Writing ODP Enterprise Specifications in Maude A Case Study*

Francisco Durán and Antonio Vallecillo

ETSI Informática. Universidad de Málaga. Spain.
{duran,av}@lcc.uma.es

Abstract. Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In this paper we explore the possibility of using Maude as a formal notation for writing and reasoning about RM-ODP enterprise specifications. An example is used to illustrate our proposal and to compare it with other approaches that also try to formalize this viewpoint.

1 Introduction

Distributed systems are inherently complex, and their complete specifications are so extensive that fully comprehending all their aspects is a difficult task. To deal with this complexity, system specifications are usually decomposed through a process of separation of concerns to produce a set of complementary specifications, each one dealing with a specific aspect of the system. Specification decomposition is a well-known concept that can be found in many architectures for distributed systems. For instance, the Reference Model of Open Distributed Processing (RM-ODP) framework [5] provides five generic and complementary viewpoints on the system and its environment: *enterprise*, *information*, *computational*, *engineering* and *technology* viewpoints. They enable different participants to observe a system from a suitable perspective, and at a suitable level of abstraction [8].

One of these viewpoints—the *enterprise* viewpoint—focuses on the purpose, scope and policies for the system and its environment. It describes the business requirements and how to meet them, but without having to worry about other system considerations, such as particular details of its implementation.

Although the ODP reference model provides abstract languages for the relevant concepts, it does not prescribe particular techniques to be used in the individual 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 they should be represented. Several concrete languages or notations have been proposed for the different viewpoints by different authors, which seem to agree on the need of some formal methodology for such languages/notations in order to be able to deal with the complexity of distributed systems. For example, formal description techniques (FDTs) such as LOTOS or SDL have been proposed for the computational viewpoint [5], and

* Partially supported by CICYT project TIC99-1083-C02-01.

Object-Z for the information and enterprise viewpoints [16]. It is perhaps the enterprise viewpoint the one for which it is more unclear the FDT to be used, although a certain need to develop appropriate notations for ODP enterprise specifications has been identified by different authors [1, 3, 8, 14, 5] in order to increase the applicability of the ODP framework.

In this paper we would like to explore another alternative for specifying the enterprise viewpoint. We propose Maude, an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems [4, 11]. This choice not only introduces new benefits over the Object-Z approach (and other approaches), but also allows to overcome some of its limitations, as we shall discuss later.

The structure of this document is as follows. First, Sections 2 and 3 serve as brief introduction to the ODP enterprise viewpoint and Maude, respectively. Then, Section 4 presents our proposal, describing how to write enterprise specifications in Maude. Section 5 is dedicated to a small case study that illustrates our approach, which is compared to other similar proposals in Section 6. Finally, Section 7 draws some conclusions and describes some future research activities.

2 The Enterprise Viewpoint

An enterprise specification of an ODP system is an abstraction of the system and a larger environment in which the ODP system exists, describing those aspects that are relevant to specifying what the system is expected to do in the context of its purpose, scope and policies [6]. An enterprise specification describes the behaviour assumed by those who interact with the ODP system, explicitly including those aspects of the environment that influence its behaviour —environmental constraints are captured as well as usage and management rules.

A fundamental structuring concept for enterprise specifications is that of a *community*. A community is a configuration of enterprise objects modeling a collection of entities (e.g., human beings, information processing systems, resources of various kinds, and collections of these) that are subject to some implicit or explicit contract governing their collective behaviour, and that has been formed for a particular objective.

The scope of the system is defined in terms of its intended behaviour, and this is expressed in terms of *roles*, *processes*, *policies*, and their *relationships*. Roles identify abstractions of the community behavior, and are fulfilled by enterprise objects in the community. Processes describe the community behaviour by means of (partially ordered) sets of *actions*, which are related to achieving some particular sub-objective within the community. Finally, policies are the rules that constraint the behavior and membership of communities in order to make them achieve their objective. A policy can be expressed as an obligation, an authorization, a permission, or a prohibition. Actions contrary to rules are known as violations.

