Formalising ODP Enterprise Policies

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

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 enterprise viewpoint. In doing so we focus on the expression of enterprise policies that govern the behaviour of enterprise objects. The language we develop is a combination of structured english and simple predicate logic, and is built on top of the formal object-oriented specification language Object-Z. We illustrate its use with a case study that presents an enterprise specification of a library support system.

Keywords— **ODP enterprise viewpoint, formal methods, enterprise policies, policy specification.**

I. Introduction

The enterprise viewpoint is an important part of the reference model for open distributed processing (RM-ODP) [1], [2]. This viewpoint provides a set of concepts and rules for structuring enterprise descriptions. In this paper, we develop a specification language to support this viewpoint and in particular to support the expression of enterprise policies.

The RM-ODP is a framework for specifying and implementing complex distributed systems. The complete specification of any non-trivial distributed system involves a very large amount of information. Attempting to capture all aspects of the design in a single description is generally unworkable. In order to manage this complexity the RM-ODP uses multiple viewpoints that enable different participants to observe a system from a suitable perspective and at a suitable level of abstraction [3]. Requirements and specifications of an ODP system can be made from any of these viewpoints. In the enterprise viewpoint, one can specify the purpose, scope and policies for a system and its environment without having to worry about the details of its implementation.

ODP is a framework for standardisation rather than a design methodology. It does not prescribe particular notations for use in the individual viewpoints. There has thus been a considerable amount of work developing and adapting notations for use within particular viewpoints. Due to the complexity of distributed systems it has been acknowledged that a certain amount of formality in the viewpoint notations is necessary. Languages such as LOTOS and SDL have been proposed for the computational viewpoint, and Object-Z for the information viewpoint. However, there has been little work on instantiating the enterprise viewpoint. Hence, there is a need to develop specification notations that support ODP enterprise description. In this paper, we focus on 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 [4], and here we develop a simple language as a starting point for formally specifying enterprise policies.

The language we propose allows specifiers to express policies in a combination of english and simple predicate logic, and it is built on top of the formal object-oriented specification language Object-Z. The use of english and simple predicate logic, 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. we illustrate the use of the language with a case study that presents an enterprise specification of a library support system.

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 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

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

direction, and in particular consistency checking strategies have been developed for z and Object-Z [5], [6]. 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.

The structure of this paper is as follows. In section II we discuss the ODP enterprise viewpoint and in section III policy specification in ODP. Section IV then introduces our case study, the enterprise specification of policies in a library. This case study is used in section V to define a policy specification language by example. Section VI defines the semantics of policy statements by translating them to Object-Z. We conclude with a discussion of the open issues in section VII.

II. The Enterprise Viewpoint

The RM-ODP defines a framework for the specification and development of open distributed systems. In order to deal with the inherent complexity of such systems, the RM-ODP defines five different abstractions (referred to as *viewpoints*) from which distributed systems may be modelled. For each viewpoint, it provides a *viewpoint language* that defines concepts and structuring rules for specifying ODP systems from the corresponding viewpoint.

Of the five ODP viewpoints, the enterprise viewpoint is currently the least well defined. Nevertheless, there is a growing awareness that enterprise specification has an important role in the development of open distributed systems. In recognition of this trend, the enterprise viewpoint language is currently undergoing refinement and extension within the ISO. As all other viewpoint languages, the ODP enterprise viewpoint language is an abstract language in the sense that it does not prescribe the use of any particular notation. Hence our interest in finding suitable notations for enterprise specification.

Central to the enterprise viewpoint is the concept of a *community*. A community is a group of enterprise objects (comprising both human users and automated systems) that has been formed for a particular objective. 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 the system and its environment. The former perspective may be relevant for analysis of the enterprise before a system is developed, or for purposes of business process reengineering. The latter perspective is useful for identifying the scope of a system.

A community is 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. In a companion paper [7], we discuss the structuring of communities in terms of the roles and their relationships and how the UML can be used to support such specifications. The current paper focuses on the specification of enterprise policies, which are expressed in terms of obligations, permissions and prohibitions governing the behaviour of the enterprise.

III. Policy specification

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” [8]. 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” [8].

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?

In the absense of a policy, enterprise objects are free to follow their own insights in determining their behaviour. The purpose of policies, therefore, is to constrain the behaviour and membership of communities in such a way as to achieve the common 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 or direct behaviour.

How are policies specified?

