ODP Enterprise Viewpoint Specification

M.W.A. Steen and J. Derrick
Computing Laboratory, University of Kent at Canterbury, UK
{M.W.A.Steen,J.Derrick}@ukc.ac.uk*

January 13, 2000

Abstract

The Open Distributed Processing (ODP) standardisation initiative has led to a framework by which distributed systems can be modelled using a number of viewpoints. These include an *enterprise viewpoint*, which focuses on the objectives and policies of the enterprise that the system is meant to support. Although the ODP reference model provides abstract languages of relevant concepts, it does not prescribe particular techniques that are to be used in the individual viewpoints. In particular, there is a need to develop appropriate notations for ODP enterprise specification, in order to increase the applicability of the ODP framework.

In this paper we tackle this concern and develop a specification language to support the current draft of the enterprise viewpoint. In doing so we analyse the current definition of the ODP enterprise viewpoint language. Using the UML, a meta-model of the core concepts and their relationships is constructed, and we also investigate to what extent the UML can be used for enterprise viewpoint specification. We then focus on the expression of enterprise policies that govern the behaviour of enterprise objects. We develop a policy language which is a combination of structured English and simple predicate logic, and is built on top of the formal object-oriented specification language Object-Z, into which the complete language is translated. We illustrate the ideas in the paper with a case study that presents an enterprise specification of a library support system.

Keywords: Open Distributed Processing (ODP), ODP enterprise viewpoint, UML, Object-Z, formal methods, enterprise policies, policy specification.

1 Introduction

The aim of this paper is to consider how the Open Distributed Processing (ODP) enterprise viewpoint can be specified using a combination of formal and informal notations.

More than ever organisations depend on complex distributed systems to support their business. The purpose of these systems is to improve the quality, reduce the cost, or increase the timeliness of the business processes they support. In order to meet these goals, it is important that the way in which the business processes are organised are taken into consideration when designing distributed systems. The *enterprise viewpoint* [11] of the Reference Model for Open Distributed Processing (RM-ODP) [15] addresses this issue by providing a language for modelling distributed object systems from the perspective of the enterprise. More specifically, the enterprise viewpoint deals with the purpose, scope and policies for a system and its environment.

*This work was partially funded by the EPSRC under grant "ODP viewpoints in a Development Framework".

The RM-ODP defines a holistic framework for the specification of distributed systems. In order to deal with all aspects and complexities of such systems, the reference model defines five different abstractions (referred to as *viewpoints*) from which distributed systems may be modelled [15]. For each viewpoint, it provides a *viewpoint language* that defines concepts and structuring rules for specifying ODP systems from the corresponding viewpoint. (Note that, in addition to viewpoints, further abstraction levels or views within a particular viewpoint are possible.)

Of the five ODP viewpoints, the enterprise viewpoint is currently the least well defined. Initial efforts of the standardisation community concentrated on the computational and engineering aspects of open distributed processing. Nevertheless, there is growing awareness that enterprise specification has an important role in the development of open distributed systems and in their integration into the enterprise which they serve. In recognition of this trend, the enterprise viewpoint language is currently undergoing extension and refinement within the ISO in order for the enterprise viewpoint to meet these requirements. In this paper, we analyse the current definition of the ODP enterprise viewpoint language. To this end, we construct a meta-model of the core concepts and their relationships using the Unified Modelling Language (UML) [6, 21]. At the time of writing the RM-ODP enterprise viewpoint is a committee draft, and readers should be aware that the final standard may well differ in parts to that described here.

It is important to note, at this point, that the RM-ODP is not prescriptive about the use of any particular notation for the viewpoints. The viewpoint languages, defined in the reference model, are abstract languages in the sense that they define what concepts should be supported, not how these concepts should be represented. Currently, the UML is emerging as a *de facto* standard for object-oriented modelling. A second objective of this paper, therefore, is to investigate to what extent the UML can be used as a concrete notation for enterprise viewpoint specification. Using the UML for enterprise specification has the advantage that many people are already familiar with it and can read enterprise specifications without much additional instruction. At a structural level (i.e. when defining the roles and communities) the UML will be seen to work well.

However, due to the complexity of distributed systems it has been acknowledged that a certain amount of formality in the viewpoint notations is necessary. Indeed, languages such as LOTOS and SDL have been proposed for the computational viewpoint, and Object-Z for the information viewpoint. These arguments also apply to the enterprise viewpoint, particularly when we consider the specification of enterprise *policies*, i.e., those parts of an enterprise specification that govern the behaviour of the enterprise under consideration. The need for more rigorous specification of policies is well recognised [14], and we augment the use of the UML with a simple language for formally specifying enterprise policies.

The language we propose allows specifiers to express policies using predicate logic, and it is built on top of the formal object-oriented specification language Object-Z. The use of predicate logic together with some simple structuring rules, on the one hand, facilitates communication with non-formalists. On the other hand, the approach is rigorous in the sense that policy statements can be translated to Object-Z to pin down their precise meaning. Moreover, this link into a formal language enables formal analysis of policies.

The benefits of formalising enterprise policies, as opposed to using an informal notation, can be summarised as follows:

- It enables policies to be specified concisely and precisely.
- It enables consistency checks to be performed on the policies themselves.
- It enables verification of actual enterprise behaviour against a policy specification.
- Ultimately, it enables the automatic derivation of implementation constructs for enforcing policies.

The potential for the last but one of these arises in ODP via the links with other viewpoints. Although each viewpoint can be developed independently of the others, there are necessarily relationships between them, as they ultimately all describe the same system. correspondence rules identify where the viewpoints overlap or constrain each other. This opens, for example, the possibility of checking a computational viewpoint description for conformance against an enterprise description. However, for this to be achieved it is necessary to provide techniques for checking consistency between viewpoints. Some progress has been made in this direction, and in particular consistency checking strategies have been developed for Z and Object-Z [3, 4]. Object-Z has also been suggested as a notation for the information viewpoint, which could potentially ease this process as both enterprise and information specifications would be written in the same language.

We illustrate our ideas with a simple case study that presents an enterprise specification of a library support system.

The paper is structured as follows. In section 2, we discuss concepts and notations for ODP enterprise viewpoint specification. We first analyse the current definition of the enterprise viewpoint language by constructing a meta-model for it in section 2. Then, in section 3, we propose to use a subset of the UML as a concrete notation for enterprise specification. In section 4, we discuss the issue of policy specification and how this can best be supported by using a combination of informal and formal notations. The proposed notations are given a formal semantics using Object-Z in section 5. Section 6 discusses tool support, and section 7 offers some conclusions.

2 The ODP enterprise viewpoint language

In the RM-ODP, the *enterprise viewpoint* is defined as "a viewpoint on an ODP system and its environment that focuses on the purpose, scope and policies for that system" [10]. The *enterprise viewpoint language* provides the concepts and structuring rules for the specification of an ODP system from the enterprise viewpoint. The ODP enterprise language is an abstract language in the sense that it does not prescribe the use of any particular notation. The ISO is currently in the process of refining and extending the enterprise viewpoint language.

In this section, we analyse the current definition of the enterprise viewpoint language by constructing a meta-model for the language. The meta-model is represented in the UML. Although, as mentioned above, there are shortcomings with respect to the semantics of UML, such a meta-model still has a use in illuminating the text of Part 2 of the RM-ODP. The enterprise language makes use of various concepts defined in Part 2 (Foundations) of the RM-ODP [9]. In so far necessary, we will include these concepts in the meta-model.

2.1 Communities

Central to the enterprise viewpoint is the concept of a *community*. From a business modelling perspective, on the one hand, a community may be viewed as the participants (both people and systems) in a business process. From a system modelling perspective, on the other hand, we can view a community as a system and its environment. The former perspective may be relevant for analysis of the enterprise before a system has been developed and deployed, or for purposes of business process re-engineering (BPR). The latter perspective is useful for identifying the scope of a system. Note that the system under consideration is not constrained to be an IT system, see the definition of system in Part 2 of the RM-ODP.

In the reference model, a community is defined as "a configuration of [enterprise] objects that is formed to meet an objective." The objects in an enterprise model are (abstractions of) the people,

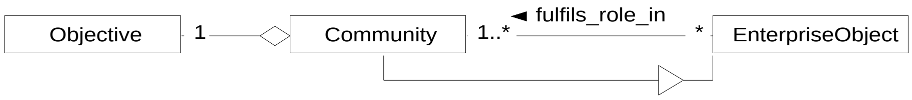

Figure 1: The community concept

systems and other entities that influence whether the community's objective will be achieved. An enterprise object may itself be refined as a community at a more concrete level of detail. Dually, a community may be expressed as a composite object appearing at a more abstract level of detail.

These structuring rules for communities are captured in the UML diagram in figure 1. [1] Note that the enterprise language allows communities to have no associated enterprise objects, allowing very loose specification. If it was felt that this was undesirable, then perhaps a minimum cardinality of one or two should be imposed. Also note that the enterprise objects are not related to communities by containment. Instead an association (fulfils_role_in) is used to model the fact that enterprise objects may exist outside a particular community. However, enterprise objects taking part in a community, i.e., fulfilling roles in that community, are bound by an implicit or explicit agreement to work together towards the objective of that community. This may mean that these objects have to constrain their potential behaviour to satisfy the ruling policy of the community.

2.2 Specification of communities

Communities are specified by means of a contract or community template, which identifies the different roles that objects may play in the community and a policy that governs the behaviour of these objects while fulfilling roles in the community.

There is a subtle distinction between an enterprise *model* and an enterprise *specification*. A model is a description of an existing situation — a real enterprise — and will describe the specific objects making up the community and their actual behaviour. A specification describes how an enterprise should be organised and how it should ideally behave. If a specific model is consistent with a specification, then we say that the model satisfies or conforms to the specification. For a particular specification, there may be many ways of implementing it, resulting in different models.

One possible way of interpreting the structuring rules for community specifications in the draft enterprise viewpoint language document [11], is depicted in figure 2. A *contract* is an "agreement governing part of the collective behaviour of a set of objects. A contract specifies obligations, permissions, prohibitions for the objects involved" [9, Clause 11.2.1]. The community contract is a template specifying the structure and potential behaviour of a community by defining the community's roles and their associations. A *role* is an "identifier for a behaviour" [9, Clause 9.14]. *Role relationships* define the associations that may exists between roles and determine the ways in which roles can interact. This concept is currently not defined in the enterprise language. Note, however, that concepts not defined in a particular viewpoint (such as relationships between roles) may possibly be represented using constructs from Part 2 or from other languages, so, for example, relationships between roles may be described using invariant schemas.