Of the five RM-ODP viewpoints, the enterprise viewpoint is perhaps the least well defined. Initial efforts of the standardization community concentrated on the computational and engineering aspects of open distributed processing. Nevertheless, there is a

growing awareness that enterprise specification has an important role in the development of open distributed systems and in their integration into the enterprise they serve.

3 Rewriting Logic and Maude

Rewriting logic [9] is a logic in which the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of types (sorts) and operations, and E is a set of (conditional) equational axioms. The dynamics of such a system is then specified by rewrite *rules* of the form $t \to t'$, where t and t' are Σ-terms that describe the local, concurrent transitions possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied. In fact, the rewriting process operates on equivalence classes of terms modulo E, in such a way that rewriting is liberated from the syntactic constraints of term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty.

More precisely, given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form "$r : [t]_E \longrightarrow [t']_E$ *if* C", where t and t' are Σ-terms, C is a boolean condition[1] and, given a term s, $[s]_E$ denote the E-equivalence class of s. A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where (Σ, E) is an equational specification that axiomatizes the system's state space, L is a set of *labels*, and R is a set of labeled sentences called *rewrite rules* that axiomatize the system's local transitions, i.e., its dynamic behavior. Rewriting logic is parameterized by its underlying equational logic, which can be unsorted, many-sorted, order-sorted, or membership equational logic [10].

Maude [4] is a high-level language and a high-performance interpreter and compiler that supports equational and rewriting logic specification and programming of systems. The underlying equational logic chosen for Maude is membership equational logic, that has sorts, subsorts, operator overloading, and partiality. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In particular, it supports very well concurrent object-oriented computation [11].

In Maude, object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. Each class is declared with the syntax "`class` $C \mid a_1 : S_1, ..., a_n : S_n$", where C is the name of the class, a_i are attribute identifiers, and S_i are the corresponding sorts of the attributes. Objects of a class C are then record-like structures of the form `<` $O : C \mid a_1 : v_1, ..., a_n : v_n$ `>`, where O is the name of the object, and the v_i are the current values of its attributes. Objects can interact in a different number of ways, including message passing.

[1] See [9] for the concrete form of rule conditions.

In a concurrent object-oriented system the concurrent state, which is called the *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the *communication events* between some objects and messages. The general form of such rewrite rules is

$$
\begin{array}{lll}
crl\ [r] & : & M_1 \ldots M_m \\
 & & < O_1 : C_1 \mid atts_1 > \ldots < O_n : C_n \mid atts_n > \\
 & & \longrightarrow < O_{i_1} : C'_1 \mid atts'_{i_1} > \ldots < O_{i_k} : C'_r \mid atts'_{i_k} > \\
 & & \quad < Q_1 : C''_1 \mid atts''_1 > \ldots < Q_p : C''_p \mid atts''_p > \\
 & & \quad M'_1 \ldots M'_q \\
 & & \quad if \;\; Cond \ .
\end{array}
$$

where r is the rule label, M_i are messages, O_i and Q_j are object identifiers, C_i, $C'{}_j$ and C''_h are classes, $i_1, \ldots, i_k$ is a subset of $1 \ldots n$, and $Cond$ is a boolean condition (the rule 'guard'). The result of applying such a rule is that:

- messages $M_1 \ldots M_m$ disappear, that is, they are consumed;
- the state, and possibly the class of objects $O_{i_1}, \ldots, O_{i_k}$ may change;
- all the other objects Oj vanish;
- new objects $Q_1, \ldots, Q_p$ are created; and
- new messages $M'_1 \ldots M'_q$ are created, i.e., they are sent.

When several objects or messages appear on the left-hand side of a rule, they need to synchronize in order to such a rule to be fired. Thus, a rule with several objects or messages in its left-hand side is called *synchronous*, while one involving just one object and one message in its left-hand side is called an *asynchronous* rule.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` is a particular case of a subsort declaration `C < C'`, by which all attributes, messages, and rules of the superclasses, as well as the newly defined attributes, messages and rules of the subclass characterize its structure and behavior. Multiple inheritance is supported [4].