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. The idea here is that the cost will function as a deterrent and coerce the subject into fulfilling his or her obligation. Likewise, there may be punishments for violating prohibitions.

IV. An example enterprise policy

As an example of an enterprise policy, let us look at the regulations of a university library. The following is loosely based on the regulations of the Templeman Library at the University of Kent at Canterbury (also see http://www.ukc.ac.uk/library/about.htm), but most of it will apply to any library. 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.

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 our case study we will consider an ODP enterprise viewpoint description of such a system. In essence, a library maintains a collection of books, periodicals, and other items, that may be borrowed by its members. The primary objective of a library community thus 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 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 III). 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. Such instantiation policies are dealt with in [7], but will not be considered further here.

V. 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 suffi-

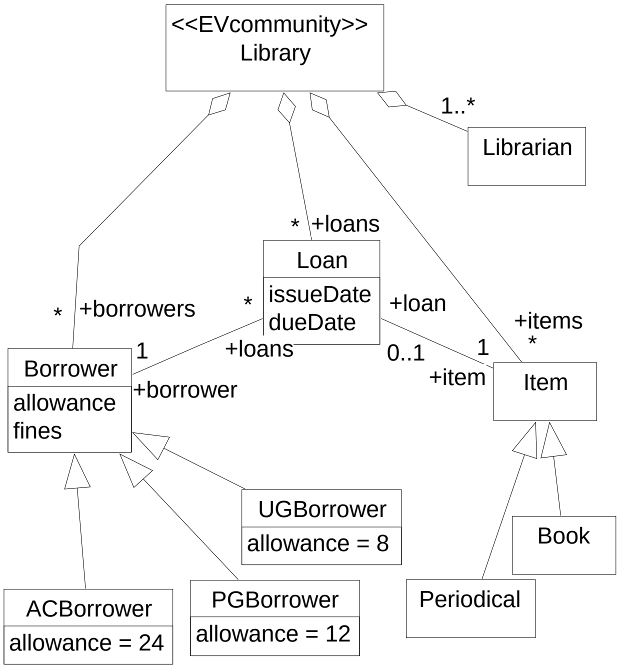

Fig. 1. Library community structure

ciently structured and precise to be able to equip it with a formal semantics (see section VI). The result is a combination of structured English and simple predicate logic for formulating policy statements. However, before policy statements can be made, an enterprise specification must define the specific vocabulary and structure of the enterprise to which the policy applies [4]. 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, which also serve to illustrate the expressive power of the language.

A. *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 [7], we have described how the structure of communities may be specified using the UML. Without going into further details here, we offer figure 1 as an example of a community specification. It depicts the roles and relationships contained in the library community. The (inter)actions the roles may be involved in can be specified in a UML use case diagram [7].

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).

B. *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 actions[1]. 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 specification. 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.

C. *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, the subject, and represents either a permission, an obligation or a prohibition for that role. The general format is: "A <role> is

[1]The history model is consistent with the model of enterprise behaviour put forward by Linington *et al.* in [4]. 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.

<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).

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

R3 A UGBorrower is forbidden to do Borrow(item:Item), where Periodical→includes(item).

This statement contains a where-clause that constrains the parameter of the Borrow action to be also in the set of periodicals. (Types are 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 | loan.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 (OCL) [9], originally developed by IBM and now incorporated into the UML. The otherwise-clause is used to indicate what will happen if the obligation is violated, which is specified in another policy statement, not included here.

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, policy statements should satisfy 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>].

VI. Expressing policies in Object-Z

In order to provide a formal, and therefore mathematically tractable, semantics for our policy specification language, we show here 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. It also opens 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.

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.

Given that the static structure of a community may be specified using UML one could ask whether it is possible to use UML together with OCL, say, to describe the policies formally. However, OCL suffers from two difficiencies which make it unsuitable for our purposes. The first is that the description language it contains is not sufficiently expressive (e.g., there are no powersets). The second is that it lacks a semantics, and our purpose here (and reason for using Object-Z) is that it has a precise meaning derived from its formal semantics.

