Using Policies in the Checking of Business to Business Contracts

Peter F. Linington and Stephen Neal
University of Kent, Canterbury, Kent, CT2 7NF, UK
pfl@ukc.ac.uk, sn7@ukc.ac.uk

Abstract

The mechanization of business-to-business contract enforcement requires a clear architecture and a clear and unambiguous underpinning model of the way permissions and obligations are managed within organizations. Policies will need to be expressed in terms of the basic model, and the expressive power available will depend, in part, on the ability to compose sets of policies derived from different sources. The models used must reflect the structure of the organizations concerned and how the behaviour of organizations is constrained by broader shared rules. This paper considers a contract monitoring system intended to provide automated checking of business to business contracts, sets out a suitable model and explains how it can be used to guide the representation and control of contracts in a prototype monitoring system.

1. Checking business to business contracts

As an increasing amount of routine commercial activity becomes automated, the importance of techniques for checking the correctness of interactions and flagging incorrect behaviour increases. Many situations involving the supply of goods and services are carried out on the basis of periodic demands regulated by a previously established proforma agreement – a contract. Such a contract will specify the parties involved and the constraints on the behaviour of each of them. The steps described are likely to be a mixture of real world events, such as delivery of goods, and synchronous or asynchronous interactions between systems in the IT infrastructure. This may be via web services, workflow systems or various other forms of middleware. Some of these steps may be directly observable with suitable tools, and some may need to be inferred indirectly from subsequent reports.

The unpredictability of human activities and the need to satisfy conflicting requirements makes this problem much more complex than previous requirements to check the correctness of distributed behaviour, such as protocol conformance testing. In this environment more flexibility of interpretation is needed and a certain degree of backtracking and reassessment is very likely to be required.

The work described here builds upon previous work at the University of Kent on the automated checking of the correct application of design patterns [1][2], which had many similar features. It also draws upon the work within the International Organization for Standardization on Open Distributed Processing (ODP) [3][4] and particularly on the definition of the Enterprise Language for ODP [5] to give a framework for modelling contracts and organizational structures. In this area it takes a similar approach to the earlier work on Business Contract Architectures (BCA) [6][7].

This paper starts with a review of the problems associated with electronic business contracts (in section 2) and then gives an overview of the architecture adopted (in section 3). It then reviews the modelling approach being taken to express organizational structures in terms of communities (in section 4) and how this relates to the checking of the correct composition and application of policies (in section 5). Finally, it describes how the checking mechanism is being implemented (in section 6), examines the likely impact of such systems on the business process (in section 7) and draws conclusions for future work (in section 8).

2. Assumptions about contracts

2.1. The Form of Contracts

Contracts are rarely self-contained. A contract is an incremental piece of specification that depends on already established social and legal norms, and draws on the organizational structure and place in it of the parties to the contract. Some of the issues of rule composition and behavioural inheritance that arise from this positioning within an overlapping set of environments were explored in [8]. The most important feature for our current purposes is the imposition of rules from the environment to constrain or modify features in the contract. This process needs to be reflected in the contract interpretation and checking mechanisms if they are to give effective guidance on correctness to the business process.

The ODP reference model defines a contract in terms of a set of permissions, prohibitions and obligations. From the conformance point of view, prohibition is the easiest of these to interpret. If something is prohibited, a single contrary observation is sufficient to demonstrate non-conformance, and the absence of a prohibited event is unremarkable.

Permissions are slightly more complex to interpret. This is because, although absence of a permitted event is still unremarkable, and occurrence of a permitted event confirms that the permission is being satisfied, the failure of an event is problematical. Although the failure can be observed, demonstration that it is a failure to honour the permission depends on the analysis of the cause of the failure. This is not hard if a failed event is labelled as being caused by a lack of permission, but is much more difficult if failure could arise from a number of causes. 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?

Obligations are even more difficult, because of the need to assess the urgency with which the obligation is to be discharged (and the related issues of continuing and recurrent obligations). Thus the failure to fulfil an obligation *as soon as is reasonably possible* is extremely difficult to assess from observation. The fact that the required action has not yet been observed is evidence for a failure to meet the obligation only in simple cases where the contract specifies an obligation for action before an explicitly stated deadline. In general, failure to observe an action does not imply failure to attempt it, and many kinds of extenuating circumstances are possible.

If these problems are to be overcome, it will be necessary to base the implementation on a solid theory of contract interpretation, particularly with respect to hierarchies of permissions and obligations, and to represent this by having a clear reference model for the steps in performing actions subject to contract. The approach taken is to express the semantics of a contract in terms of the behaviour of a core policy interpreter.

The aim of the system described here is to test the correctness and expressive power of our contract model by experiment with a pilot implementation of the contract checking function, and by making trials with a set of example contracts.

2.2. Contract specification

The starting point for contract specification is the declaration of the context in which the contract is to be applied; doing this establishes the constraints under which the contract is to operate. The contract then needs to be both consistent with these constraints and internally self-consistent. Checking consistency can be expensive in complex cases, but the costs can be reduced considerably by using suitable conservative approximations. See [18] for an approach to conflict resolution using logic programming.