In addition to the structure, a contract specifies the objective and the policies of the community. In the remainder of this section, the identified enterprise concepts are further refined.

[1] The UML diagrams in this meta-model are meant to illustrate the reference model, not replace it. They are therefore simplifications which do not necessarily represent all aspects. For example, in figure 1 the invariants of the relationships are not depicted.

2.3 Enterprise roles

There are four sub-classifications of roles that are of interest to enterprise specifications: *actor* roles and *artefact* roles, and *principal* roles and *non-principal* roles. It is currently not made explicit that these classifications are pair-wise mutually exclusive and total. Thus, all roles can be classified as *non-principal actor*, *non-principal artefact*, *principal actor*, or *principal artefact*. This classification is depicted in figure 3.

An *actor role* is defined as “a role with significant behaviour in that it initiates at least one interaction.” An *artefact role* is defined as “a role for objects that are referenced by some community behaviour, and initiate no interaction in this community.” An enterprise object fulfilling an actor role participates in the actions of that role, whereas an enterprise object fulfilling an artefact role only appears in an enterprise specification because it is referred to in some action of an actor. In a library community, for example, a borrower is an actor, whereas a book is an artefact (also see section 3.2).

2.4 Relationships between roles

The concept of role relationship is not explicitly defined in the current version of the enterprise viewpoint language [11], although the ISO General Relationship model may be useful here. The following therefore represents one possible interpretation of this concept. There are at least two different ways in which roles can be related. Firstly, roles can be related because they interact. In a library community, for example, the role of borrower and the role of loans desk assistant are related because they share the actions of borrowing and returning items. These relationships can usually be left implicit. However, there are other relationships between roles that should be made explicit for modelling purposes. For example, when a borrower borrows a book, a conceptual relationship (a loan) is established between that borrower and that book (also see section 3.2).

2.5 Instantiation

Communities are created by instantiating the corresponding contract template. Instantiation of a contract involves the assignment of enterprise objects to roles. In general, an object may fulfil many roles, in any number of communities. For example, university staff may fulfil a teacher role in an educational community and also fulfil a borrower role with respect to the university library. Alternatively, a single role could be fulfilled by more than one enterprise object — a library has many borrowers, a company may have more than one network administrator, etc.

The relationships between the specification concepts of *community template* and *role*, on the one hand, and their instantiations, i.e., *communities* and *enterprise objects*, on the other hand, is depicted in figure 4.

Note that the fulfils_role_in association between enterprise objects and communities is refined by

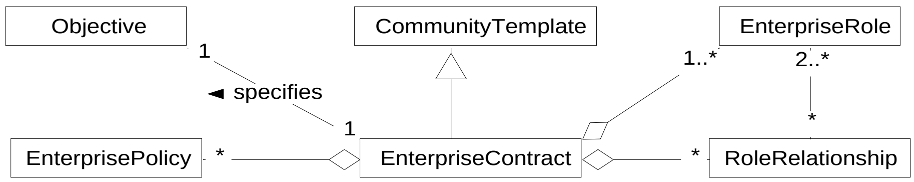

Figure 2: The community contract

this diagram to the *enterprise object* fulfilling a *role* which is contained in the *community template* of the *contract* that is instantiated by the *community*.

In figure 5, the fulfils association between enterprise objects and roles is further refined to the *enterprise object* containing a bit of behaviour, for each *role* that it fulfils, that realises or implements the *role*. We call this bit of behaviour the *role realiser*.

2.6 Enterprise behaviour

Roles are used to identify the behaviour of enterprise objects in the community. To this end, actions are associated with roles (note that actions can in fact be associated with more than one object). In fact, all [enterprise] actions are identified by at least one actor role (the initiator). Since behaviour is defined to be "a collection of actions with a set of constraints on when they may occur," a role definition should also specify these constraints. Thus, a role specifies the behaviour that an object fulfilling that role may exhibit. Note that the concept of *behavioural constraint* is not defined in the RM-ODP. The reason for this is that the type of constraints that may be specified depends very much on the specification language that is being used. In a specification language like LOTOS [5], for example, constraints on behaviour are specified in terms of a temporal ordering of actions, which can then be composed with a synchronisation operator. In a model-based specification language, such as Z [23], on the other hand, behaviour is constrained, amongst others, through invariants on a state space.

Figure 6 depicts one possible interpretation of the clauses pertaining to role behaviour. An *enterprise role* defines some *enterprise behaviour*, which consists of a set of *enterprise actions* and a set of *constraints* on these actions, with each *constraint* relating to at least one *enterprise action*. At least one *enterprise role* participates in each *enterprise action*. In addition, each *enterprise action* is initiated by precisely one *actor role*. Actions that involve more than one role are called *interactions*.

2.7 Enterprise policies

Policies constrain the behaviour of enterprise objects that fulfil actor roles in communities. The policy concept is closely related to that of enterprise behaviour: both represent constraints on the behaviour of objects. They are not the same though. A specification of enterprise behaviour defines the physically possible behaviour of roles in a community. A policy specification may then constrain this behaviour further in order to achieve the objectives of the community. Thus, in general an enterprise specification will include a description of enterprise behaviour together with a policy specification. Another way of viewing the difference is to think of policies as describing the ideal and desirable behaviour within an enterprise, whereas the enterprise behaviour is a model of the actual behaviour. The latter may or may not conform to the ideal expressed in the policies.

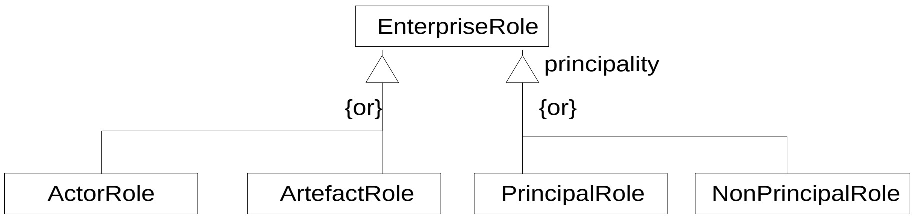

Figure 3: Enterprise roles

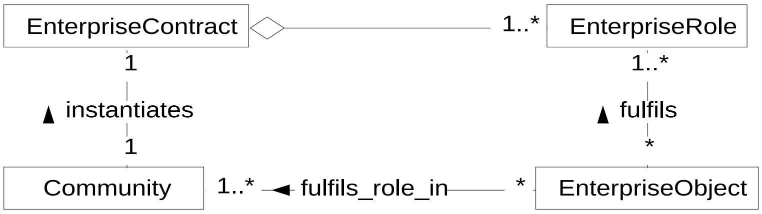

Figure 4: Instantiation of communities

Policy specifications, therefore, often contain prescriptions of what to do in case some rule is violated.

Policies are designed to meet the objective of the community. Policies are specified by the community contract. In addition, Part 2 of the reference model [9, Clause 11.2.7] states that a policy is "a set of rules related to a particular purpose" and that "a rule can be expressed as an obligation, a permission or a prohibition." The structure of policies is depicted in figure 7.

There are basic definitions of the concepts of *obligation*, *permission*, and *prohibition* in Part 2. They have their usual meaning, but pertain to behaviour only. For example, an obligation is a prescription that a particular behaviour is required. These definitions are clearly too restrictive for enterprise specification and have to be refined to include the concept of agency. In the enterprise viewpoint we are often more interested in questions such as *who* has the obligation to perform a certain behaviour, rather than the question *which* behaviour is required. In addition, we would like policies to be first-class concepts, so that we can make statements such as "X has the obligation to grant Y permission to do A." A more in depth discussion of such behavioural enterprise policies and a language for policy specification can be found in section 4.

Instantiation or population rules form another important class of policies. An example of such a policy is that the role of treasurer and the role of auditor cannot be fulfilled by the same person. In section 4.2.1, we present an example of such an instantiation constraint.

2.8 Summary

Figure 8 depicts a summary of the ODP enterprise viewpoint language given as a complete meta-model. In the next section, we use this as the abstract syntax for developing a concrete notation for enterprise specification based on the UML.

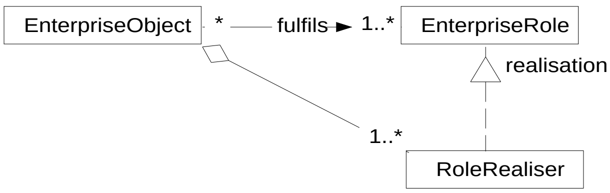

Figure 5: Role realisation

3 Using the UML to specify the enterprise viewpoint

The ODP enterprise viewpoint language is an abstract language in the sense that it does not prescribe the use of any particular notation. It merely identifies a set of concepts and a set of structuring rules. These have been captured in the meta-model presented above, which effectively defines an abstract syntax for the language. In this section, we investigate how the UML can be used to provide a concrete syntax for (parts of) the enterprise viewpoint language.

The Unified Modelling Language brings together a number of widely used object oriented diagramming techniques, such as class diagrams, use case diagrams, state charts and sequence diagrams into a single language. The purpose of this section is to investigate how these diagramming techniques can be used to represent ODP enterprise viewpoint specifications. Using the UML for enterprise specification has the advantage that many people are already familiar with it and can read enterprise specifications without much additional instruction. The disadvantage of using UML is that the language is still evolving and does not have a precise meaning (although there is work on defining a semantics for UML, see [13, 20]).

At a structural level (e.g. when defining the communities) the advantages outweigh the disadvantages. However, when it comes to policy specification, where a precise meaning is crucial, it is necessary to use a notation with a defined semantics, and Section 4 considers how this may best be supported.

The use of UML is illustrated by means of a small case study in enterprise specification.

3.1 An example community

As an example of an enterprise viewpoint specification, we specify the enterprise of a university library. This example is loosely based on the Templeman Library of the University of Kent at Canterbury, but most of the specification will apply to any library.

Anyone will have some idea of what goes on in a library and there clearly is scope for distributed information systems to support the processes of the library. Nowadays, most libraries have one or more automated systems in place to keep track of their collection, the outstanding loans and the borrowers. In the following, we will consider an ODP enterprise viewpoint description of such a system and its environment.

In essence, a library maintains a collection of books, periodicals, and other items, that may be borrowed by its members. A library community comes into being with the establishment of its collection. In the case of a university library, this may have been ordered by decree when the university was founded, but a collection can also be established simply by some people getting together and putting their books in a shared space. In any case, the primary objective of a library community is to share this collection amongst the members.

