community. For example, the start date of the contract is a value container whose content is set to a specific date when the contract is created, and the number of days for invoice payment is a value container that will be referred to in the event definition representing the payment deadline. The community definition may also contain the definitions of:

- notification types, stating requirements for the generation of messages to interested parties as the contract progresses;
- behaviour for updating the structure of the community, such as assigning a new entity to a role; or
- user defined methods.

Rather than defining a community for each individual contract instance, we write a community template, which represents a proforma contract. This contains an instantiation rule that states when new instances should be created from the template.

```
Community: SLA_Framework
   InstantiationRule:
      TriggerEvent: ContractAgreed where
         ContractAgreed.TemplateIdElement = SLA_Framework
      InstanceIdentifier: ContractIdElement
      CorrelationIdentifierLocation: ContractIdElement
   Community: Server
      ...
```

In general, a new instance of the community is created from the template whenever a defined trigger event is received. The instance identifier for this community will be obtained from the trigger event's content; in the example, the trigger event is a ContractAgreed event with a TemplateIdElement indicating the framework contract template. The instance identifier is taken from the ContractIdElement, which is also a field in the ContractAgreed event. The instantiation rule also specifies the location for correlation information

which is used to correlate the events generated by this community or received by it with the correct community model instance. Here correlation is simply performed by matching the ContractIdElement field in any events observed.

The community template for the contract contains the definitions for value containers to be filled during contract instantiation. The necessary information is transferred from the event that created the instance in a way governed by the declaration of the correlation mechanism.

The framework contract contains the community template for handling the server sub-contract of the contract, which includes policies for handling downtime limits. In a particular server contract instance, the sub-community's template instantiation rule will cause a new instance of the sub-community to be created every time a server-purchase event occurs.

The server sub-community contains policies and some corresponding event definitions which are used for downtime monitoring for the server. As there will be multiple servers, the events related to a server's downtime will have to be correlated with the appropriate sub-community instance. Similarly, policy violation events generated within a sub-community instance will have to be appropriately identified with that instance, and through this, with the appropriate server.

The BCL correlation mechanism again ensures that the events received and generated are correlated with the appropriate community and sub-community instances. This means that if a server downtime event is generated, that event is sent to any sub-community which is concerned with the server responsible for the event. To achieve this, we use the following definition for the sub-community instantiation rule:

```
Community: Server
   InstantiationRule:
      TriggerEvent: NewServerSpacePurchase
      InstanceIdentifier: ServerIdElement
      CorrelationIdentifierLocation: ServerIdElement
```

5.2 Policies

Consider the availability policy: "The maximum permitted downtime for the server will be twenty minutes in any rolling seven-day period". Assume that there is a complex event that is generated when there have been twenty minutes of downtime within a week. This policy can be defined as:

```
Policy: MaximumDowntime
   Role: Supplier
   Modality: Prohibition
   Condition: OverTwentyMinutesDowntime
```

Within a community, it may be possible to specify policies expressing default conditions such as "all events that are not explicitly permitted are prohibited". We are currently considering the inclusion of such features within BCL. If this community contained such default policies, it would be possible to specify this policy as a permission for the downtime to reach twenty minutes in any week.

5.3 Event Patterns

The contract specifies that "Downtime is defined by there not being HTTP access to the server". We can assume that there is some program that is monitoring availability of HTTP access, which might try accessing the server at regular intervals. The details of such a program are outside of the scope of this paper, but we can assume that it generates an AccessDown event when it detects access is down, and an AccessUp event when it detects that it has come back up.

We can specify a complex event, a Downtime event, saying that the server was down between the AccessDown and AccessUp events. We will often need to perform such a "mapping" from the actual events being generated within the enterprise into the higher-level events that the contract conditions deal with.

```
EventType: Downtime
   GenerateOn:
      AccessDown; AccessUp
      // where ';' is the sequential composition operator
```

Notice that after the AccessDown event occurs, the Downtime event becomes instantiated but is not completed (and thus not yet generated) until the AccessUp event occurs. It is possible to query the current duration of such a complex event that has started but not yet completed. This can be important, for example, if a sliding window (see below) moves while a complex event is in progress, since movement of the window itself generates an internal event which is likely to trigger contractual policies on excessive downtime if the current Downtime event contributes sufficiently to the cumulative total. Alternatively, we could use periodic events to trigger a check on some condition, as in the OverTwentyMinutesDowntime event type, which checks the duration of its corresponding ContractualDowntime event every minute, to see whether it has yet exceeded twenty minutes.