One would expect the contract model to support:

- declaration of pre-existing constraints from the environment;
- a range of kinds of composition operations on contracts, allowing their incremental combination and supporting reuse of existing fragments;
- the delegation of responsibility (although this may be expressed by using a suitable compositional structure of communities in the description of the organizational parties – see below);
- the description of how permissions and obligations are incorporated into the contract.

One of the distinctive features of our model is in the reification of both permissions and obligation, using a capability-like mechanism (see, for example, [9][10][11] for background to capabilities in the world of operating systems). This approach unifies the treatment of distribution of permissions and obligations, particularly permissions or obligations to make or change policies or to override pre-existing behaviour. However, a straightforward capability model is not sufficient; we will need to combine it with a constrained role structure so as to give some dual control mechanism explicitly to support separation of duties. This can be expressed by use of a combination of roles in action templates and roles in communities (see below).

Other policy languages, such as Ponder [17], use unreified systems of permissions and prohibitions to perform similar modelling. They distinguish between permitted and prohibited behaviours, but rely on the definition of defaults when neither a prohibition nor a permission is specified. We solve the same problem by deriving constraints from a series of overlapping communities, so that there is no need for a concept of default, merely visibility of unmodified rules from an overarching community in cases where no specific constraint is being applied. This is in some ways similar to the way nested contexts are built up in natural language, an approach that the Opera group in Cambridge are investigating [15].

3. The control Architecture

The objective in this work is to test the completeness of our contract model by the construction of a free-standing contract checking component and the demonstration that it can identify a significant proportion of the contract violations that occur. This, and a wish to evolve towards a Model-Driven Architecture approach, leads us towards a repository-based architecture. Although the starting point is different, this has led us to similar conclusions to the authors of the Business Contract Architecture [6].

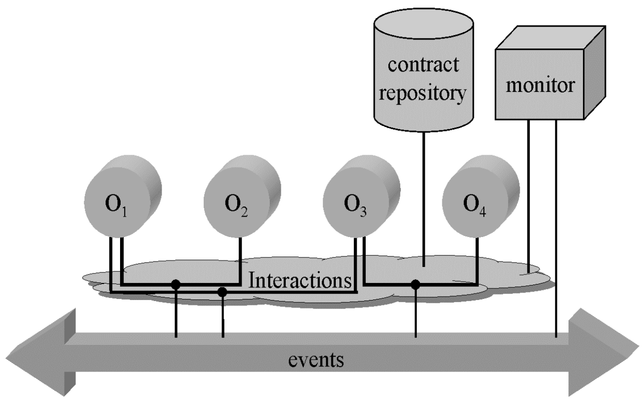

Figure 1. The main architectural components.

The main components of an application environment for the current purposes are thus considered to be:

- a set of enterprise (application) objects;
- a federated contract repository, so that the contract or contracts applying in any particular situation can be accessed;
- a monitor that uses the information from the repository to check events and generate summaries of correctness and exception reporting events;
- an event distribution infrastructure, capable of distinguishing between event types, based on a publish and subscribe model;
- a pervasive infrastructure capable of supporting abstract binding to form communities and the associated filling of roles by objects. This infrastructure will also originate events to describe changes in bindings, although such information may not always be available;
- some mechanisms for trading or broking available services to support the dynamic configuration of the environment and the set of participants covered by the contract.

Since the current emphasis is placed on testing the expressive power of the contract model and the checking system, we exclude for the present such security related components as a notary. Non-repudiation support will be essential in a practical system, but does not change the contract-checking behaviour significantly, and so can be omitted here.

In addition to the identification of these major components of the architecture, there are a number of issues that need to be clarified about their responsibilities and how they are to interact. The first issue is the closeness of coupling of the monitor to the repository. One would expect policy management tools to act on the repository and changes to be pushed to the monitor, but it needs to be clear when these changes take effect. Changes may take effect:

- immediately they are available in the repository;
- for any new activities starting after the insertion of the changes;
- at a pre-defined time, specified as part of each of the changes.

Distinct business scenarios can be found to correspond to each of these, and the expected behaviour should be specified as part of the change on a case-by-case basis.

The second issue concerns the status of the monitor. Such a component could function as an independent third party, or it could be acting on behalf of one of the existing parties to the contract being checked. If it is independent, it would be expected to report all violations uniformly to all parties. If it is acting as an agent for a particular party, then emphasis will be placed on particular elements of the behaviour, based on the objectives of that party. The perceived checking priorities might well be modified by structures within the environment, as, for example, where some of the parties to a contract have an over-arching allegiance based on common ownership. On the other hand, the checker might be expected to exercise a Chinese Wall policy, giving balanced reports without regard to the authority responsible for it.

Finally, there is a need to decide how close the control loop between the checker and the business processes is to be. At one extreme, the checker could be seen as an out of band activity, recording violations for subsequent, largely unrelated, corrective action. At the other end of the spectrum, the checker could be expected to make short-term responses, triggering corrective action immediately. If such corrective behaviour is foreseen within the contract, the checking component has effectively become a party to the contract in its own right. If the control loop is very closely coupled there will also be a need to consider the interaction of contract violation events with, for example, any transactional structure of the application.