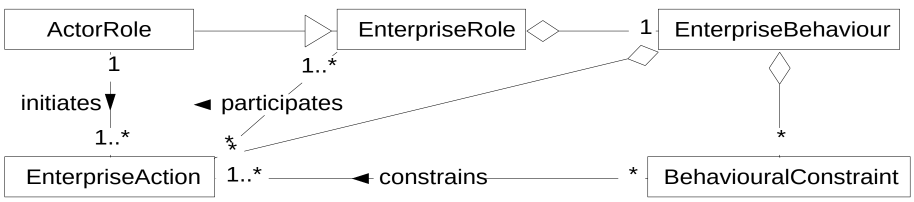

Figure 6: Role behaviour

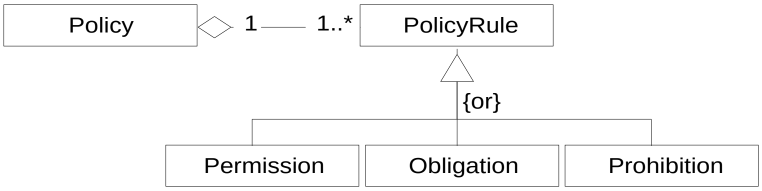

Figure 7: Policy structure

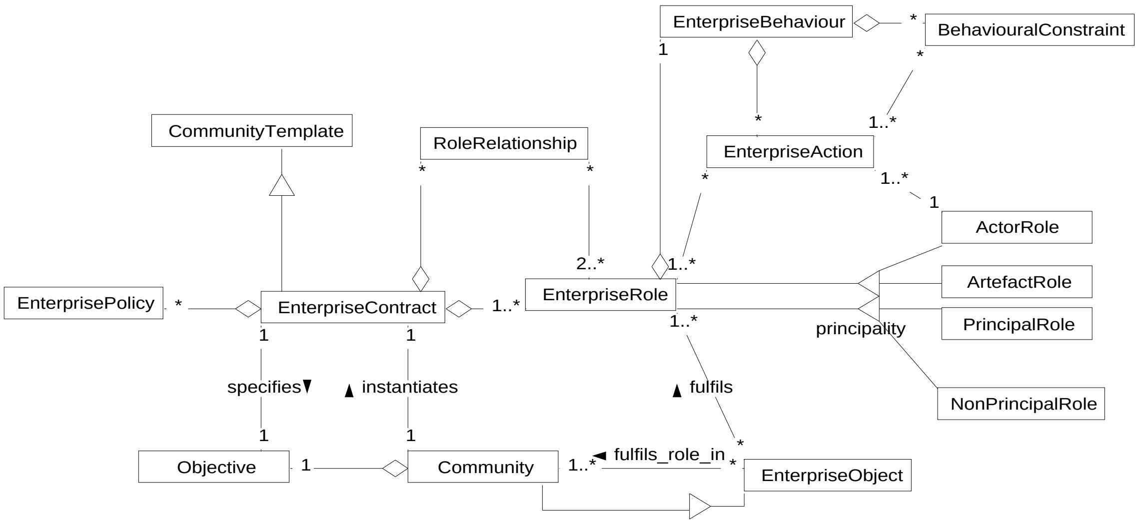

Figure 8: A meta-model for the ODP enterprise viewpoint language

In the UML, a community can be represented by a package (see figure 9). The objective, specified in natural language, can be attached to the community in a description field of the package.

3.2 Modelling roles and their relationships

The roles of the library community are easily identified. From the Templeman Library Regulations (see http://www.ukc.ac.uk/library/about.htm), for example, we can derive the following descriptions.

- The role of **Borrower** may be fulfilled by academic staff, and postgraduate and undergrad-

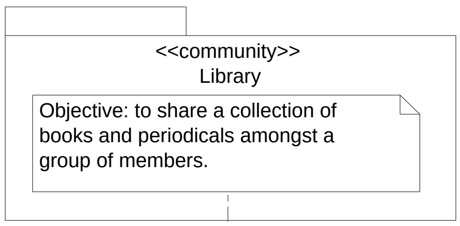

Figure 9: The library community and its objective

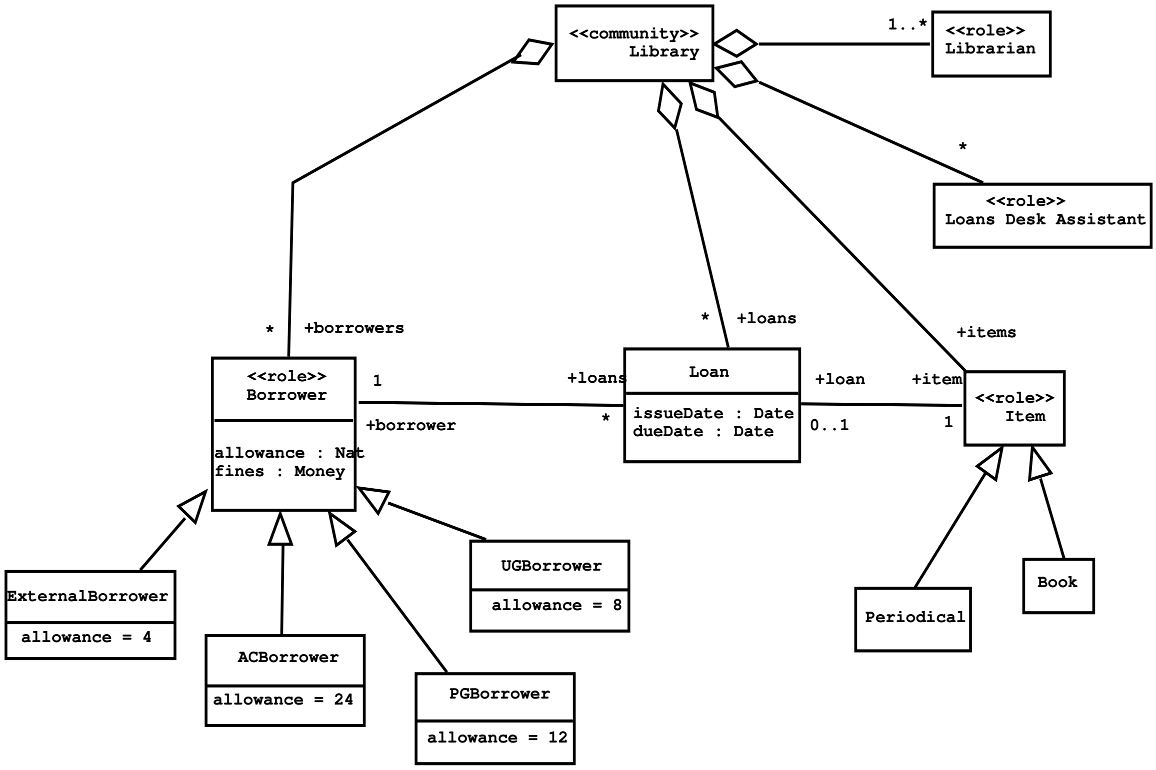

Figure 10: The library community structure

uate students of the University, a related community. In addition, other people and libraries may request special permission to use the Library. They can fulfil the **Borrower** role if such permission is granted.

- The role of **Loans Desk Assistant (LDA)** may be fulfilled by any member of the Library staff. In this role they may issue and receive books and other items available for loan to users.
- Within the library community, the **Librarian** role represents the person with ultimate power. This power may be delegated: "The Librarian may delegate all or any powers under these Regulations to such member or members of the Library staff as may be appropriate."
- The **Item** role is fulfilled by all books and periodicals in the library's collection.

The roles of borrower and librarian are clearly actor roles: they may initiate interactions. Borrowers can, for example, borrow or return items, and librarians can recall items. It is less obvious that loans desk assistants are actors or artefacts. They are involved in the borrow and return actions, but they do not initiate them. Still, we view loans desk assistant as an actor role, because it is fulfilled by human-users. Item clearly is an artefact role. Items do not initiate any interactions, but are involved in most interactions between borrowers and loans desk assistants.

The Loan class, also depicted in figure 10, is an example of a conceptual relationship between the borrower and item roles. Whenever a borrower borrows an item, an instance of this relationship is established. The relationship will again be dissolved upon the return of the item. The advantage of explicitly modelling such conceptual relationships between roles is that we can attach certain attributes with the relationship. In the case of a loan, we may be interested in the issue date and the date the item is due for return. In figure 10 the process of delegation has been elided.

3.3 Modelling enterprise behaviour

The objects fulfilling roles in the library community may be involved in the following enterprise actions:

- The **Borrow** action in which a Borrower requests to borrow a number of items. The request will be handled by a Loans Desk Assistant who may issue the items or deny the request depending on the status of the Borrower and the items.
- The **Return** action in which a Borrower returns previously borrowed items to the loans desk. A Loans Desk Assistant will verify that the items are returned in time and, if necessary, collect a fine.
- The **Reserve** action in which a Borrower places a reservation for an item on loan to another Borrower.
- The **Renew** action in which a Borrower requests to extend the loan period for an item already on loan to him or her. The request will be handled by a Loans Desk Assistant who should refuse the renewal in case the item has been reserved by another Borrower.
- The **Recall** action in which a Borrower is informed to return an item on loan to him or her. After an item has been in the possession of any Borrower for a week, it may be recalled if required by another user. In addition, the Librarian may recall an item on loan at any time.
- The **PayFine** action in which a Borrower pays off a certain amount of his outstanding fines at the loans desk.

The library community can now be refined to a use case diagram (see figure 11) in which the library roles are depicted as UML actors and the enterprise actions they are involved in as use cases. The association stereotype ≪init≫ is introduced to indicate which role initiates the interaction.