While downtime can be determined in this matter, the contract adds some additional conditions over what should be considered as effective downtime for which the supplier is solely responsible and which should thus be used to reflects their non-performance. Effectively, the complex event *Contractual-Downtime* excludes any periods of downtime where any of the following holds:

- 48 hours notice of maintenance 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 Force Majeure.

Scheduled maintenance can be defined as a complex event corresponding to a ScheduledMaintenanceStart event followed by a ScheduledMaintenanceEnd event, where a MaintenanceNotification event occurred at least 48 hours before the ScheduledMaintenanceStart event. Emergency maintenance can be defined as a complex event corresponding to an EmergencyMaintenanceStart event followed by an EmergencyMaintenanceEnd event. The events defining the starts and ends of scheduled and emergency maintenance would have to be supplied by the enterprises being monitored.

We can represent the period between the deadline for payment of an invoice and the payment of that invoice by an UnpaidPayment complex event defined as a PaymentLate event followed by a Payment event. These two events are quite simple and are not shown here.

We can then define the ContractualDowntime event type to be any period of downtime that does not overlap with a ScheduledMaintenance event, an EmergencyMaintenance event or an UnpaidPayment event.

```
EventType: ContractualDowntime
   GenerateOn:
      Downtime intersection not ( EmergencyMaintenance or
                                  ScheduledMaintenance or
                                  UnpaidPayment )
```

Here *intersection* takes two or more event patterns as arguments, and generates complex events representing the times where matches of the argument event patterns are both occurring at the same time.

This does not include, however, handling of Force Majeure. Usually, the system will not be able to know *at the time* that the downtime was caused by Force Majeure. Thus, to handle Force Majeure the system needs to allow us to specify retrospectively that some period of downtime was caused by Force Majeure and thus should not be considered as contractual downtime. This, in itself, is not difficult, but the difficulty comes from the consequences of having already considered that time as being contractual downtime; for example,

restitution payment may already have been made. This is a complex area, because reassessment of the event history after identification of Force Majeure might well lead to the conclusion that the client had underpaid, triggering the outstanding invoices exception clause in the downtime specification and leading to an unreasonable cascade of changes to the downtime history. Thus the change of downtime status must be retained in the event history, indicating why the actions were reasonably taken so as to block unbounded reassessment.

The details of how reassessments might be handled are still being studied. We believe it is possible to automate some of the handling of such cases. One way of handling this situation is to allow people to modify the state of the system, so that they can go back and specify that Force Majeure was in effect at a certain time, and that a violation caused by this should not be counted as a violation, and thus the penalty for it no longer applies. BCL allows a *system override* facility (see 3.4), which lets any suitably authorized user override any decisions made by the system, and notes the fact in the trace of events that has occurred and modifies the values of any value-containers or states held by the system.

5.4 Temporal Aspects

The contract states that it will "... be for a fixed period of twelve months from an agreed start date". We define the start and end dates of a contract using the concept of a value container — a named piece of data that can be referenced by other parts of the definition of the contract:

```
Value: StartDate
   1 February 2003, 12:00:00 EST

Value: EndDate
   StartDate + 12 Months
```

In other circumstances, we could specify the end date as a literal time point or as occurring when a certain condition has been met (such as when all the goods have been paid for).

BCL does not mandate any particular way of handling of start and end dates, giving the users flexibility to manage them as necessary. The system could, for example, send notifications to the parties on these dates. Further, it is possible to specify that all input events that occur before or after these dates are ignored, or to send a notification if any such events occur. Standard ways of handling start and end dates could be included in libraries of standard definitions.