A. Translating the community structure

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:

$$
\begin{array}{l}
\textbf{class } \mathit{Borrower} \\
\quad \begin{array}{|l}
\mathit{allowance} : \mathbb{N} \\
\mathit{fines} : \mathit{Money}
\end{array} \\
\quad \mathit{Borrow} \\
\quad \begin{array}{|l}
\mathit{item?} : \mathit{Item}
\end{array} \\
\quad \mathit{Return} \\
\quad \begin{array}{|l}
\mathit{item?} : \mathit{Item}
\end{array}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{class } \mathit{Loan} \\
\quad \begin{array}{|l}
\mathit{borrower} : \mathit{Borrower} \\
\mathit{item} : \mathit{Item} \\
\mathit{issueDate} : \mathit{Date} \\
\mathit{dueDate} : \mathit{Date}
\end{array}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{class } \mathit{Borrower} \\
\quad \begin{array}{|l}
\ldots \\
\mathit{loans} : \mathbb{P}\, \mathit{Loan}
\end{array}
\end{array}
\qquad
\begin{array}{l}
\textbf{class } \mathit{Item} \\
\quad \begin{array}{|l}
\mathit{loan} : \mathbb{P}\, \mathit{Loan} \\
\hline
\#\mathit{loan} \leqslant 1
\end{array}
\end{array}
$$

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.

$$
\begin{array}{l}
\textbf{class } \mathit{Library} \\
\quad \begin{array}{|l}
\mathit{borrowers} : \mathbb{P}\, \mathit{Borrower}_{©} \\
\mathit{items} : \mathbb{P}\, \mathit{Item}_{©} \\
\mathit{librarians} : \mathbb{P}\, \mathit{Librarian}_{©} \\
\mathit{loans} : \mathbb{P}\, \mathit{Loan}_{©} \\
\hline
\forall\, \mathit{loan} : \mathit{loans} \bullet\ \mathit{loan.borrower} \in \mathit{borrowers} \\
\qquad \wedge\ \mathit{loan.item} \in \mathit{items} \\
\qquad \wedge\ \mathit{loan} \in \mathit{loan.borrower.loans} \\
\qquad \wedge\ \mathit{loan.item.loan} = \{\mathit{loan}\}
\end{array}
\end{array}
$$

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.

B. Translation of policy statements

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.

B.1 Permissions and prohibitions

As said before, 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. An action may be referred to in more than one policy. In the Object-Z translation the preconditions obtained from the individual rules are conjoined together in the Object-Z operation corresponding to that action. With these ideas in place the permissions and prohibitions from section V-C 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
$fines < 5 * pound$

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

Borrow
$item? : Item$

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

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 V-C.

Borrower

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

$\#loans \leqslant allowance$

...

B.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). Rule R4 from section V-C provides us with a typical example.

R4 A Borrower is obliged to do Return(item:Item) before (today > dueDate), if (loans.exists(loan | loan.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*:

Borrower
...

Return
$item? : Item$

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

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 at this level of abstraction), 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 [10] by introducing a variable (which we call *now* here) which denotes the current time. The predicate of an operation can then refer to this variable. For example, we can model the *UpdateFines* operation as follows.

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

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, he or she eventually has to return to the desired behaviour. Eventually, his or her only option will be to return the items and settle his or 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.

VII. Discussion and Conclusion

In this paper, we have provided a simple language for the specification of enterprise policies in the ODP enterprise viewpoint. This language 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 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 our policy 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 or information system policies. In particular, see [11] 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, [12] 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 [13]. 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 [14], 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.

An important difference and advantage of our approach 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 could define the semantics of policy statements by translating them to Object-Z. Furthermore, this allows us to use tools already available for Object-Z to analyse policy specifications. 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.

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. In future, we should therefore consider how high-level policies could be mapped to implementations.

References

[1] "Open Distributed Processing – Reference Model: Foundations," ITU-T Recommendation X.902 | ISO/IEC IS 10746-2, Jan. 1995.

[2] "Open Distributed Processing – Reference Model: Architecture," ITU-T Recommendation X.903 | ISO/IEC IS 10746-3, Jan. 1995.

[3] P. F. Linington, "RM-ODP: The Architecture," in *Open Distributed Processing II*, K. Raymond and L. Armstrong, Eds. Feb. 1995, pp. 15–33, Chapman & Hall.

[4] P. Linington, Z. Milosevic, and K. Raymond, "Policies in communities: Extensing the ODP enterprise viewpoint," in *Proceedings of the Second International Enterprise Distributed Object*

Computing workshop (EDOC'98), C. Kobryn, Ed. 1998, IEEE Com. Soc. Press.

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

[6] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen, "Constructive consistency checking for partial specification in Z," *Science of Computer Programming*, December 1999, To appear.