The use case diagram in figure 11 simply identifies the possible enterprise actions, for example, it does not describe what happens when a borrower loses an item. In help understand what should happen in such cases, explicit policies specified in the enterprise viewpoint are essential. It does not define the enterprise behaviour. However, we can give a simple example of how enterprise behaviour might be specified. Consider the behaviour of the borrower role. Its actual behaviour will consist in part of borrowing books and then perhaps returning them. However, a borrower can only return a book if it was previously borrowed by that borrower. Therefore we could describe the borrower's behaviour as a causal or temporal ordering of borrow and return actions, the latter happening after the former. We could for example use an activity diagram to specify this constraint on the borrower's behaviour. However, this would not quite capture the fact that this sequence of actions can be interleaved with similar actions for a different loan (of a different item). The precise behaviour can be captured more precisely in LOTOS [5] as follows:

$$Borrower := borrow?item : Item;\ (return!item;\ stop \mathbin{|||} Borrower)$$

This specifies that a borrower can only return an item once she has borrowed it, and that this temporal constraint for a particular loan is repeated for every loan, but that these loans can be interleaved. For example, the sequence

$$borrow(book1);\ borrow(book2);\ return(book2);\ return(book1)$$

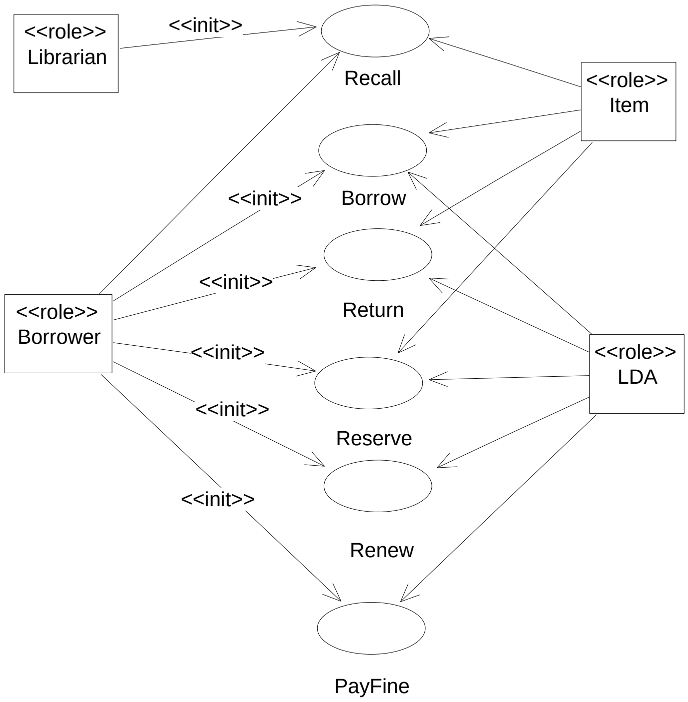

Figure 11: Library Community

is possible according to this specification, assuming that *book1* and *book2* were books that the borrower was entitled to borrow.

The advantage of representing enterprise actions as use cases is that these may be further refined into interaction diagrams (sequence or collaboration diagrams). In the library case, it is not clear though whether such a detailed description of community behaviour would be desired or appropriate. One has to be careful not to be lured into a computational description of the borrowing process in this viepwoint.

3.4 Relating communities

A complete enterprise specification consists of a number of related community specifications. So far, we have considered the library community in isolation, but it is of course closely related to the university community that it serves. In the following, we briefly consider this relationship.

In a university community, we find students, researchers, teachers, heads of department, support staff, etc. In order to relate the university community to the library community, we need to identify amonsgt others:

- the role of **Academic**, which is fulfilled by all research and teaching staff of the university; and
- the role of **Student**, which is fulfilled by all people on undergraduate or postgraduate degree courses.

These roles are fulfilled by people, who may at some point also fulfil the role of borrower in the library community. This relationship, of enterprise objects (in this case: people) fulfilling the

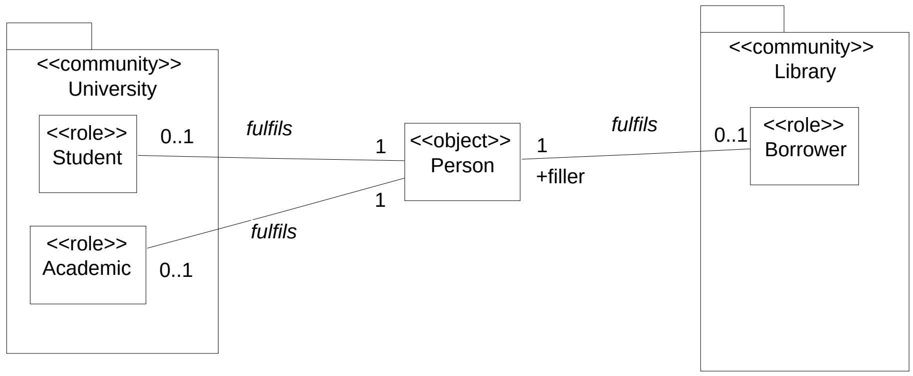

Figure 12: Relating communities

above two roles in different communities, between the university and the library community is depicted in figure 12.

3.5 Summary of mappings

Table 1 summarises how the UML can be used as a concrete notation for the ODP enterprise viewpoint. It describes the mapping from the enterprise language concepts to UML constructs and, if appropriate, the introduced stereotypes.

Enterprise concept	**UML construct**	**Stereotype**
Community	Package	Community
Community	Class	Community
Enterprise Object	Class	Object
Enterprise Role	Class	Role
Role relationship	Class	
Action	Use Case	

Table 1: Enterprise language mappings

A community can either be represented as a package or, when abstracted as an enterprise object, as a UML class. Enterprise objects, or actually, object templates, are represented by a class with the ≪object≫ stereotype. An ODP role is represented as a class with the ≪role≫ stereotype, not to be confused with the UML concept of role. Role relationships are also represented by a class, while enterprise actions are depicted as use cases on a use case diagram.

4 Policy specification

The previous section discussed the use of UML for ODP enterprise viewpoint specification, however, one aspect not considered in depth was the specification of enterprise policies. We consider that issue now. We begin with a discussion on the nature of policies (4.1), illustrated by our library case study (4.2). In section 4.3 a policy specification language is defined. The aim was to develop a language that is sufficiently expressive to capture realistic enterprise policies, yet sufficiently structured and precise to be able to equip it with a formal semantics, section 4.3 illustrates the languages and subsequently section 5 will consider the formal basis for the notation.

4.1 The nature of policies

What is a policy?

Before venturing into the definition of a language for policy specification, we will need to establish what we mean by an enterprise policy. Often a policy is seen as "a high-level overall plan embracing the general goals and acceptable procedures of an organisation" [18]. An enterprise may strive, for example, to maximise its profits, or to obtain a specified market-share. In RM-ODP terminology, however, these would be classified as objectives. There the purpose of a policy for a community is to achieve that community's objective. Thus, it may be company policy that employees should not travel first-class when on company business, in order to maximise profits. In the context of the ODP enterprise viewpoint, it is therefore more appropriate to view a policy as "a definite course [...] of action selected from amongst alternatives [...] to guide and determine present and future decisions" [18].

What can be subject of policy specification?

A policy for a community prescribes what choices objects fulfilling roles in that community may or must make. Hence, only those actions that enterprise objects actually have a choice in can be subjected to policy specification. For example, drivers can decide how fast they drive their car. Therefore, the road traffic regulations defining minimum and maximum speed limits constitute a policy. However, drivers have no direct control over the level of carbon dioxide in their exhaust fumes, but car manufacturers do. Therefore, regulations limiting the levels of carbon dioxide cannot be imposed on drivers, only on manufacturers.

What does a policy specify?

The purpose of policies is to constrain the behaviour and membership of communities in such a way as to achieve their objective. They prescribe what behaviour is allowed and what behaviour is required. Likewise, policies may specify what objects are allowed or required to fill certain roles. An example of the latter is that the role of treasurer and the role of auditor cannot be fulfilled by the same person. In this paper, we focus mainly on the specification of policies that constrain behaviour.

How are policies specified?

As discussed in section 2.7, the ODP enterprise viewpoint language introduces the concepts of *obligation*, *permission* and *prohibition* for specifying the allowable and required behaviour in communities. An obligation expresses that certain behaviour is required. The specification of an obligation will usually refer to the role or group that has the obligation to see to it that the required behaviour actually occurs.

Permissions and prohibitions together prescribe the allowed behaviour. Which form of specification is more appropriate depends on the default policy in a given context. For example, when a pilot receives permission to take off, then he may behave in such a way as to make the plane take off. Without permission, he would not normally be allowed to do so; by default, it is forbidden to take off. In contrast, people are normally allowed to smoke, but they will be explicitly forbidden to do so during take off.

How to deal with violation?

Policies prescribe the ideal or desired behaviour of the participants in a community. Especially if we are dealing with human actors, the actual behaviour may not always conform to the ideal. In the examples above, the pilot might attempt to take off without clearance and a passenger might light a cigarette during take off. Although the former is less likely than the latter, since most pilots are well aware of the risk involved, a policy specification is not complete without also specifying how to deal with such violations. This can be in the form of 'corrective measures' to be taken when the actual behaviour of an object deviates from the ideal behaviour specified for the role it fulfils. Usually, offenders will incur some 'cost'. Speed offenders risk to be fined; not paying your rent, may result in eviction, etc.

4.2 An example enterprise policy

As an example of an enterprise policy, let us look at the regulations of the Templeman Library at the University of Kent at Canterbury. At first sight, this may look like a trivial example, not representative of the large enterprise systems that the ODP enterprise viewpoint will be concerned with. The library policies nevertheless contain some interesting intricacies and issues that one would encounter also on a larger scale.

As discussed in section 3.1 the primary objective of a library community is to share this collection amongst the members, as fairly and efficiently as possible. In order to ensure that this objective is met, a borrowing policy is established, which documents the permissions, obligations and prohibitions for the various roles in the library community. Below we list some actual fragments of the Templeman Library regulations that pertain to the borrowing process.

- *Borrowing rights are given to all academic staff, and postgraduate and undergraduate students of the University.*
- *There are prescribed periods of loan and limits on the numbers of items allowed on loan to a borrower at any one time. These limits are detailed below.*
  - *Undergraduates may borrow 8 books. They may not borrow periodicals. Books may be borrowed for four weeks.*
  - *Postgraduates may borrow 16 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for one month.*
  - *Teaching staff may borrow 24 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for up to one year.*
- *Items borrowed must be returned by the due day and time.*
- *Borrowers who fail to return an item when it is due, will become liable to a charge at the rates prescribed until the book or periodical is returned to the Library.*
- *Failure to pay charges may result in suspension by the Librarian of borrowing facilities.*

Although not explicitly mentioned as such, these rules define the permissions, obligations and prohibitions for the people, systems and artefacts playing a role in the library community. The verb "may" clearly alludes to a permission. The phrase "Undergraduates may borrow 8 books," for example, can be read as: "Undergraduate borrowers have permission to borrow up to 8 books at a time." On the other hand, it could also be seen as an implicit prohibition that "Undergraduate borrowers are forbidden to borrow more than 8 books." This strengthens our argument that permissions and prohibitions are opposite sides of the same coin, together delimiting the allowable behaviour (cf. the discussion in section 4.1). An explicit prohibition is that undergraduates are forbidden to borrow periodicals. Obligations are usually indicated with the verb "must". The rule that "Items borrowed must be returned by the due date and time," for example, could be read as: "Borrowers have the obligation to return any items that they borrowed before the due date." Clearly, there should also be an implicit permission for borrowers to do so. The last two rules deal with violations of this obligation.