The contract specifies that the payment must be made within 28 days of the

service commencement. Having a deadline for the fulfilment of an obligation is a very common pattern in contracts. This can be expressed in BCL using the *before* operator, which allows us to check that the payment came before the deadline.

```
Policy: PaymentChecks
   Role: Client
   Modality: Obligation
   Condition:
      PaymentMade before ( ServiceCommencement + 28 Days )
```

This also illustrates the polymorphic nature of events. The two arguments to the *before* operator are time points, but placing an event in a context where a time point is expected yields the time when the event occurs. Events can be used in arithmetic expressions to yield a time point relative to an earlier event pattern.

Complex events extend over some period of time, as illustrated by the ContractualDowntime event, which represents periods of time that does not overlap with maintenance or emergency downtime, or when the client has an overdue bill. There is the durationOf method for finding the duration of a complex event (or its duration so far, if the occurrence it is representing has not completed yet).

6 Related work

The capabilities of the XML and Web Service standards to support cross-organizational, real-time and on-demand business collaborations have renewed interest in studying the semantics of contracts and their architectural implications for e-commerce systems, as initially discussed in [16].

The early work on electronic representation of contracts by Lee [11] was concerned with developing a logic model for contracting by considering their temporal, deontic and performative aspects. In many respects, our BCL approach adopts a similar philosophy, but is developed from a different angle — the enterprise modelling considerations related to open distributed systems. Our approach, based on the ODP community concept [7] [9] and inspired by deontic formalisms, gives prominence to the problem of defining enterprise policies as part of organizational structures. Further, we treat contracts as a group of related policies that regulate inter-organizational business activities and processes [12] [13]. In this respect we take a similar approach to that of van den Heuvel and Weigand [22], who developed a business contract specification language to link specifications of workflow systems.

In fact, we consider contracts as the principle coordination mechanism for the extended enterprise and, considering possible non-compliance situations, we provide architectural solutions to the problem of monitoring the behaviour stipulated by a contract as initially proposed in [16]. In addition, this monitoring makes use of sophisticated event processing machinery similar to that of Rapide language [14], which was initially described in [19] and is further elaborated in this paper.

The WSLA work is concerned with service level agreements for Web Service standards (essentially addressing end-point aspects of Web Services), but our approach, while related, is addressing a higher layer in the Web Services stack — the business level agreements.

We anticipate that current developments in the BPEL and Web Services Choreography Language (as part of W3C standardization) will provide powerful facilities for expressing cross-organizational business processes and we plan to investigate options for positioning our solutions as part of these efforts. We note that current BPEL specification and our BCL work have adopted a similar event-driven and declarative rule-based design philosophy with full exploitation of underlying XML and Web Services standards. Both approaches express behaviour patterns — our specification considers more general expression of behaviour, while BPEL focuses on the business process style of expressions.

Finally, some new e-business standardization efforts have started work on developing standards for expressing the semantics of business contracts and agreements. The OASIS standards body has recently formed a Technical Committee to develop an XML e-contract standard, aimed at developing "open XML standards for the markup of contract documents to enable the efficient creation, maintenance, management, exchange and publication of contract documents and contract terms" [21]. UN/CEFACT has also started work on extending EDIFACT and ebXML standards to support agreements and contracts, within their Unified Business Agreements and Contract project. Our work can be regarded as a contribution to these standardization efforts.

7 Conclusions and Future work

In this paper we have presented a model for describing coordination in the extended enterprise. This generic model, based on the ODP community concept is unified because it enables the same style of expressions for internal enterprise behaviour/structure and for cross-enterprise behaviour/structure, i.e. the extended enterprise. In the latter case, the community representing the extended enterprise can itself be considered as an autonomous entity.

The coordination model includes basic behaviour concepts such as individual actions and compositions of such actions using, for example, sequential, parallel or non-deterministic choice composition operators while taking into account other constraints such as state and temporal relationships. The model also allows the description of modal behaviour constraints such as permissions, prohibitions, obligations and holding of the rights and authorities. An important part of this model is an unambiguous expression of mechanisms for managing permissions and obligations within the extended enterprise. The model adopts a novel approach for passing obligations and permissions between objects as recently proposed in [13]. It treats the permissions and obligations themselves as objects that are created and destroyed as they are assigned and discarded. These token objects are called permits and burdens respectively.