4. A basic organisational model

This model is a refinement of the model underpinning ODP. It uses the foundation concepts defined in the ODP Reference Model [3][4]. However, it goes beyond the ODP work in providing a more integrated treatment of permissions and obligations. Before looking at the detail, we review some of the general properties of such models to set the scene.

4.1. Actions and objects

An *action* represents something that happens; it refers to a tangible behaviour in some system implementation; for instance, a communication between two parties could be considered an action. An action is associated with one or more objects; if more than one object is involved, the action is an interaction.

In object-oriented development, the term *object* is commonly used to refer to a computational entity. In our model, objects may represent more than this; for example: a human that interacts with the system may be represented as an object, as too may the hardware in use by the system, or a data-item within it.

An object has a behaviour, and this behaviour is expressed as constraints on the sequence of actions the object can be involved in. Actions can be used to express the contribution to the behaviour of a system from the system's human users, the computational model employed, the hardware used and the system's data state.

All actions in our model are part of a single type hierarchy which has a root type called simply "action". We employ this type so that we can generalise about the types of actions referred to in specifications. We assume that different naming conventions can be resolved by the mechanisms that support the federation of the contract repository.

The observed occurrence of an action is an *event*. An event will usually relate to one or more objects; this relation will typically indicate which object has initiated the performance of the underlying action. The model is not restricted to this, and events could, for example, be used to report the occurrence of a predefined sequence of actions.

An event will therefore contain information regarding the type of action that has occurred and information about the objects observed to be involved in the action. However, not all participants are equal; there will generally be some causal labelling distinguishing the object that is the initiator of the action. This leads to the need to distinguish action-roles, discussed further in 4.3.

4.2. Permission and obligation objects

The behaviour of an object constrains the actions it can perform. The description of this behaviour can be simplified by dividing it into two aspects, by separating out those constraints that express the holding of rights, authorities or obligations to perform particular actions from the rest of the behaviour. We can express the first kind of behaviour in a uniform way by talking about the permissions and obligations themselves as objects. Thus for an object having a permission to perform a particular action, it is established practice in certain security systems to talk about the object holding a capability and about the management of its authority in terms of the passing or revoking of capabilities. The passing of a capability is, of course, an action, and it is in turn under the control of a capability [9].

Although this has not previously been considered, a similar idea can be applied to obligations. An interaction that results in one of its participants being required to undertake some further actions can be described as passing an obligation object, and obligations can be passed in interactions, subject to the possession of appropriate capabilities. Doing so modifies the behaviour of the receiver of the obligation. Thus, we can pass some object an obligation object to impose further obligations on it, or to pass specific capabilities on to any objects it controls.

What about the repudiation of such obligations by the object holding them? If one were considering the idea of an operating system to police obligations, it would be necessary to prevent an object from simply discarding an obligation. However, this is not a problem in a specification language. It should be kept in mind that the reified obligations introduced here are part of the description of the behaviour required by the contract, and not necessarily an implementation mechanism. We can therefore define the specific conditions under which an obligation object can be destroyed, corresponding to the discharge of the obligation and state that it remains in existence until these conditions are met. A contract will need to say what happens if an object ceases to exist while holding an obligation. The obligation object may survive and become associated with some larger organisational unit.

To emphasise this uniform distinction between objects and unlocalised constraints, and to avoid further overloading the term "capability", we will introduce new terms, referring to reified permissions as *permits*, and to reified obligations as *burdens*.

Just as with capability systems, one can define variants of the model depending on the extent to which checking of the principal exploiting or taking responsibility for the object is prescribed in the model. The "purest" form of permit or burden is one which is characterized entirely by the pattern of behaviour described, but we can also define variants in which the object being passed includes some restriction on the set of principals that can validly invoke it. Such a restriction could be based on community membership, naming or could be an arbitrary predicate.

4.3. Interactions and action templates

The previous section considered the relation between an object and the actions it performs from the perspective of the object concerned.

Interactions are slightly more complicated. The different participants in an interaction may require different permits and pass different burdens, because the participants are involved in different roles with respect to the action.[1] For example, the interaction "deliver goods" involves a *deliverer* action-role and an *acceptor* action-role, and carrying this one action out involves filling both action-roles, checking the two distinct permits. The action will result in a burden of responsibility for payment passing in one direction and a burden for dealing with complaints about faulty goods passing in the other. Note that, in this example, the burdens are instantiated by performing the interaction; a burden (or permit) factory may also be included in the community specification to support this. In other cases, such as an interaction establishing some delegation relationship, an existing burden may be passed on during the interaction.

In any particular instance, these action roles can be associated with the community roles of the objects concerned (see below), but in general the action-roles and the community roles have different scopes and lifetimes. The action role focuses on the properties of the action, abstracting away from the behaviour in which it occurs, while the community role focuses on the behaviour of a collection of objects as a whole and the place of some object within it.

The nature of an action can be represented by an action template, which defines the possible action-roles involved in performing the action. The granting of a permit gives an object the authority to be involved in a particular action-role, and does not necessarily permit it to be involved in other action-roles in the same action. Thus a particular object might have the permit to act as client or as server in a particular client-server interaction and these roles are not simply interchangeable.

[1] The ODP foundation concepts define the concept of role as part of a general instantiation mechanism based on templates. However, this generality has been obscured by the practice in the ODP Enterprise Language of using role as a contraction of the more specific concept of enterprise-role.