4 Enterprise Specifications in Maude

In general, ODP systems are modeled in terms of objects. An object is a representation of an entity in the real world; it contains information and offers services. A system is therefore composed of interacting objects. In the case of the enterprise viewpoint we talk about *enterprise objects*, which model the entities defined in an enterprise specification. The use of the object paradigm is key in ODP, since it allows abstraction and encapsulation, two important properties for the specification and design of complex systems.

An enterprise specification is composed of specifications of the elements previously defined in Section 2. That is, it consists of the specification of the system's communities (set of enterprise objects), roles (identifiers of behavior), processes (sets of actions), policies (rules that govern the behavior and membership of communities to achieve an objective), and their relationships [6].

The first step towards building the enterprise specifications of a system is to identify all those elements, which can be done according to the following process:

1. Identify the communities, the roles in the communities, and the relationships among those roles.
2. Identify the enterprise objects in each community, and how they fill the roles.
3. Identify the possible actions, and the participant objects in them. Objects may participate as *actors*, *artifacts* (if they are just referenced in the action), and *resources* (artifacts essential to the action that may become unavailable or used up) [6].
4. Finally, identify the policies that rule the actions (permissions, obligations, authorizations, prohibitions), and the effects of the possible violations of those policies.

Points 1 and 2 deal with the (static) structure of the system in terms of communities, roles and their relationships. Point 3 defines the behavior of the system in terms of the possible actions allowed, and point 4 defines the rules that govern such a behavior.

Once we have identified all those entities, we are ready to model the system. For that, we propose Maude, using the following modeling approach:

1. Each **role** is modeled by a Maude class, whose members are the objects exhibiting a behavior compatible with the one identified by the role. The name of class modeling a role is the same as the role name, and the class attributes describe the properties that characterize the objects fulfilling that role. If role A specializes other role B, this is modeled by class `A` inheriting from class `B`.
2. Each **relationship among roles** is modeled by a class with the name of the relationship as its name, and whose attributes are the identifiers of the participants and the relationship's attributes.
3. **Enterprise objects** are modeled by Maude objects. In Maude, each object belongs to a class. The class of an object is obtained by composing all the Maude classes that model the different roles that the object fulfills. This is realized in Maude by multiple class inheritance.
4. **Communities** are compositions of enterprise objects, and therefore are naturally modeled by Maude's *configurations* (i.e., multisets of objects).
5. **Actions** are modeled by rewrite rules. The left-hand side and guard of a rewrite rule represent the conditions that must satisfy a particular subsystem for a rule to be triggered on it, that is, what has to happen for an action to take place; its right-hand side represents the effect of such an action on such a subsystem.
6. Finally, **policies** (both membership and behavioral) determine the form of the rewrite rules. The way to model them will depend on the kind of policy:
   – **Permissions** allow state transitions. Therefore, they will be modeled by a rule whose left-hand side determines the scenario of the permitted action and its participants, and whose right-hand side describes the effects of such an action.
   – To model **obligations** we need to differentiate whether they are *internal* or *external* ones. By internal obligations we mean those actions that the system is forced to undertake as part of its intended behavior (i.e., its scope [6]). They will be modeled as normal rules that determine the behavior of the system, perhaps restricting any other behavior by appropriate guards. However, it is difficult to impose obligations on actions that are due to external agents of the system (e.g., a borrower of a book in a library that does not return the item). In this case we shall implicitly permit the obliged actions, but introducing as well

the appropriate rules for allowing the observation of the possible violations of such obligations. Those '*watchdog*' rules will determine the appropriate corrective (penalty or incentive) actions.