The first rule above is different from the rest in the sense that it does not constrain the behaviour of objects fulfilling roles in the library. It is an instantiation rule, stating that only those people fulfilling a role in the related university community (academic, postgraduate, or undergraduate) may fulfil the borrower role.

Even with an example as simple as a University library it is clear that there is a need for formalisation of the policies in order to provide an unambiguous interpretation of them. With the list presented above there are a number of ambiguities. For example, what happens if an actual borrower has a number of different roles? Indeed some of our colleagues (i.e. teaching staff) are also registered as postgraduates on a teacher training course. They have just one library card - how many books can they borrow - 16, 24 or 40? The policies given above are ambiguous, and only through experimentation did we find that the actual implementation of the current software allows 40 books to be taken out in these circumstances. It is not clear that this was what was intended! There is also an issue of how the individual policies are combined, are the implicitly conjoined together, or is a more complex mechanism necessary when faced with related but overlapping policies. This last point leads to the need for a meta-rule which describes the combinations of individual policies. Rarely is such a meta-rule made explicit, and this is often the source of conflice when policies are found to be inconsistent.

4.2.1 Expressing policies

The above regulations define the permissions, obligations and prohibitions for borrowers and the other roles in the library. Most of these policy rules represent constraints on the behaviour of the objects fulfilling those roles. Examples are that "undergraduate borrowers are forbidden to borrow periodicals", that "academic borrowers are permitted to borrow 24 items" and that "borrowers are obliged to return the books they borrow." This last example nicely demonstrates the difference between policies and enterprise behaviour. The enterprise behaviour given for the borrower role in section 3.3 places constraints on the temporal ordering of the borrow and return actions. However, no limit is placed on the number of items that can be borrowed at any one time. The latter constraint is a policy.

When it comes to expressing these behavioural policies with the UML the disadvantages outweigh the advantages. The main problem is that the interaction model of the UML is based on message exchange between objects, whereas interactions in the enterprise viewpoint are seen as bits of shared behaviour. As policies constrain the behaviour of roles, we could consider the Object Constraint Language (OCL) [25] (originally developed by IBM and now incorporated into the UML), in which constraints may be expressed as invariants, or as pre- and post-conditions. However, OCL does not provide powersets, uses the same interaction model, and lacks a formal semantics. Finally, policy statements often involve timing constraints, (e.g., an item has to be returned by its due date), for which there is currently no provision in the UML/OCL combination.

Hence, we have developed a dedicated language for expressing policies to complement the UML in this respect, which is described below. However, although we have chosen Object-Z as the vehicle with which to give the language a precise meaning, a stable and mature version of OCL could be used in its place, offering a greater integration of the community structure with the policy specification.

Nevertheless, there are other types of policy statement that can be expressed within the UML. We refer here to the, so called, instantiation policies, which constrain the way in which roles can be populated. The following rule represents a typical example.

- Borrowing rights are given to all academic staff, and postgraduate and undergraduate students of the University.

This means that only those people fulfilling a role in the university community (academic or student) may fulfil the borrower role. The OCL provides us with a suitable language for expressing such constraints. OCL constraints always have to be interpreted within the context of a UML diagram. For the above constraint, figure 12 provides a suitable context as it already relates the two communities involved. The constraint may now be expressed as follows:

$$\begin{array}{ll} \textbf{context} & \underline{\textbf{Borrower}} \\ \textbf{inv:} & \textbf{self}.\mathit{filler}.\mathit{student} \rightarrow \textbf{notEmpty() or} \\ & \textbf{self}.\mathit{filler}.\mathit{academic} \rightarrow \textbf{notEmpty()} \end{array}$$

There is clearly scope for further work on modelling such constraints in UML and integration into other forms of policy specification.

4.3 A policy specification language

In this section, we define a language for the specification of enterprise policies. Our aim in developing this language was, on the one hand, to provide a language that is sufficiently expressive to capture realistic enterprise policies. On the other hand, we wanted the language to be sufficiently structured and precise to be able to equip it with a formal semantics (see section 5). The result is a combination of keywords and simple predicate logic for formulating policy statements. A complete enterprise specification will consist of a number of related community specifications, each with their own policy. Policies in turn consist of a number of statements expressing a permission, a prohibition or an obligation. The language is introduced by examples drawn from the library case study.

4.3.1 The Policy Context

All policy statements are made in a context. This context is formed by the community or even the specific role to which the statement applies. The context determines which attributes and roles the statement may refer to.

Although we are only concerned with the specification of policies here, we need to describe as much of the structure of the enterprise as is necessary to express the policies. In section 3 above, we have already described how the structure of communities may be specified using the UML. In the following, we refer to figures 10 and 11 for the definition of the library's structure and actions, respectively. The following paragraph serves as a reminder.

The borrower role may be fulfilled by academic staff, and postgraduate and undergraduate students. In this role, they may borrow and return items. In a more elaborate model we might also consider the possibility for borrowers to reserve an item or to renew a loan. For each category of borrower, we introduce a subclass, viz. ACBorrower, PGBorrower and UGBorrower. In this way, we can, if necessary, formulate different policy rules for each category of borrower. The librarian role is fulfilled by the staff of the library. In this role, they may issue items and receive returned items. Item is an artefact role that is fulfilled by all books and periodicals in the library's collection. Items do not initiate any interactions, but are involved in most interactions between borrowers and librarians. Furthermore, borrowers and items may be related by a loan relationship. A loan has two attributes containing, respectively, the date on which the item was issued to the borrower (issueDate) and the date on which the item is due for return (dueDate).

4.3.2 Enterprise Behaviour

Policies are intimately tied to the behaviour of a community. The purpose of a policy is to constrain the behaviour of the participants in a community in such a way as to achieve some desired pattern of behaviour. Policy statements may directly reference certain enterprise actions, and express that they are required or allowed. Another way of constraining the behaviour of a community is to specify that certain states of affair are allowed or not. Any behaviour leading to a forbidden state is then considered prohibited.

From the above, it may be clear that in the formulation of policy statements one may want to reference both the actions roles or communities are able to perform and the states that roles or communities may be in. This naturally leads us to a history-based model of enterprise behaviour, where histories are alternating sequences of states and actions1[2]. This history-based model of enterprise behaviour is one naturally supported by Object-Z. An Object-Z specification is defined in terms of allowable states, which are altered by the occurrence of actions. This is one of the reasons why we have selected Object-Z as a vehicle for expressing policies.

Because of this model of enterprise behaviour, each policy specification will involve both states and actions, and in order to formalise policies we need to identify relevant actions and states from the informal description together with the community description. For example, in the library example it is clear that relevant actions include borrow, return, pay off fines, etc., which are drawn directly from the descriptions of each role, and the use case diagram (figure 11) helps us with their indentification.

4.3.3 Policy Statements

Each policy consists of a number of statements. The policy statements are numbered to facilitate cross-referencing. Each policy statement applies to a role[3], the subject, and represents either a permission, an obligation or a prohibition for that role. The general format is: "A <role> is <modality> to ...," where the modality is one of "permitted", "forbidden", or "obliged".

All three types of policy statements (permissions, obligations and prohibitions) may refer to the execution of an action. The permission for borrowers to borrow items, for example, is expressed as follows:

[R1] A Borrower is permitted to do Borrow(item:Item).

Note that role name and action denotation are capitalised and printed in sans serif font. The action denotation consists of an action name followed by optional parameter specifications in brackets. The Borrow action takes one parameter of type Item. As there is no side-condition, this rule states that borrowers are allowed to perform the borrow action in any state. Of course, there are other rules that prevent this, for example, in the case where the borrower already has reached the maximum number of loans allowed. There also is a rule stating that borrowers are no longer permitted to borrow once their amount of outstanding fines reaches 5 pounds. This is an example of a conditional permission, which is expressed using an if-clause:

[R2] A Borrower is permitted to do Borrow(item:Item), if (fines < 5*pound).

There is an immediate issue here of how these rules are combined which is discussed below.

[2]The history model is consistent with the model of enterprise behaviour put forward by Linington et al. in [14]. There, enterprise behaviour is viewed as a forward-branching tree, where each branching point represents a future choice of action. Each single complete branch of such a tree represents a history.

[3]In general a policy might apply to a collection of roles, it should be clear how to extend the syntax given below to such a scenario.

The prohibition for undergraduate students to borrow periodicals may be expressed as follows:

[R3] A UGBorrower is forbidden to do Borrow(item:Item), where item.isKindOf(Periodical).

This statement contains a where-clause that constrains the parameter of the Borrow action to be also in the set of periodicals. (We model types as sets. Hence, Periodical is a subset of Item, because it is a subtype.)

Obligations are slightly more complicated. They prescribe some required behaviour, and therefore, often contain some deadline for this behaviour to occur. For this purpose, we introduce the before-clause, which contains a condition upon which the obligation should have been fulfilled. For example, the obligation for borrowers to return items that they borrowed by the due date is expressed as follows:

[R4] A Borrower is obliged to do Return(item:Item) before (today > dueDate),
if (loans→exists(loan | oan.item = item)),
where (dueDate = loans→select(loan | loan.item = item).dueDate),
otherwise see R6.

This obligation is conditional upon the item to be returned actually being on loan to the borrower, which is captured by the if-clause. The where-clause constrains the logic variable dueDate to be equal to the dueDate of the loan in question. The logical conditions in the before-, if- and where-clauses are expressed using the Object Constraint Language (the predicates are sufficiently simple that it does not matter whether they are expressed using OCL of Object-Z syntax). The otherwise-clause is used to indicate what will happen if the obligation is violated, which is specified in another policy statement, which is defined below (in section 5.2.2).

Instead of referring to actions, permissions and prohibition (but not obligations) may alternatively refer to a condition, which may or may not be satisfied. The permission for borrowers to borrow up to their allowance, for example, can be expressed as follows:

[R5] A Borrower is permitted to satisfy (loans→size <= allowance).

The condition states that the number of loan relationships that the borrower is involved in should be less or equal to the borrower's allowance. This condition should ideally hold in all states the borrower will be in. Implicitly, it means that any behaviour that changes this condition from being true to being false, is forbidden.

To summarise, our policy statements conform to the grammar below. Here, non-terminals (role, action, condition) are placed between angled brackets (“<”,”>”); everything between square brackets (“[“,”]”) is optional and “|” indicates a choice.

[R?] A <role> is (permitted | obliged | forbidden) to (do <action> [before <condition>] | satisfy <condition>)[, if <condition>][, where <condition>][, otherwise see <number>].