4.4. Communities

A *community* is a group of objects that work together to achieve a common goal or goals. Within a community, actions are performed to provide the required behaviour to achieve these goals. The collection of actions that are of interest in a particular community may be referred to as that community's *action alphabet*.

The behaviour of any community is governed by the permits and burdens it holds, and by how it distributes them. The permits and burdens will each originally be received from some source of authority, or created as a result of some authorised action being performed. In our model, these relationships to an authority are made explicit by associating them with the community's membership of one or more superior communities with distinct responsibilities to act as authorities. The authority is responsible for determining the legality of the actions that are performed within the community, and this is modelled by its providing permits or burdens to the community being controlled. The ability to create new permits may be maintained at a single point within a community or shared among multiple points to model a shared style of management or policing.

The authority of a community will dictate permissions and obligations that are assigned to community roles and qualify the behaviour of the objects in these community roles by providing the permits for the action-roles the community object can be involved in. This can apply for any of the actions in the community's action alphabet.

In any model, there must always be a root authority that holds ultimate responsibility for the system as a whole. The root authority may choose to delegate authorisation to child communities to allow certain behaviour as they see fit. The root authority might also choose to withhold permits from, or impose burdens on, a child community to modify its behaviour; in this way, a community may specify constraints on the behaviour of the objects in it, by allocating burdens in a way that the participating objects cannot refuse.

4.5. Roles

Community specifications are organised around the roles that the community members play. They allow the specification to be parameterised in a flexible way. In order that community types may be specified prior to knowing which actual objects will be combined to form the community instances, roles are used to indicate the position that an object may hold within a community. For example, a hospital community may have roles for administrative staff, doctors, nurses and patients.

The rules for any given community will specify the permitted behaviour for objects playing these roles (or any combination of them), constrained by the permits the objects themselves hold, and the burdens placed on them. Since an object may have obtained permits through its membership of multiple communities, the communities can influence their shared behaviour.

Roles can be considered as formal parameters for a community. When a community is formed, objects will be selected, in some way, to play these roles. Roles may also specify their cardinality. Should a role specify a minimum cardinality of greater than zero, then that role must be filled at all times during that community's lifecycle – including when the community is initially formed.

The community specification may also constrain the number of roles filled. In situations where a role must always be played, fallback mechanisms to deal with exceptional circumstances may be required to cater for how a role should be filled in an emergency; for example: to describe how an acting CEO should be appointed in response to the current CEO's sudden arrest.

Since communities are themselves objects, one community can fulfil a role in another, and it is in this way that hierarchies of communities are created. A single object can also fill roles in several communities, coupling their observable behaviours as a result.

When a permit to perform an action-role is given to an object, this permit also applies, unless otherwise stated, to the same action-role in all subtypes of the action. However, refinement of the action may result in new distinctions that can have different associated permissions. This is an example of the legal principle of the specific overriding the general, and allows the specification of child communities to refine the permit, prohibiting some action sub-types. Whether a child community can refine an action that is not permitted by its parent depends on the higher levels of the hierarchy; it would have to be permitted at some higher level and not prohibited at any point on the downward path.

4.6. Policies

Neither technical systems nor organizational systems have a static specification; both evolve over time, but in many cases, the aspects most likely to change can be predicted at the time the initial design is performed [12]. In such cases, parts of the specification can be identified as mutable.

These mutable collections of rules form the community's *policies.* An object that is a member of a community adheres to that community's policy. That is to say, an object must conform to both the fixed rules given in the predefined parts of the community's specification, but it must also conform to the specific collections of rules representing the current policies. In general, the behaviour of a community with a set of policies applied is a refinement of the possible behaviour allowed by the union of all valid policies (the policy envelope). However, this general envelope for community behaviour may not itself represent a valid policy; there may be mutually exclusive choices within the set of possible policies.

A policy is therefore a named placeholder for a piece of behaviour used to parameterise a specification in order to facilitate response to later changes in circumstances. The behaviour of systems satisfying the specification can be modified by changing the policy value, subject to constraints associated with the policy in the original specification.

Stating a policy involves a number of key steps:

- defining a set of circumstances in which the policy is to apply;
- identifying some non-trivial choice to be made under the control of the policy (a specific set of rules);
- identifying an envelope that constrains the range of behaviours that can be specified for the choice made by the policy;
- identifying what information must be available for the policy to interpret;
- defining a decision procedure to be applied in assessing the situation and in actually making the choice;
- defining any invariants that may need to be respected by the system in general for the policy to be effective.

4.7. Contract

In legal terms a contract involves agreement, consideration, certainty and intention, and only sometimes involves written formality.

In our model, a contract is an agreement between a number of objects (although often only two). A contract should be capable of specifying accurately:

a) the order of actions that the objects in the contract should be involved in;
b) the timeliness of these actions;
c) whether these actions transfer permits or burdens, and whether they discharge burdens;

d) fallback strategies to adopt, or penalties to apply, should the above requirements not be met.

Communities model groups of objects and express the ways in which the permits and burdens that the objects in the community hold can be modified by the community behaviour.

A contract instance can be represented as a short-lived community, with rules specifying the obligations of that contract.