– **Authorizations** will be modeled as permissions, explicitly permitting the corresponding actions. But, as for obligations, watchdog rules need to be defined for determining the system's behavior in case a violation of the authorization occurs.
– **Prohibitions** can be treated in two different ways, depending on its nature. The first way is to express them as conditional statements, using the rules' guards for explicitly banning such actions. In this way, the system will automatically prevent the prohibited action to happen. For actions whose occurrence escapes from the control of the system, the second way to deal with prohibitions is by using watchdog rules again, which detect the occurrence of the prohibited action and determine the appropriate behavior of the system in that case.

Table 1 summarizes the correspondence between the enterprise viewpoint concepts and the Maude concepts used to model them.

ODP concept	*Maude concept*
Community	Configuration
Role	Class
Relation between roles	Class
Enterprise object	Object
Action	Rewrite rule
Policies	(+*strategies*)
→ Permissions	Rewrite rules
→ Obligations	Rewrite rules
– internal	Rewrite rules
– external	Watchdog rules
→ Authorizations	Rewrite rules & watchdog rules
→ Prohibitions	Rewrite rules
– internal	Rule guards
– external	Watchdog rules

Table 1. Correspondence between the Enterprise and the Maude concepts

Once the system specifications are written using this modeling approach, what we get is a rewrite logic specification of the system, which can be executed in Maude, thus simulating its behavior. This means that, in addition to being able to formally reason about such a system, we will have a running prototype of it. Maude implements a default top-down rule-fair strategy for the execution of rewrite systems [4], but also provides the possibility of defining our own strategies for controlling the execution, which will allow us to carry out simulations, model checking, and other dynamic analysis of the system [11].

5 A Case Study

In order to illustrate our proposal we will specify in Maude a simple example, which was first used by Steen and Derrick in [16] to illustrate the use of Object-Z for the specification of the enterprise viewpoint as well. We decided to use the same example in this work with the purpose of comparing both approaches. The example is based on the regulations of the Templeman Library, at the University of Kent at Canterbury, especially those that rule the borrowing process of the Library items:

1. *Borrowing rights are given to all academic staff, and postgraduate and undergraduate students of the University.*
2. *There are prescribed periods of loan and limits on the number 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.*
3. *Items borrowed must be returned by the due date and time.*
4. *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.*
5. *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 artifacts playing a role in the library community. Note the high level at which this description is given, and the many details left unspecified. Restricting ourselves to a faithful specification of such a description will be one of the main challenges of this exercise.

5.1 The Structure of the System

Let us begin with the static aspects of this community, i.e., its structure. Following our proposed approach, we can identify here three main roles, namely **borrowers**, library **items**, and **librarians**. There are three special kinds of borrowers (academic staff, undergrats, and postgrats), and two kinds of items (books and periodicals) (see Figure 1). There is also a relationship between borrowers and items, defined by a `Loan` class. In this case, each enterprise object fulfills only one role.

The borrower role is modeled by the `Borrower` class, which has two attributes (`borrowedItems` and `fines`) for keeping record, respectively, of the number of items borrowed by a borrower and of his/her fines. Its subclasses do not need any additional attributes:

```
class Borrower | borrowedItems : Nat, fines : Money .
classes Academic Undergrat Postgrat .
subclasses Academic Undergrat Postgrat < Borrower .
```

An item can be either a book or a periodical, and its only attribute will indicate whether it is available or on loan. We model this by declaring classes Book and Periodical as subclasses of class Item, which has a single attribute status of a sort ItemStatus with values free and onloan.

```
class Item | status : ItemStatus .
classes Periodical Book .
subclasses Periodical Book < Item .
```