The policy language outlined here is very simple, its purpose is to illustrate the ideas behind the use of predicate logic for a behavioural specification of policies. We have chosen a behavioural interpretation, i.e. one that translates into a model of a system (here written in Object-Z), to open up the possibility of providing links to the other viewpoints, e.g., to the information and computational viewpoints, which may also be formally specified using Object-Z. With an explicit model of the enterprise it is possible to combine it with other viewpoints or check them for consistency. This is not possible using an approach based, for example, on deontic logic [19] which has been proposed as a medium in which to express policy specification.

One issue that we notice even with this simple language is the problem of how to combine the rules. For example, R1 and R2 relate to the same role and action, should we consider them to be inconsistent? There is no uniform correct answer to this, as the interpretation depends upon what the specifier intended. In our interpretation (given via the semantics below) we view these two rules as consistent, and define their combined effect to be the conjunction of R1 and R2. Therefore given a permission, we view a subsequent permission as potentially restricting its scope. Different interpretations to this can be specified and their effects represented in the semantics accordingly.

In order to provide a formal, and therefore mathematically tractable, semantics for our policy specification language, the next section shows how to translate policy statements to the formal specification language Object-Z. This translation enables us then to analyse the policy specification using the theory and tools for Object-Z. In addition, it becomes possible to compare the actual enterprise behaviour, which could also be modelled using Object-Z, with the desired behaviour as specified in the policy.

5 Semantics

In the development of the enterprise viewpoint notations in the previous section, we have sought to provide a formal, i.e., mathematically based, semantics. In this section we discuss how this is achieved. Sections 5.1 and 5.2 show informally how the structure of communities and community policies, respectively, can be expressed in Object-Z. The actual mappings from our Enterprise Viewpoint notations to Object-Z are given in section 5.3. The complete Object-Z specification for our running example is given in section 5.4.

5.1 Specifying communities in Object-Z

In this section, we show by example, using the library example from the previous section, how the structure of communities could be expressed in Object-Z.

Object-Z is an object-oriented extension of the specification language Z. It has been developed over a number of years and is perhaps the most mature of all the proposals to extend Z in an object-oriented fashion. Like Z, it has been advocated as one of the languages suitable for use in the ODP viewpoints, particularly in the information viewpoint.

Object-Z uses a class schema, represented as a named box, to encapsulate a state schema together with the operations acting upon that state. The class schema may include local type or constant definitions, at most one state schema and initial state schema together with zero or more operation schemas. A class may also inherit a number of other classes.

The static structure of a community, i.e., the identified roles and their relationships, translates quite naturally to Object-Z. For each role, we introduce a class definition. The *Borrower* role, for example, is represented as follows:

```
┌─ Borrower ──────────────────────────────
│ ┌──────────────────────────────────────
│ │ allowance : ℕ
│ │ fines : Money
│ └──────────────────────────────────────
│ ┌─ Borrow ─────────────────────────────
│ │ item? : Item
│ └──────────────────────────────────────
│ ┌─ Return ─────────────────────────────
│ │ item? : Item
│ └──────────────────────────────────────
└─────────────────────────────────────────
```

The state schema defines the attributes associated with the *Borrower* role. Each borrower has a certain *allowance*, and an amount of outstanding *fines*. These two attributes are derived directly from the community structure diagram in figure 1. The operations and their parameters correspond to the enterprise actions that borrowers can be involved in. In addition to the *Borrower* class, we would also define an *Item* class and a *Librarian* class.

Loans are relationships between borrowers and library items that have further attributes containing the date of issue and the due date (see figure 1). In Object-Z role relationships are also represented by a class:

Loan

$$\begin{array}{l} borrower : Borrower \\ item : Item \\ issueDate : Date \\ dueDate : Date \end{array}$$

Here the first two attributes are references to the borrower and the item that are related by the loan. For purposes of policy specification, we will also add attributes to the Borrower and Item classes to aid navigation from their instances to loans. From figure 1, we derive that a borrower can have zero or more loans, and an item can be involved in zero or one loan:

Borrower

$$\begin{array}{l} \ldots \\ loans : \mathbb{P}\, Loan \end{array}$$

Item

$$\begin{array}{l} loan : \mathbb{P}\, Loan \\ \hline \# loan \leqslant 1 \end{array}$$

Once classes for all roles and their associations have been derived, we can translate the community itself to an Object-Z class. The attributes of this class are sets of instances of the role classes (*borrowers*, *items* and *librarians*), and instances of the association class (*loans*). The state invariant ensures that the *loan* attributes and the *borrower* and *item* attributes correctly code up the associations in figure 1.

Library

$$\begin{array}{l} borrowers : \mathbb{P}\, Borrower \\ items : \mathbb{P}\, Item \\ librarians : \mathbb{P}\, Librarian \\ loans : \mathbb{P}\, Loan \\ \hline \forall\, loan : loans \bullet\ loan.borrower \in borrowers \\ \qquad \wedge\ loan.item \in items \\ \qquad \wedge\ loan \in loan.borrower.loans \\ \qquad \wedge\ loan.item.loan = \{loan\} \end{array}$$

The class definitions introduced above provide templates that will be further refined below. They will be completed with operations, state invariants, and pre- and postconditions determined by the policy statements.

5.2 Expressing policies in Object-Z

In this section, we show how the policy statements specified in the previous section can be translated into Object-Z. We begin by considering the permissions and prohibitions, which together describe the allowable behaviour and then turn to the obligations, which describe the required behaviour. For each action the Object-Z specification will include an operation with the same name in the class corresponding to the appropriate role.

5.2.1 Permissions and prohibitions

As said before, we view permissions and prohibitions are different views on the same concept. They express actions that may or may not be performed, or conditions that may or may not be satisfied. Whenever they refer to an action, the condition upon which they depend translates into a precondition: positive for permissions, and negative for prohibitions. Whenever they refer to a condition, this condition translates to an invariant on the state space of the role. The permissions and prohibitions from section 4.3.3 are translated as follows:

Rule R1 imposes no restrictions. It translates to the trivial precondition (true) for the *Borrow* action of *Borrower*. Rule R2, on the other hand, is a conditional permission. The condition in the if-clause maps straightforwardly to a precondition. The two preconditions generated by rule R1 and R2 are conjoined to obtain the final precondition for the *Borrow* action of *Borrower*:

```
┌─ Borrower ─────────────────────────────
│ ⋮
│ ┌─ Borrow ─────────────────────────────
│ │ item? : Item
│ ├──────────
│ │ true
│ │ fines < 5 * pound
│ └──────────────────────────────────────
└────────────────────────────────────────
```

Notice that the conjunction of the effect of the two policies is modelling the meta-rule for combining individual rules discussed at the end of section 4.3.3.

Rule R3 is a prohibition. Therefore, its condition is negated to obtain a precondition for the *Borrow* action of *UGBorrower*. Note how *UGBorrower* is obtained from *Borrower* through inheritance.

```
┌─ UGBorrower ───────────────────────────
│ Borrower
│ ┌──────────────────────────────────────
│ │ allowance : ℕ
│ ├──────────
│ │ allowance = 8
│ └──────────────────────────────────────
│ ┌─ Borrow ─────────────────────────────
│ │ item? : Item
│ ├──────────
│ │ ¬ (item? ∈ Periodical)
│ └──────────────────────────────────────
└────────────────────────────────────────
```

Rule R5 specifies a permitted condition. This is translated into a state invariant for *Borrower*. An invariant is a condition that should always be maintained. This corresponds to the informal interpretation given to permitted conditions in section 4.3.3.

$$\begin{array}{|l}
\hline
\textit{Borrower} \\
\quad \begin{array}{|l}
\hline
\mathit{allowance} : \mathbb{N} \\
\mathit{fines} : \mathit{Money} \\
\mathit{loans} : \mathbb{P}\, \mathit{Loan} \\
\hline
\#\mathit{loans} \leqslant \mathit{allowance} \\
\hline
\end{array} \\
\quad \vdots \\
\hline
\end{array}$$

5.2.2 Obligations

The translation of obligations is less straightforward. In general, we can only partly capture the concept of obligation in Object-Z. There are two issues here. One is that obligations usually involve timing constraints, for example, in the form of a deadline before which the obligated behaviour must have occurred. Of course, more complicated permissions may also refer to time. The second is that within these constraints policy may be implicit, whereas in an Object-Z specification all behaviour has to be explicit. Therefore the translation of an obligation produces an Object-Z template in which further explicit modelling may be required for certain actions.

The kinds of obligation we consider here express that a certain action must occur before a certain deadline — a condition that should not hold until the action is performed (i.e., until the obligation is fulfilled)[4]. Rule R4 from section 4.3.3 provides us with a typical example.

[R4] A Borrower is obliged to do Return(item:Item) before (today > dueDate),
if (loans→exists(loan | oan.item = item)),
where (dueDate = loans→select(loan | loan.item = item).dueDate),
otherwise see R6.

Object-Z offers the possibility to express constraints on the behaviour of objects using temporal logic. These temporal logic constraints can be included in an Object-Z class description, and are known as *history invariants*. Obligations could be seen as eventuality properties, and one option is to express them as history invariants. However, the supported fragment of temporal logic is too limited for our purposes. Another disadvantage of this approach is that violations cannot be dealt with within the Object-Z framework, but appear at a meta-level. Whether a history invariant holds or not can only be established by model checking or proof. So, one can establish whether an obligation is violated or not, but it is not possible to formulate corrective measures for violations in the specification itself.

Our solution to this problem is, firstly, to explicitly permit the obliged action and, secondly, to introduce some cost or penalty for the object that violates an obligation, as an incentive to comply with the required behaviour. We feel this is a realistic way of modelling obligations as this is the way in which most laws and regulations enforce desired behaviour.

The permission to return a book on loan translates straightforwardly into a precondition for the *Return* action of *Borrower*:

[4] Note that in general obligations do not always have to specify certain deadlines. Those that do not are known as standing obligations. There may also be obligations which are periodic, for example, requiring daily actions

Borrower

$$
\begin{array}{l}
\vdots \\
\textit{Return} \\
\quad item? : Item \\
\quad \hline \\
\quad \exists\, loan \in loans \bullet loan.item = item?
\end{array}
$$

In the library case, the penalty for not returning items on time is that the borrower becomes liable to a fine, and that if the total amount of fines reaches (say) 5 pounds the permission to borrow will be revoked. This is specified in policy statements R6 and R7.

[R6] The Library is permitted to do UpdateFines(), if (now = midnight and loans→exists(loan | loan.dueDate < today)).

[R7] A Borrower is forbidden to do Borrow(item:Item), if (fines > 5*pounds).