4.8. The execution of actions

To summarize the basic model for the performance of actions, the general behaviour of an object is the result of a number of interlocking factors. It depends on the nature of the objects concerned, the communities they form part of and the environment created by the history and composition of the overlapping and overarching communities involved. In particular:

- an object has some broad intrinsic behaviour determined by its object type; an object can never perform actions outside this basic behaviour;
- each of the communities in which the object participates constrains the behaviour available, although the form of constraint may be affected by the way multiple communities are composed or role-filling constraints can be applied;
- the resultant potential behaviour of the object is then restricted to a subset determined by the subset of actions and action roles for which each object holds permits. An object can perform a potential action if it has the appropriate permit, irrespective of the route by which the permit was obtained. It is not in general possible to determine, where a potential action is allowed, the contribution to the composition of community behaviours from a particular community. The potential behaviour is consistent with the composition and hence necessarily with all of the components, but the permit can come as a result of behaviour involving a permit factory from just one community.
- if the above conditions are satisfied, the action can take place, and observations of it are considered valid. Whether or not an object is willing actually to perform the action will depend on the goal seeking objectives of the communities, particularly the burdens it is obliged to discharge. Performing the action can result in creation or transfer of burdens to or from the objects involved.
- if the completion of the action fulfils the obligation represented by a burden supported by any of the objects involved, that burden is discharged and ceases to exist.

Note that there is no inconsistency inherent in an object holding a burden which needs to be discharged by an action the object currently has no permit to perform. The burden will have some associated urgency or failure conditions, such as a time limit, and it may be validly discharged at any earlier time by acquiring the necessary permit. Alternatively, the burden may be disposed of by passing it on to some other object, which has behaviour specifying that it is willing to accept the responsibility.

Consider, for example, the “deliver goods” action discussed earlier. An object in the *deliverer* action-role may acquire a burden to carry out the delivery as a result of the “place order” action being completed, but may remain unable to do so because specific arrangements for delivery have not been made. Thus, although the deliverer object carries the delivery burden, the preconditions for discharging it are not satisfied, and a reasonable contract should not flag a violation.

This situation changes, however, when the acceptor transfers a permit making performance of the “deliver goods” action possible (together with other specific parameterisation of it, such as a delivery address and agreed time). When this is done, delivery can be expected within the agreed interval, and failure to perform the action in that interval is a violation.

5. Composition of communities and policies

5.1. The problem of composition

This section relates to both communities that are used to represent contracts and to those used specifically to model the more structural aspects of a system. There has been a considerable amount of work on the requirements for contract composition; see, for example, [13][14][16]. A brief example illustrates some of the problems arising from the composition of communities.

Consider a situation in which a number of clubs and societies exist, and the social norms dictate that they be explicit about whether their members can perform actions of importance to their objectives. That is to say, they are expected to define a number of policies, where relevant.

As a member of a gun club, I am given a *permit* to fill the ‘shooter’ action-role on the fire-gun action. Considering that I have this permission, a pacifist organisation might choose not let me join[2] their

[2] They may allow me to join regardless of this role but make objections when I exercise the right to fire a gun. This would amount to a dynamic detection of policy infringement rather than a pre-emptive static one.

community as they specify a policy that states that the 'pacifist' role in their society is *prohibited* from filling this action-role. However, a local sailing club has no policy for this action-role and therefore allows both pacifists and shooters to join their community.

What this approach offers is the ability to declare a policy where the ability to fill an action-role may not be of relevance to a community. In our example, the sailing club is not interested in whether or not I am capable of firing a gun, and allows me to join regardless of this. In any circumstances, a permit is needed to perform an action-role. Any given community has an associated action alphabet and only the actions in this set are of relevance to rulings made by this community. By restricting the alphabet of the sailing club to not include the fire-gun action, we can indicate that it is not of interest to this community. The consequence is that members may or may not be capable of performing this action.

This scenario is illustrated in figure 2. The Gun club community (GC) and the Pacifist community (P) both have the fire-gun action in their alphabet. Only the gun club grants permits for this action and the pacifist do not. Members of GC may fire guns, members of P may not, and the SC community is not concerned with such events.

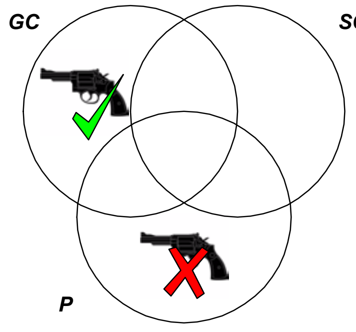

Figure 2. Actions in overlapping communities

It is often the case that a parent community does not prevent behaviour that it does not permit (for example by under-specification). It may still subsequently give a permit to a child community to allow actions it does not itself specify. In this case, if an object performs such an action, it will be judged by the behaviour rules of the other communities of which the object is a member. The use of permits to allow a piece of behaviour is decoupled from the behaviour by which the permit is obtained, and may be derived from different community specifications. The same can be said of objects filling more than one role in a single community; a permit obtained in one role may be used in another, and it is because of this that additional constraints are needed to control role-filling to support separation of duties.

5.2. The authority to authorise

As discussed above, authorisation is managed by passing permits to administer policies from a parent to a child community.