A library role is modeled by a class containing the information concerning deadlines and number of items that a borrower is allowed to borrow simultaneously. Such information is modeled as partial functions associating the appropriate values to each of the different actors involved, with an operator _[_] which takes a partial function and a domain element and returns its image, or a default value if it has no associated image. Thus, class Library has an attribute maxLoans, of sort PFun[Cid, Nat], that returns the maximum number of items allowed to borrow for a given class name (e.g., 24 for an academic, etc.). This function is not defined for a class name other than Academic, Undergrat, and Postgrat. Similarly, there is a loanPeriod attribute such that, given the names of a borrower class and an item class, it gives back the number of days that a borrower is allowed to hold an item (e.g., 7 days for an academic to have a periodical, etc.). Library objects also have an attribute suspendedUsers, which is a set with the identifiers of the borrower objects who are suspended.

```
class Library | maxLoans : PFun[Cid, Nat],
                loanPeriod : PFun[Tuple[Cid, Cid], Nat],
                suspendedUsers : Set[Oid] .
```

Please note that such a specification will allow us, for instance, to easily 'compose' communities with different particular details (e.g., the borrowing limits may change from a library to another), allowing them to easily co-exist. Also notice the use of the predefined sorts Oid and Cid for object identifiers and class identifiers, respectively.

Finally, we have classes Librarian and Loan. Each object of class Loan will establish a loan relationship between a borrower and an item, whose identifiers are kept in attributes borrower and item, respectively.

```
class Librarian .
class Loan | dueDate : Date, borrower : Oid, item : Oid .
```

Sort Date specifies dates. This sort can be defined in many different ways. Given the use that we are going to make of it, we can assume, for example, dates as given by integer numbers, and a calendar object that keeps the current date, which gets increased by a rewrite rule.

```
class Calendar | date : Date .
rl [new-day] :
   < O : Calendar | date : D > => < O : Calendar | date : D + 1 > .
```

We do not worry here about the frequency with which the date gets increased, the possible synchronization problems in a distributed setting, nor with any other issues related to the specification of time. See, for example, the work by Kosiuczenko and Wirsing [7] or the work by Ölveczky and Meseguer [15] on the specification of real-time systems in rewriting logic and Maude for a discussion on these issues.

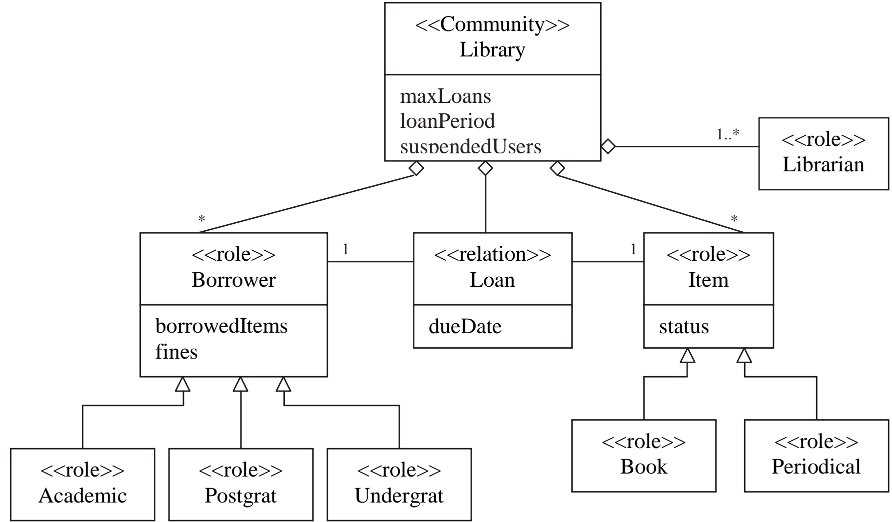

Fig. 1. Structure of the library community.

As suggested in [16], the structure of communities can be specified using UML, as a first step towards formalizing it. We show in Figure 1 the structure of the system in our example. The strong correspondence between the UML model classes and the Maude classes allows an easy translation between both models. This fact is very important, since it allows the stakeholders of the system to use a more user-friendly graphical notation like UML's class diagrams to express the system's structure, and then translate it into Maude classes. Special care should be taken here, since the semantics of UML is often weak and imprecise, as opposed to the semantics of Maude. An automatic translation of UML class diagrams into Maude, inside a more ambitious project trying to give semantics to UML, has been proposed by Toval-Álvarez and Fernández-Alemán in [17].