As UpdateFines is an action initiated by no one in particular (it is a community action), it is modelled by an internal operation in Object-Z. Operations are made internal by not listing them in the visibility list at the top of the class. As they do not require the co-operation of other roles, they are executed as soon as their precondition is satisfied. At the same time, an operation to pay off outstanding fines is introduced which will be enabled whenever the value of outstanding fines is non-zero.

In order to model this policy (and any policy that involves time) we need a formalism that will support the description of time. Recent work on extensions to Object-Z have incorporated such a facility [22] by introducing a variable (which we call *now* here) which denotes the current time. The predicate of an operation can now refer to this variable. For example, we can model the *UpdateFines* operation as follows.

Borrower

$$
\begin{array}{l}
\upharpoonright(Borrow, Return, PayFine) \\
\vdots \\
\textit{UpdateFines} \\
\quad \Delta(fines) \\
\quad \hline \\
\quad now = midnight \\
\quad fines' = fines + 20 * pence * \#\{loan : loans \mid today > loan.dueDate\} \\
\\
\textit{PayFine} \\
\quad \Delta(fines) \\
\quad amount? : Money \\
\quad \hline \\
\quad fines > 0 \\
\quad fines' = fines - amount?
\end{array}
$$

These operations could not directly be derived from the policy statements, as we had to assume by how much and how often fines would be increased. Because we are using a model based notation we have to explicitly model this aspect of the policy that is implicitly given in the informal description. However, this model could be viewed as a refinement of the informal policy statements.

Now, since a borrower is forbidden to borrow further items once the total amount of outstanding fines reaches 5 pounds, she eventually has to return to the desired behaviour. Eventually, her only option will be to return the items and settle her fines. Repeated violation of obligations effectively results in the increasing restriction of free choices for a borrower, and ultimately results in only one possible course of action. Notice that the borrower can avoid paying fines by never visiting the library again. This is in accordance with the informal policies given in Section 4.2: all those policies require is that borrowing facilities were suspended in this situation. If enforcement of paying debts is required then this needs to be stated explicitly as a policy.

5.3 Formal translation

The previous section described informally how the community structure and policy statements may be expressed in Object-Z. Here, we make the translation precise. This translation is the basis for the tool support currently being developed.

5.3.1 Translating the community structure

The community structure is translated to Object-Z according to the following rules:

1. Each role definition translates to a class with the same name.
2. If the role specialises another role, this is translated to inheritance between their respective class representations.
3. The role attributes translate to corresponding attribute definitions in the state schema of the class. If an attribute has a associated value, this translates to an equation in the predicate of the state schema.
4. The role action signatures translate corresponding operation schemas of the class representing the role. All parameters are seen as input parameters, and thus obtain a postfixed "?" decoration.
5. Each relationship definition translates to a class with the same name.
6. Relationship attributes translate to corresponding attribute definitions in the state schema of the class.
7. Each community definition translates to a class with the same name (and therefore it follows that a community is a composition relationship).
8. Attributes of the community translate to corresponding attribute definitions in state schema of the class.
9. Translation of the enterprise actions associated with a community can only be partially automated. They translate to corresponding operations of the class representing the community. All parameters are seen as input parameters, and thus obtain a postfixed "?" decoration. It may, however, be necessary to add additional constraints by hand.

5.3.2 Translation of policy statements

Each policy statement applies to a particular role. The translations of the policy statements are added to the Object-Z class representing the role in question, under the following rules:

1. A permission to do an action translates to a precondition for the corresponding operation in Object-Z. If the permission is conditional upon a certain constraint, then this constraint becomes the precondition. If the permission is unconditional, the precondition is true.
2. A permission to satisfy a certain predicate translates to an invariant (a predicate in the state schema) of class corresponding to the role definition.
3. Prohibitions are translated in the same way as permission, except that the associated predicates are negated.
4. Translation of obligations can only be partially automated. Often they will first have to be refined to a strategy for enforcing them (e.g., the borrowers obligation to return their books is enforced through a fining policy). If the obligation to do a certain action is conditional, this condition should at least be translated to a precondition on the corresponding operation.
5. Predicates in policy statements are expressed using OCL or Object-Z syntax. Predicates written using OCL should be translated to their equivalent Object-Z representations.

5.4 The complete Object-Z specification of the library

Application of the translation process results in the following complete Object-Z specification of the library policies. There are no built-in types for dates, time or money in Object-Z. Below, we define these as well as some constants and variables of these types.

$$[Date, Time, Money]$$

$$
\begin{array}{|l}
today : Date \\
day : Date \\
week : Date \\
now : Time \\
midnight : Time \\
pound : Money \\
pence : Money
\end{array}
$$

Item is an artefact role; items never initiate interactions, but can be referenced in interactions initiated by actor roles. Its responsibility is to maintain the availability status of the physical item it represents. Items can be checked out or checked in. An item can never be loaned more than once.

$$
\begin{array}{|l}
\hline
\textit{Item} \\
\hline
\quad onloan : \mathbb{B} \\
\quad loan : \mathbb{P}\, Loan \\
\hline
\quad \#loan \leqslant 1 \\
\hline
\quad \textit{CheckOut} \\
\quad \Delta(onloan) \\
\hline
\quad \neg\ onloan \land onloan' \\
\hline
\quad \textit{CheckIn} \\
\quad \Delta(onloan) \\
\hline
\quad onloan \land \neg\ onloan' \\
\hline
\end{array}
$$

Books and periodicals are kinds of item, which is modelled by inheritance.

Book

Item

Periodical

Item

Loans are associations between borrowers and items (or subtypes thereof). A loan has an issue date and a due date, but no operations.

Loan

$borrower : Borrower$
$item : Item$
$issueDate : Date$
$dueDate : Date$

Borrower is an actor role; they may initiate interactions. Their responsibility is to return items when they are due and to pay fines when appropriate. Borrowers may borrow up to a certain allowance. When the amount of outstanding fines is more than 5 pounds, no further items may be borrowed. An item can only be returned if it was previously borrowed. The UpdateFines operation is not in the visibility list, which means it can happen as soon as its precondition is true without interaction with the environment.

Borrower

$\upharpoonright(Borrow, Return, PayFine)$

$allowance : \mathbb{N}$
$fines : Money$
$loans : \mathbb{P}\, Loan$

$\#loans \leqslant allowance$

Borrow

$item? : Item$

$fines < 5 * pound$

Return

$item? : Item$

$\exists\, loan : loans \bullet loan.item = item?$

PayFine

$\Delta(fines)$
$amount? : Money$

$fines > 0$
$fines' = fines - amount?$

UpdateFines

$\Delta(fines)$

$now = midnight$
$fines' = fines + 20 * pence *$
$\quad \#\{loan : loans \mid today > loan.dueDate\}$

The allowances for the different categories of borrower are defined as follows.

$ACBorrower$

$Borrower$

$$allowance : \mathbb{N}$$
$$allowance = 24$$

$PGBorrower$

$Borrower$

$$allowance : \mathbb{N}$$
$$allowance = 12$$

$UGBorrower$

$Borrower$

$$allowance : \mathbb{N}$$
$$allowance = 8$$

$Borrow$

$$item? : Item$$
$$\neg\,(item? \in Periodical)$$

Note that undergraduate borrowers may not borrow periodicals.

The Librarians role is to prevent unauthorised loans and to maintain the loan records. Librarians may issue and return items or receive fines from borrowers.

$Librarian$

$Issue$

$$borrower? : Borrower$$
$$item? : Item$$

$$\#(borrower?.loans) < borrower?.allowance$$
$$borrower? \in UGBorrower \Rightarrow$$
$$item? \notin Periodical$$
$$borrower?.fines < 5 * pound$$

$Return$

$ReceiveFine$

$$amount? : Money$$

The library community consists of borrowers, items and librarians. The community actions (borrow, return and pay fine) are interactions between two or three different roles. Additional constraints on the borrower and return actions ensure that appropriate loans are created or destroyed.

$$
\begin{array}{|l}
\hline
\textit{Library} \\
\hline
\begin{array}{|l}
\hline
borrowers : \mathbb{P}\, Borrower \\
items : \mathbb{P}\, Item \\
librarians : \mathbb{P}\, Librarian \\
loans : \mathbb{P}\, Loan \\
\hline
\forall\, loan : loans \bullet\, loan.borrower \in borrowers \\
\qquad\qquad\quad \land\, loan.item \in items \\
\qquad\qquad\quad \land\, loan \in loan.borrower.loans \\
\qquad\qquad\quad \land\, loan.item.loan = \{loan\} \\
\hline
\end{array} \\
Borrow \mathrel{\widehat=} [\, \Delta(loans);\ b? : borrowers; \\
\qquad\qquad i? : items;\ l? : librarians \,] \bullet \\
\qquad\quad (b?.Borrow[i?/item?] \parallel \\
\qquad\quad\ i?.CheckOut \parallel \\
\qquad\quad\ l?.Issue[b?/borrower?, i?/item?]) \bullet \\
\qquad\quad [\, \exists\, loan : Loan \bullet loan \notin loans \\
\qquad\quad\ \land\, loans' = loans \cup \{loan\} \land \\
\qquad\quad\ loan.borrower = b? \land loan.item = i? \,] \\
Return \mathrel{\widehat=} [\, \Delta(loans);\ b? : borrowers; \\
\qquad\quad i? : items;\ l? : librarians \,] \bullet \\
\qquad\quad (b?.Return[i?/item?] \parallel \\
\qquad\quad\ i?.CheckIn \parallel l?.Return) \bullet \\
\qquad\quad [\, \exists\, loan : loans \bullet loan.borrower = b? \\
\qquad\quad\ \land\, loan.item = i? \land loan \notin loans' \,] \\
PayFine \mathrel{\widehat=} [\, b? : borrowers;\ l? : librarians \,] \bullet \\
\qquad\qquad (b?.PayFine \parallel l?.ReceiveFine) \\
\hline
\end{array}
$$

6 Tool support

Prototype tool support for the notation described in this paper is currently under development. In this section we describe its scope and architecture. The ultimate aim of the prototype tool-set is to provide an integrated environment for the specification and analysis of open distributed systems according to the ODP viewpoints, and one component of this is an enterprise specification tool.

The enterprise specification tool under development supports the specification of the ODP enterprise viewpoint, and in particular the description of communities and their policies. The immediate advantage of employing such a specification tool is that enterprise specifications can be automatically checked for syntactical errors and semantic inconsistencies. Moreover, once an enterprise specification has been stored electronically, it can be analysed or manipulated for a variety of purposes. For example, it may be related to other viewpoint specifications for the same system for inter-viewpoint consistency checking. Alternatively, model-checking techniques could be employed to verify if some actual enterprise conforms to a policy specification.