One such permit is that which allows authorisation to be passed in such a manner. Any given community can (if it is permitted to do so by its parent) pass this authority on to the objects that play roles within the community.

Passing this permit on to the objects in the community provides a model where the objects in the community are capable of dictating their own rules and guidelines. This might allow the objects to take on new burdens, or delegate existing ones to other objects in the model.

By making the passing of such permits and burdens explicit, it becomes possible to track the line of responsibility in a system from the objects acting in it back to the goals for each community.

When an object uses a permit to alter its own permissions and obligations, it does so with respect to one of the roles it plays. Therefore, should the object stop playing that role, the obligations and permissions for that role will no longer apply to it; instead they will apply to the next object to play the role. Care should be taken to ensure that obligations continue to be met when roles are swapped in this way – this issue comes down to a correctly designed model (see sub-section 5.3 below).

A community (or contract) will be defined with a particular goal in mind. In order to achieve this goal certain presumptions will almost certainly need to be made regarding the permits of this community; for example, members of the community may require access to privileged data.

In order that communities may be specified without concern for such issues, an appropriate syntax should make all such presumptions explicit. This will also help to facilitate a negotiation stage when the community is realised. At this stage, the requirements of the community can be negotiated with the community's parent community and the appropriate permits granted or refused.

A contract will therefore have two levels of correctness: firstly, the contract should be well formed, or syntactically correct; secondly, in order to be used the contract should be contextually valid. This means that the requirements of the community, as specified, are viable in the context in which it is to be used. So, for example, a contract in which one of the parties grants a cost discount to another will be well formed if it is syntactically correct. However, it will generally only be contextually valid if the community that uses that contract has the ability to grant permits to give cost discounts.

In addition to this, to facilitate community composition, there must be a mechanism that allows a parent community to decide whether or not to grant a permit to a child. This is done by introducing a permit for the delegation action. This too will need to form part of the community's specification and should allow decisions to be made based upon a simple predicate logic notation. This notation should specify to what level the child community has control over this permit. Syntactic features to manipulate composite permits give a compact representation of the related permit to act and permit to delegate control of the action. This can be used to indicate whether the child is allowed to forward the permit to its children or to others.

For example, a parent community may provide its child with a variety of sets of control objects:

a) if a permit for action-role A is provided on its own, then the receiving community can fill that action-role but cannot delegate it to another community (although it does decide which of its members will perform the action – this distinction is not visible until the receiving community is refined, and so is not visible in the permit passed).

b) if a permit for the action-role A is provided together with a burden that would be discharged by performing it, the receiving community can be expected to perform the action eventually. However, the degree of urgency will depend on the details of the burden.

c) if a permit and burden for action-role A are both provided, together with a permit to transfer them to all members of a particular group of objects (other than the child community concerned, which would be a trivial extension for the reasons given in (a)), the receiving community can make a local choice to perform the action or pass on the two objects. It could also pass on just one of them, but this is unlikely to satisfy its goal seeking behaviour.

d) if a permit and burden for action-role A are both provided, together with both a permit and a burden to transfer them to all members of a particular group of objects, then the receiving community must pass them on; if it performs the action itself, the first burden will be discharged, and so the second cannot be.

e) If a burden for action-role A is provided, together with a permit and burden to pass it on, the receiving community must attempt to pass them on. However, their recipient will not be able to discharge the burden unless it already has, or can acquire, the necessary permit from some other route.

These are just a few of the many possible combinations that can lead to a rich variety of behaviours.

Finally, there should be a revocation mechanism in place. The predicates that determine whether permit are granted should be subject to review so that the parent community is given the chance to withdraw permits that it has previously granted.

5.3. Jurisdiction and delegation

Objects may belong to any number of communities. When an object belongs to multiple communities, it is bound by the conjunction of the rules and policies of these communities. The policies of all the communities must be observed by the object at all times.

It is possible that a new community is formed solely for the purpose of allowing two, otherwise disparate, communities to interact. In this case, the objects so enabled in each of the disparate communities become subject to the rules of the new community.

In order to form such a community, it is necessary that the two parents are both prepared to grant permits via the new community in order that the required actions be performed. Contradictions in policy could prevent this from happening; such conflicts could be detected in advance of this and negotiated out before the new community is formed. Alternatively, the new community could be formed regardless of conflicts (as these may be too restrictive) and its members then monitored for illegal behaviour. This will allow communities to be formed more freely, as the policies may contain contradicting rules, but only when an object behaves in violation of these rules will there be a problem. In many cases the object should be aware of its limitations and behave correctly; exceptions to this will be detected. The distinction being made here is similar to the distinction between inherent behaviour and social behaviour in [8].

One area where the relationship between parent and child communities can be quite complex is delegation. Here it is necessary to be able to express dynamic changes in responsibility in a flexible way. This can be done by introducing an additional indirection into the role-filling process. Instead of the normal process of filling community roles by objects, a community can be defined that represents a dynamic role mapping between some role it appears to fill and its member objects. This is, in effect, a mutable compound binding and allows a richer variety of reconfigurable chains of permission and responsibility with a single point of configuration control. The resulting structure can be combined with a suitable reconfiguration trigger to overlay delegation processes on normal behaviour.