The community modelling elements can be applied to express coordination activities for the extended enterprise as defined by a business contract. This paper shows the expression of these coordination activities in terms of constraints on the behaviour of parties as specified in business contracts, in particular the monitoring of deontic constraints such as permissions, obligations and prohibitions. In future work we plan to explore other roles for contract languages; this may involve other aspects of business process modelling, and offers opportunities for closer integration with BPEL4WS, for example, to bring in other ways of expressing the coordination activities associated with business contracts.

The community modelling concepts can also be used to derive concepts of the extended business contracts languages needed to support e-contracts. This paper shows how a community model was used as a basis for deriving concepts of one such language - aimed at supporting contract monitoring activities. In future work we will investigate the use of the community model for deriving other contract languages, such as for contract negotiation, contract validation and contract enforcement.

References

[1] F. Arbab, The IWIM model for coordination of concurrent activities. In Coordination Languages and Models, LNCS 1061, Springer, 1996

[2] Business Process Execution Language for Web Services, 1.1, May 2003, http://www-106.ibm.com/developerworks/library/ws-bpel/

[3] J. Cole, J. Derrick, Z. Milosevic and K. Raymond, Author Obliged to Submit Paper before 4 July: Policies in an Enterprise Specification, Proc. Policy Workshop, Bristol, UK, January 2001

[4] IBM, Web Service Level Agreements (WSLA) Project: SLA Compliance Monitoring for e-Business on demand, http://www.research.ibm.com/wsla/

[5] ISO/IEC IS 10746-1, Open Distributed Processing Reference Model - Part 1: Overview, ISO 1995

[6] ISO/IEC IS 10746-2, Open Distributed Processing Reference Model - Part 2: Foundations, ISO 1995

[7] ISO/IEC IS 10746-3, Open Distributed Processing Reference Model - Part 3: Architecture, ISO 1995

[8] ISO/IEC IS 10746-4, Open Distributed Processing Reference Model - Part 4: Architectural Semantics, ISO 1995

[9] ISO/IEC IS 15414, Open Distributed Processing - Enterprise Language, ISO 2002

[10] A. A. Holzbacher, A software environment for concurrent coordinated programming. In Coordination Languages and Models, LNCS 1061, Springer, 1996

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

[12] P. F. Linington, Z. Milosevic and K. Raymond, Policies in Communities: Extending the ODP Enterprise Viewpoint, Proc. 2nd International Workshop on Enterprise Distributed Object Computing (EDOC'98), San Diego, USA, November, 1998

[13] P. F. Linington and S. Neal, Using Policies in the Checking of Business to Business Contracts, Policy Workshop, Como, Italy, June 2003

[14] D. Luckham, The Power of Events, Addison Wesley, 2002

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, Specifying distributed software architectures. In Proceedings of the 5th European Software Engineering Conference, Sept. 1995

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

[17] Z. Milosevic and G. Dromey, On Expressing and Monitoring Behaviour in Contracts, Proc. 6th International Conference on Enterprise Distributed Object Computing (EDOC'02), Lausanne, Switzerland, September 2002

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

[19] S. Neal, J. Cole, P. F. Linington, Z. Milosevic, S. Gibson and S. Kulkarni, Identifying requirements for Business Contract Language: a Monitoring Perspective, Proc. 7th International Conference on Enterprise Distributed Object Computing (EDOC'03), Brisbane, Australia, September 2003

[20] S.Neal and P.F.Linington, Tool support for development using patterns, Proc. 5th International Conference on Enterprise Distributed Object Computing (EDOC'01), Seattle, USA, September 2001

[21] OASIS LegalXML eContracts Technical Committee Charter, http://www.oasis-open.org/committees/legalxml-econtracts/charter.php

[22] W-Jan van den Heuvel and H. Weigand, Cross-Organisational Workflow Integration using Contracts, Decision Support Systems, 33(3): p. 247-265