5.2 Actions and Policies Governing the System's Behavior

Five actions can be identified in the example: a borrower borrows an item, a borrower returns a borrowed item, a librarian fines a borrower, a fined borrower pays his/her debts, and a librarian suspends a borrower for being late in paying his/her fines. We have tried to give the specification of the system at the same level of abstraction of the

textual description of the system given above, thus leaving many details unspecified. Of course, this specification may be further refined as many times as we wish, until we get to the right level of abstraction.

The modeling of these actions will be conditioned by the policies given. The enterprise policies that can be extracted from the text are the following:

1. Any borrower is *permitted* to borrow an item if the number of his/her borrowed items is less than his/her allowance (allowances as per the text: 8 items for undergrats, etc.).
2. An `Undergrat` is *forbidden* to borrow a `Periodical` item.
3. Any borrower is *permitted* to borrow an item for a given period of time. The loan period depends on the kind of borrower and the kind of the item being borrowed (times as per the text).
4. Any borrower is *obliged* to return his/her borrowed items before their due date.
5. A *violation* of the previous rule may result in fines of certain amount by the librarian until the items are returned.
6. Any fined borrower is *obliged* to pay his/her fines to the librarian.
7. A *violation* of the previous rule may result in a suspension action to the borrower by the librarian.

Note how in these policies there are some details left unspecified, for example, *when* the actions after a policy violation are executed (expressed by "*may result in*" in policies 5 and 7), or the precise amount of the fines in policy 7. In the same sense, most policies say *what* needs to happen under the occurrence of an action, but not the reasons or circumstances that triggered that action in the first place. For instance, policy 7 permits a librarian to suspend a late-payer, but it does not specify when, or why.

5.3 The Maude Rules Specifying the System's Actions and Policies

In this section we discuss the rules specifying the actions and policies described in Section 5.2.

In the first place, borrowing a book needs the borrower object not to be suspended, and that the number of its borrowed items is smaller than its allowance. We specify such an action with the synchronous rule below, in which there are several objects involved, namely, a borrower borrowing a book, the book, a librarian, the library, and a calendar object supplying the current date.

```
crl [a-borrower-borrows-a-book] :
   < B : Borrower | borrowedItems : N >
   < I : Book | status : free >
   < L : Library | maxLoans : ML, loanPeriod : LP, suspendedUsers : OS >
   < O : Librarian | >
   < C : Calendar | date : Today >
   => < B : Borrower | borrowedItems : N + 1 >
      < I : Book | status : onloan >
      < L : Library | >
      < O : Librarian | >
```

```
      < C : Calendar | >
      < A : Loan | dueDate : Today + LP[(class(B), Book)],
                   borrower : B, item : I >
      if not B in OS and N < ML[class(B)] .
```

In Maude, as described in [4], those attributes of an object that are not relevant for an axiom do not need to be mentioned. Attributes not appearing in the right-hand side of a rule will maintain their previous values unmodified. Note as well the use of attributes `borrower` and `item` in loan objects, which make explicit that the loan relationship is between the borrower and the item specified by these attributes. Finally, the due date is obtained by adding to the current date the value assigned to the pair borrower-item in the partial function `LP` in the attribute `loanPeriod`.

In addition to the conditions required for the borrowing of a book, the borrowing of a periodical is only allowed if the borrower is not an undergrat. This can be specified as shown in the following rule.

```
crl [a-non-undergrat-borrows-a-periodical] :
   < B : Borrower | borrowedItems : N >
   < I : Periodical | status : free >
   < L : Library | maxLoans : ML, loanPeriod : LP, suspendedUsers : OS >
   < O : Librarian | >
   < C : Calendar | date : Today >
   => < B : Borrower | borrowedItems : N + 1 >
      < I : Periodical | status : onloan >
      < L : Library | >
      < O : Librarian | >
      < C : Calendar | >
      < A : Loan | dueDate : Today + LP[(class(B), Book)],
                   borrower : B, item : I >
      if not B in OS and class(B) =/= Undergrat and N < ML[class(B)] .
```