5.4. Action hierarchies

At different level of abstraction, it should be possible to specify contracts that define general rules for a community.

Child communities should then be capable of giving yet more specific rules for their members, perhaps even overriding the rules specified by the new child's grandparent, if the permits held by the new parent permit the appropriate actions.

In order to simplify this, all actions are defined as belonging to a single inheritance system, although, to allow constraints between non-adjacent generations, it will be a directed acyclic graph, not a hierarchy. High-level communities, then, can specify rules for abstract actions and the children may be granted permission by the parents to overrule their policies, within limits.

For example, in a high level policy it may be stated that all employees must have prior approval before committing company funds. However, this might be refined in the purchasing department's community by introducing a role distinct from employee, to allow monthly budgets for community members to be used with post-hoc justification; the size of such budgets would vary depending upon the staff seniority.

6. Implementation of the checking framework

6.1. The core interpreter

The aim in designing the core policy interpreter has been to keep it as flexible as possible; we need to be able to experiment with different forms of constraint expression and composition. The policies as a whole are expressed in XML, including the behavioural specifications, which are represented as simple tree-structured forms in which the internal nodes represent process-algebra style composition operators.

An eXecutable XML Language (XXL) framework has been created to support the interpreter. XXL uses an XML DOM to represent a program's canonical structure and interprets it to support threads, stacks, scoped variables, and basic control flow structures. Programs for XXL are therefore well-formed XML documents where the operators in the language are the elements within the document. XXL has been developed in house at UKC to allow simple languages to be prototyped rapidly.

In addition to the core XXL language, additional constructs are defined for the specification of contracts. These constructs launch threads for each contract instance, which in turn process all events for that contract as they are received.

Within a contract there are definitions for the roles that the contract uses, and these are then used in defining a series of clauses that the contract supports. These clauses can be used to define the permitted sequences of actions that the contract will allow, together with any temporal constraints on them. The operators in the core language mean that these sequences of events can be very complex and can even be based upon data values detected in the actions; for example, it might be specified that an agent should apply a 10% discount for any purchase events with a value over $1,000,000.

6.2. Reporting events

Events are detected at some suitable point in the infrastructure close to the point at which the actions being observed were originated. This is done by inclusion of a suitable reporting mechanism within the stub code supporting the interactions, so that no additional requirements are placed on the application. The actual reporting can be provided by a separate remote invocation, but the load placed on the application components can be minimised by using a lightweight publish and subscribe mechanism for event distribution. Using a well-established publish and subscribe mechanism makes it more likely that the participants will have the necessary level of trust in the event reporting mechanism, although the actual event generation will still need to be validated. However, the looser coupling inherent in a publish and subscribe channel might aggravate timing problems.

Clock synchronization is an ever-present problem in distributed systems. Here it manifests itself in the difficulty of establishing a correct global ordering for events. If events are time-stamped at source using an unreliable clock, the order of pairs of events that occur close together may be inverted, leading to the incorrect reporting of behavioural violations.

If it is known that the possible clock differences are bounded and the minimum transit delay between systems known to be greater than this, it is possible to assume that at least a causal ordering is maintained. For larger timing uncertainties, it may be necessary to attempt interpretation of all the possible orderings of events reported within a critical timing window, and to select the most plausible. Even so, frequent timing clashes should be treated as suspicious.

In the simplest situation, there is a one-to-one correspondence between contractual actions and events reported to the monitor. In general, however, there is often a filtering or summarisation process involved in the event delivery, reducing low-level events to more abstract ones at a position near their source, and pruning unwanted events by discarding them early in the process. Such facilities are found in modern publish and subscribe event notification services. However, this is an area where significant optimisation of the software can be made with the use of intelligent system probes. A naïve implementation blindly forwards all information but is very simple to implement.

Finally, there is, in general, a need for the event management subsystem to maintain suitable mappings from the infrastructure components that are reported as the source of events to the more abstract entities described in the contract. While it is often possible to initialise this mapping from observed events (such as interactions with factory objects), there will be cases where some additional explicit information, such as the registration of mappings to support new contract instances, may be needed. In the current prototype, suitable events are made available explicitly at the key stages in the lifecycle of the community roles being checked. We expect the contract specification to be supported by declarations of the minimum set of events to be reported by valid participants, and by the definition of the recognition process to be applied when identifying high-level events from patterns of simpler ones.

6.3. Interpreting Actions

In order to determine whether an observed action is legal, the monitoring tool needs to maintain a model of the communities that are present in the monitored system. This model is dynamic in nature and adapts in reaction to the observed behaviour.

To support a monitoring tool, the observed system must supply information regarding both the community model in use and the actions that take place within it. Both of these requirements can be implemented during contract agreement by using a reliable message passing system. However, exactly what information needs to be sent to the monitor depends on the monitor's needs for a particular contract; for example, object lifecycle actions may be relevant for some contracts and not others. It is assumed here that the mechanisms described above for ensuring that sufficient information is generated and forwarded to the monitor are negotiated in an initial step between the parties involved.