In order to facilitate the automatic processing of enterprise descriptions in UML, we have defined a textual representation for community descriptions. This Community Description Language (CDL) can easily be parsed and processed with readily available compiler technology. It is envisaged that the UML diagrams will be translated into this format automatically in future. The formal grammar for this language is given in [24].

The enterprise specification tool is centred around a repository. The repository holds the type and class definitions representing the communities, roles and role relationships in an enterprise specification. Community descriptions, specified in CDL, are parsed and input into the repository. Subsequently, policy specifications can be parsed and input into the repository. The tool outputs an enterprise specification in Object-Z according to the mappings defined in section 5. To ensure portability over a number of platforms the tool is being implemented in Java.

The advantage of using a central repository is that we can change the input or output notations at any time, without having to change any other parts of the tool. The community description language, for example, can eventually be replaced by the UML, by providing a UML to CDL mapping.

7 Conclusions

In this paper, we have used the UML to build up a meta-model for the ODP enterprise viewpoint language. We found that the UML is a useful instrument for such an exercise. It allowed us to depict the relationships between the various concepts defined in the enterprise viewpoint language in a clear, concise and consistent manner. Moreover, the conceptual diagrams are easily communicated between people with varying backgrounds. We managed to have meaningful discussions with both UML literates who knew nothing about ODP, as well as ODP specialists that were not familiar with the UML. Interestingly, the meta-model developed in this report is consistent with a UML model of the enterprise viewpoint drawn up at the latest editing meeting of the ISO working group for ODP. The two models even coincide for large parts. However, some differences were highlighted and suggestions for change were made (e.g. the cardinality restrictions on the enterprise objects in a community). There are currently several proposals relating to a UML metamodel for the enterprise viewpoint (for example see [2, 7, 1, 16]) and we expect in time that these will converge. There are no insurmountable differences between the proposals, however some of the suggestions (e.g. requiring communities to have at least one enterprise object) do have consequences for what constitutes a legal specification, and this impacts upon the templates needed in a particular enterprise notation.

In section 3, we considered the use of the UML as a graphical notation for enterprise viewpoint specification. In our opinion, the UML is certainly useful for depicting the static structure of communities, i.e., the roles and their relationships. For expressing enterprise behaviour, it is only partly useful, because the enterprise viewpoint assumes a different model of interaction than the usual object oriented way of method invocation through message sends. It is not clear yet whether the UML is suitable for expressing behavioural policies. For this reason, we have developed a dedicated policy specification language in section 4. The UML, or actually the OCL, can to a certain extent be used for expressing, so called, instantiation policies and the relationship between these policies and behavioural policies is an area for further work.

The policy specification language developed in section 4 enables specifiers to formalise the enterprise requirements about the allowable and required behaviour of enterprise objects fulfilling roles in communities. In the definition of this language, we have mainly focussed on the behavioural aspects. Clearly, there are many other aspects, such as quality of service, instantiation, security and delegation, which could also be subject to policy specification. Our next step will be to validate the usefulness and practicality of the policy specification language on a more substantial case study. This case study involves the formalisation of the policies for an organisation for air traffic control.

In the description of enterprise policies, we have made use of the concepts of permission, obligation and prohibition. These concepts are studied in a branch of philosophical logic called *deontic logic* — the theory of norms and normative systems. Various authors have suggested the use of deontic logic for the specification of enterprise of information system policies. In particular, see [19] and other proceedings of the DEON conference series for examples. Unfortunately, these logics often suffer from paradoxes, which raise interesting philosophical questions, but are hardly practical for

ODP specification. Nevertheless, [8] offers an interesting approach to the specification of security policies based on deontic logic, which may be relevant also to the enterprise viewpoint. Many issues concerning the specification of the library example using deontic logic are discussed in [12]. There, a method is proposed for identifying *fact*- and *act*-positions, but the method is unlikely to scale even to a complete description of the library example.

Another interesting line of work is presented by Lupu and Sloman [17], who have defined a language for the specification of network/system management policies. Compared to our work, theirs is more down to earth and low-level. In particular, their concept of obligation is a rather operational one in that obligated actions have to be executed immediately. An interesting question for future research is how this low-level policy framework could be used to implement the high-level policies from the ODP enterprise viewpoint.

Tool support is also an issue. There is clearly scope to develop tailored tools based on existing UML tools, providing perhaps structured editors for a number of interlinked viewpoints. In section 6, we have outlined some ideas and requirements for the development of enterprise specification and analysis tools. We are currently working on a prototype tool to perform the translation of policy statements into Object-Z. Future work will include the integration of such a tool into an environment for ODP enterprise viewpoint specification and the development of tools for the analysis of policies.

An important difference and advantage of our approach to policy specification over other work in this area is that our policy language is grounded in a formal model of enterprise behaviour. Moreover, the model we selected — a history model — provides the formal semantics for a well-established formal specification language, viz. Object-Z. Therefore, we define the semantics of policy statements by translating them to Object-Z, and this allows us to use tools already available for Object-Z to analyse policy specifications.

Although the formalisation of enterprise policies is a useful exercise in its own right, it would be much more valuable if they could be related to the computational and engineering features implementing them. Future work therefore needs to consider how high-level policies can be mapped to implementations and the derive the correspondences that these induce between the viewpoints.

References

[1] J.Oyvind Aagedal and Z. Milosevic. ODP enterprise language: UML perspective. In Colin Atkinson, editor, *Proceedings of the Third International Enterprise Distributed Object Computing workshop (EDOC'99)*. IEEE Com. Soc. Press, 1999.

[2] X. Blanc, Marie-Pierre Gervais, and R. Le-Delliou. Using the UML language to express the ODP enterprise concepts. In Colin Atkinson, editor, *Proceedings of the Third International Enterprise Distributed Object Computing workshop (EDOC'99)*. IEEE Com. Soc. Press, 1999.

[3] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, LNCS 1051, pages 287–306. Springer-Verlag, March 1996.

[4] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, September 1999.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language: User Guide.* Object Technology Series. Addison-Wesley, 1998.

[7] J-M. Cornily. Specifying distributed object applications using the reference model for Open Distributed Processing and the Unified Modeling Language. In Colin Atkinson, editor, *Proceedings of the Third International Enterprise Distributed Object Computing workshop (EDOC'99).* IEEE Com. Soc. Press, 1999.

[8] F. Cuppens and C. Saurel. Specifying a security policy: A case study. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW9)*, pages 123–135. IEEE Com. Soc. Press, 1996.

[9] ISO/IEC. Open Distributed Processing - Reference Model - Part 2: Foundations. International Standard 10746-2, ITU-T Recommendation X.902, January 1995.

[10] ISO/IEC. Open Distributed Processing - Reference Model - Part 3: Architecture. International Standard 10746-3, ITU-T Recommendation X.903, January 1995.

[11] ISO/IEC. Open Distributed Processing - Reference Model - Enterprise Viewpoint. Working Draft 15414, ITU-T Recommendation X.911, January 1999.

[12] A. J. I. Jones and M. Sergot. *On the Characterization of Law and Computer Systems: The Normative Systems Perspective*, chapter 12, pages 275–307. Wiley Professional Computing Series. John Wiley & Sons, Chichester, UK, 1993.

[13] S. Kent, S. Gaito, and N. Ross. A meta-model semantics for structural constraints in UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral specifications for businesses and systems*, chapter 9, pages 121–139. Kluwer Academic Publishers, Norwell, MA, September 1999.

[14] P. Linington, Z. Milosevic, and K. Raymond. Policies in communities: Extensing the ODP enterprise viewpoint. In C. Kobryn, editor, *Proceedings of the Second International Enterprise Distributed Object Computing workshop (EDOC'98).* IEEE Com. Soc. Press, 1998.

[15] P. F. Linington. RM-ODP: The Architecture. In K. Raymond and L. Armstrong, editors, *Open Distributed Processing II*, pages 15–33. Chapman & Hall, February 1995.

[16] P.F. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In Colin Atkinson, editor, *Proceedings of the Third International Enterprise Distributed Object Computing workshop (EDOC'99).* IEEE Com. Soc. Press, 1999.

[17] E. Lupu and M. Sloman. A policy based role object model. In Z. Milosevic, editor, *Proceedings of the First International Enterprise Distributed Object Computing Workshop (EDOC'97)*, pages 36–47. IEEE Com. Soc. Press, 1997.

[18] Merriam-Webster's Collegiate Dictionary: Tenth edition, 1998.

[19] J.-J. Ch. Meyer and R. J. Wieringa, editors. *Deontic Logic in Computer Science: Normative System Specification.* Wiley Professional Computing Series. John Wiley & Sons, 1993.

[20] R.B.France, A.S.Evans, K.C.Lano, and B.Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, Accepted for publication, 1998.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language: Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[22] G. Smith and I. Hayes. Towards real-time Object-Z. In *Intergrated Formal Methods'99*, LNCS. Springer-Verlag, 1999.

[23] J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[24] M. W. A. Steen and J. Derrick. Openviews deliverable 99-1: Enterprise viewpoint specification. Technical Report 9-99, Computing Laboratory, University of Kent at Canterbury, 1999.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

Biographies

Dr J. Derrick is a Reader in Computer Science at the University of Kent. Previously he gained a D.Phil from Oxford, and then joined STC Technology Ltd to work on the ESPRIT funded RAISE project. His current interests include developing specification and design techniques for use within ODP and formal definitions of consistency and performance, and he has published extensively within this area. His work has included the following projects: PROST (DTI) *Study on Conformance Testing for ODP*; (DTI EC/4346/92) *A study into Formal Description Techniques for Object Management*; (EPSRC GR/K13035) *Cross Viewpoint Consistency in Open Distributed Processing*; FORMOSA (EPSRC/DTI) *Formalisation of the ODP Systems Architecture*; (British Telecom) *Type Management in Distributed Systems*; (EPSRC) *ODP Viewpoints in a Development Framework*, and (EPSRC) *A Specification Architecture for the Validation of Real-time and Stochastic Quality of Service*.

Dr M. Steen obtained an MSc(Eng) in Computer Science from the University of Twente, the Netherlands, in 1993 and a PhD from the University of Kent at Canterbury (UKC) in 1998. Until recently he was a post-doc researcher at UKC. He is now a Scientific Researcher at the Telematica Instituut in Enschede, The Netherlands. In the past, he has worked on techniques for partial specification in process algebra, in particular on techniques for consistency checking and composition. His current work focuses on the application of these, and other, formal methods in the area of Open Distributed Processing.