[7] M. W. A. Steen and J. Derrick, "Applying the UML to the ODP enterprise viewpoint," Tech. Rep. 8-99, Computing Laboratory, University of Kent at Canterbury, May 1999.

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

[9] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

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

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

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

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

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

Maarten Steen obtained a PhD in Computer Science from the University of Kent at Canterbury in 1998. The work presented in the present paper was carried out while he was employed as a post-doc researcher at UKC. He is currently employed by the Telematica Instituut in the Netherlands. His research interests are in the application of formal methods to distributed system design.

John Derrick is a senior lecturer at the University of Kent at Canterbury. 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.*

Appendix

I. The Complete Library Specification

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.

Item

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

CheckOut

$$
\begin{array}{|l}
\Delta(onloan) \\
\hline
\neg\ onloan \wedge onloan'
\end{array}
$$

CheckIn

$$
\begin{array}{|l}
\Delta(onloan) \\
\hline
onloan \wedge \neg\ onloan'
\end{array}
$$

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

Book

$$\begin{array}{|l} Item \end{array}$$

Periodical

$$\begin{array}{|l} Item \end{array}$$

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

Loan

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

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

$$\upharpoonright(Borrow, Return, PayFine)$$

$$\begin{array}{l} allowance : \mathbb{N} \\ fines : Money \\ loans : \mathbb{P}\, Loan \\ \hline \#loans \leqslant allowance \end{array}$$

Borrow
$$\begin{array}{l} item? : Item \\ \hline fines < 5 * pound \end{array}$$

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

PayFine
$$\begin{array}{l} \Delta(fines) \\ amount? : Money \\ \hline fines > 0 \\ fines' = fines - amount? \end{array}$$

UpdateFines
$$\begin{array}{l} \Delta(fines) \\ \hline now = midnight \\ fines' = fines + 20 * pence * \\ \qquad \#\{loan : loans \mid today > loan.dueDate\} \end{array}$$

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

ACBorrower

$$\begin{array}{l} Borrower \\ \hline allowance : \mathbb{N} \\ \hline allowance = 24 \end{array}$$

PGBorrower

$$\begin{array}{l} Borrower \\ \hline allowance : \mathbb{N} \\ \hline allowance = 12 \end{array}$$

UGBorrower

$$\begin{array}{l} Borrower \\ \hline allowance : \mathbb{N} \\ \hline allowance = 8 \end{array}$$

Borrow
$$\begin{array}{l} item? : \downarrow Item \\ \hline \neg\,(item? \in Periodical) \end{array}$$

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
$$\begin{array}{l} borrower? : \downarrow Borrower \\ item? : \downarrow Item \\ \hline \#(borrower?.loans) < borrower?.allowance \\ borrower? \in UGBorrower \Rightarrow \\ item? \notin Periodical \\ borrower?.fines < 5 * pound \end{array}$$

Return

ReceiveFine
$$\begin{array}{l} amount? : Money \end{array}$$

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 borrow and return actions ensure that appropriate loans are created or destroyed.

Library

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

$$\begin{array}{l} Borrow \mathrel{\widehat=} [\, \Delta(loans) b? : borrowers; \\ \qquad i? : items;\ l? : librarians \,] \bullet \\ \qquad (b?.Borrow[i?/item?] \ \| \\ \qquad i?.CheckOut \ \| \\ \qquad l?.Issue[b?/borrower?, i?/item?]) \bullet \\ \qquad [\, \exists\, loan : Loan \bullet loan \notin loans \\ \qquad \wedge\, loans' = loans \cup \{loan\} \wedge \\ \qquad loan.borrower = b? \wedge loan.item = i? \,] \\ Return \mathrel{\widehat=} [\, \Delta(loans) b? : borrowers; \\ \qquad i? : items;\ l? : librarians \,] \bullet \\ \qquad (b?.Return[i?/item?] \ \| \\ \qquad i?.CheckIn \ \| \ l?.Return) \bullet \\ \qquad [\, \exists\, loan : loans \bullet loan.borrower = b? \\ \qquad \wedge\, loan.item = i? \wedge loan \notin loans' \,] \\ PayFine \mathrel{\widehat=} [\, b? : borrowers;\ l? : librarians \,] \bullet \\ \qquad (b?.PayFine \ \| \ l?.ReceiveFine) \end{array}$$