Some of the policies are implicit in the rules, that is, they do not need to be explicitly specified in the guards. Note the different ways of specifying them:

– A borrower cannot borrow a book if he/she is not allowed (policy 7) or the book is not free (common sense / overspecification). This is specified by given the required values in the left-hand side of the rule.
– No further items can be borrowed if the number of borrowed items is already greater or equal than his/her allowance (policy 1). This is guaranteed by the rules' guards.
– The prohibition for an undergrat to borrow periodicals (policy 2) is specified by the condition of the rule `a-non-undergrat-borrows-a-periodical`. However, if we give separate rules dealing with each of the cases, such policy would be specified just by not giving a rule for such a case.

The return of an item may be specified by the rule below. We can see how the `loan` object disappears in the right-hand side of the rule.

```
rl [return] :
  < B : Borrower | borrowedItems : N >
```

```
  < I : Item | >
  < A : Loan | borrower : B, item : I >
  < O : Librarian | >
  < L : Library | >
  => < B : Borrower | borrowedItems : N - 1 >
     < I : Item | status : free >
     < O : Librarian | >
     < L : Library | > .
```

The description of the system says nothing about when or how late borrowers are fined. We have included a Maude rule for librarians to fine borrowers, but in it, the amount of the fine is left unspecified, as well as the way of calculating that amount, the precise moment in which the librarian decides to fine the borrower, or how often borrowers are fined for late returns (maybe only once when they return the book, or everyday for a fixed daily rate, ...). Moreover, the text says that borrowers returning late an item become "*liable to a charge*", which may even allow librarians not to fine them at all.

```
crl [fine] :
  < B : Borrower | fines : M >
  < I : Item | status : onloan >
  < A : Loan | dueDate : D, borrower : B, item : I >
  < O : Librarian | >
  < L : Library | >
  < C : Calendar | date : Today >
  => < B : Borrower | fines : M + Amount >
     < I : Item | >
     < A : Loan | >
     < O : Librarian | >
     < L : Library | >
     < C : Calendar | >
     if D < Today .
```

Late payers may be suspended by the librarian, although nothing is said about the reasons that may make the librarian take such a decision (apart from "*failure to pay charges*"). This can be modeled by the following rewrite rule.

```
crl [suspend] :
   < O : Librarian | >
   < L : Library | suspendedUsers : BS >
   < B : Borrower | fines : M >
   => < O : Librarian | >
      < L : Library | suspendedUsers : B BS >
      < B : Borrower | >
      if M > 0 .
```

Finally, the payment of charges may be modeled as follows.

```
crl [pay-charges] :
   < B : Borrower | fines : M >
```

```
< L : Library | >
< O : Librarian | >
=> < B : Borrower | fines : M - Amount >
   < L : Library | >
   < O : Librarian | >
   if M > 0 and 0 < Amount and Amount <= M .
```

The amount to pay has been left unspecified, since nothing is said in the description of the system about the way borrowers pay their debts: either all due charges need to be paid at once, or fractions are allowed. With the previous rule we cover all cases where the amount is between zero and the total amount due. Note that there are many other details left unspecified. For example, nothing is said about the restoration of the borrowing rights of a borrower, and therefore no rule is given for that action.

It is important to notice at this level of abstraction that the responsibilities for each of the actions are not made explicit. If required, such responsibilities will become explicit in successive steps of refinement, in which the rules specifying the actions can be decomposed into several (either consecutive or concurrent) sub-actions. For example, borrowing an item, which is at this level modeled by rules `a-borrower-borrows-a-book` and `a-non-undergrat-borrows-a-periodical`, could be decomposed as follows: a borrower first request the borrowing of a book or periodical to a librarian, which then checks for the availability and rights of the borrower with the library, etc. Note that at this level of abstraction it is made explicit the fact that it is the borrower who initiates the borrowing action.