6.4. Dealing with ambiguity of interpretation

One of the problems in interpreting the event stream is that there can well be non-deterministic choice, associated with internal actions in the community behaviour. Since these cannot be observed, the different possible behavioural traces can only be distinguished by examining subsequent actions and it is possible that a considerable number of actions will need to be checked before ambiguity can be resolved.

This implies that the monitor may need to carry forward a number of different possible interpretations, maintaining the set of ambiguous readings until further observation allows the ambiguity to be removed. This applies to both the placement of hidden actions in the trace and the beliefs about the consequent internal state of the objects involved. Techniques for doing this in an efficient manner were described in [1]. They depend on the maintenance of a branching sequence of incremental changes to the assumed state of the objects, which is pruned as alternatives prove incorrect and the changes re-integrated whenever no further ambiguity remains.

Although these techniques have not yet been incorporated into the current prototype, there is no problem in principle in doing so. There are, however, some issues about the way error reports should be generated:

a) the reports could be withheld until ambiguity has been resolved to a sufficient degree for it to be clear that an error has definitely occurred; this might involve some considerable number of further steps taking place after a violation before notification is given;
b) a provisional report could be made as soon as one of the possible readings of the observation indicates a violation, followed by a confirmation or retraction as the ambiguity is resolved.

In either case, this process depends on the incorrect behaviour being detectable eventually from the observations. If a violation of the prescribed behaviour happens that is observationally equivalent to a different, valid, sequence, it will go undetected.

7. The role of contract checking

The creation of flexible and effective contract monitoring components will increase confidence in automated business-to-business interactions. Early adopters are likely to be in repetitive call-off supply agreements where the structures are simple, but the ubiquitous exploitation of such facilities will modify the way business-to-business systems are designed and supported.

The most likely initial consequence will be the more general adoption of explicit electronic contracts, both for commerce but also for a range of infrastructure services, increasing the prevalence of explicit definition of service level agreements, for example. A side effect of this will be the more widespread availability of contract related performance indicators, such as response times.

On a somewhat longer time-scale, adoption of contract definition notations and the reuse of common fragments will encourage some uniformity of style in the expression of contracts in general. The availability of monitoring, in particular is likely to lead to a style of expression in which there is more reliance on the existence of tightly coupled checking. This is because the general availability of checking and automated responses will make exception handling lighter weight and more likely to be used to remove incidental clutter from the main line of the contract specification.

8. Conclusions

This paper has presented a model and prototype infrastructure for the automated checking of business-to-business contracts. It has introduced a novel modelling approach to obligations, unifying the treatment of both permissions and obligations by reifying both, and describing permit and burden passing in a way analogous to the established treatment of capabilities.

As a further test of the general approach to monitoring given in this paper, the authors are currently collaborating with the group at the DSTC to integrate the prototype monitoring component into their business contract infrastructure. This exercise has helped to test the generality of the approach and identify any unintended limitations. The final stage of this collaboration is to port the resultant system onto a SunONE and J2EE infrastructure at the University of Kent to check its applicability to a standard commercial e-commerce environment. A number of the necessary components for this are already available, because the earlier patterns work was constructed on an RMI base, and so modifications of the stub generation mechanism to capture and relay significant events can be reused.

The final goal in testing the applicability of the techniques described here must be deployment in a full commercial environment, but the planning for such a trial needs to be based on further prototype results.

References

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

[2] 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

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

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

[5] ISO/IEC CD 15414, Open Distributed Processing – Enterprise Language, July 1999

[6] Z. Milosevic and A. Bond, “Electronic Commerce on Internet: What Is Still Missing?”, INET’95.

[7] A. Goodchild, C. Herring and Z. Milosevic, “Business Contracts for B2B, Proceedings of the CAISE00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing”; June 5-6, 2000

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

[9] H.M. Levy, “Capability-Based Computer Systems”, Digital Press 1984.

[10] M. V. Wilkes and R. M. Needham. “The Cambridge CAP Computer and its Operating System”, North Holland, New York, 1979.

[11] J. Shapiro, J. Smith and D. Farber. “EROS: A Fast Capability System”, in 17th ACM Symposium on Operating System Principles(SOSP'99), Charleston, USA, December 1999.

[12] P.F. Linington, “An ODP approach to the development of large middleware systems”, in Proc. DAIS99, June 1999.

[13] Z. Milosevic and R. G. Dromey, “On Expressing and Monitoring Behaviour in Contracts”, EDOC 2002

[14] N. Dunlop, J. Indulska and K. Raymond, “Dynamic Conflict Detection in Policy-Based Management Systems”, EDOC 2002.

[15] A.S. Abrahams, D.M. Eyers and J.M. Bacon, "Mechanical Consistency Analysis for Business Contracts and Policies". Proc 5th International Conference on Electronic Commerce Research (ICECR5), Montreal, Canada, 23-27 October 2002.

[16] J. Cole, J. Derrick, Z. Milosevic and K. Raymond, “Author Obliged to Submit Paper before 4 July: Policies in an Enterprise Specification”, Policy Workshop 2000

[17] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. “Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. The Language Specification - Version 2.2”, Research Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, London, 3 April, 2000.

[18] J. Chomicki, J.Lobo and S.Naqvi, “Conflict Resolution Using Logic Programming, IEEE TKDE, 15, 1, 244-249, January 2003.