Another interesting issue worth pointing out is about the use of messages as a communication mechanism between objects, which is also provided in Maude. Of course, there are other alternatives for specifying systems such as the one in the example using Maude. For instance, we could have modeled ODP actions by Maude messages and ODP policies by Maude rules. In our current approach ODP actions are modeled by Maude rules, while ODP policies are modeled by rule guards that determine when the rules are enabled. This is a more abstract and general approach than using messages. First, it allows to deal with each kind of policy in a different way, to define the so-called *watchdog* rules that determine the behavior of the system upon the occurrence of a policy violation, and to use Maude's *strategies* for controlling the execution or model-checking the system behavior based on the different occurrence of the actions. And second, Maude messages naturally correspond to ODP messages, that model interactions between objects—but in the computational viewpoint, where they naturally belong (in ODP, messages are a computational viewpoint concept), more than in the enterprise viewpoint. We believe that rules are more appropriate for modeling actions in the enterprise viewpoint than messages between objects.

6 Related Work

Formal description techniques are being extensively employed in ODP and have proved valuable in supporting the precise definition of reference model concepts [3]. Among all those works, we will focus here on two kinds of proposals: those that use rewriting

logic for specifying some of the ODP viewpoints, and those that specifically deal with the enterprise viewpoint.

In the first group, Najm and Stefani use rewriting logic to formalize the computational model of RM-ODP [13, 14]. In [13], a formal operational semantics of the ODP computational model is presented, which is extended in [14] to deal with reflection and Quality of Service (QoS) contracts using failures. In [1], Bernardeschi *et al* use the actors model to specify the computational model, defining a transformation between the information and computational models that are semantically consistent. This can be embedded into Maude by using the natural inclusion of the actor model into Maude [12].

With regard to the proposals that try to formalize the enterprise viewpoint, there is an interesting proposal for using Object-Z as a formal notation for pinning down the precise semantics of entreprise specifications [16]. The work by Steen and Derrick uses UML for describing the structure of the enterprise specification, and combines it with a simple language using predicate logic for specifying enterprise policies. A formal translation process (with future tool support) is then defined to express the (informal) specifications obtained into the formal object-oriented specification language Object-Z. As discussed in Section 5.1, the use of UML seems to us a right choice for describing the structure of the community despite its ambiguity and lack of formal underpinnings. However, the use of Object-Z for specifying the enterprise policies may present some shortcomings:

- Actions are assigned to just one actor, and included as operations in the actor's definition class. How to deal with actions in which there is more than a principal actor (e.g., in the case of synchronous actions)? In our approach actions are rules, and therefore first-class citizens.
- The treatment of policy violations is not homogeneous with the rest of the policies. Violations are not (and cannot be) specified within the Object-Z framework, but at the meta-level [16], which is not easily accessible from the Object-Z specifications.
- Temporal logic is used to express the invariants, but the fragment of temporal logic included in Object-Z is too limited, as pointed out in [16].
- The use of Object-Z forces most of the unspecified details in the 'textual' specifications to be (over)specified, since full specifications are needed. This forces the specifiers to make too many assumptions, incurring into over-specifications in many cases.
- Another disadvantage of the use of Object-Z appears when modeling enterprise roles by Object-Z classes, which is the natural way of doing it. This has the initial advantages that information about each role can be encapsulated, and that roles can be composed. However, it has the disadvantage that roles are thereby associated with fixed classes of objects, so that an object cannot change its role during its lifetime. In some applications, models which assume fixed roles may be adequate; but in others, there may be a need to represent objects which play different roles at different times, as it happens for instance in dynamically configurable networks. Again, this is not an issue in Maude, since the class of an object can be changed in a rule.
- Other notations (such as Z, LOTOS, or CSP) are used in other viewpoints, because Object-Z does not deal with all aspects. A common way of dealing with consistency

between specifications written in different notations is by translating them into one single notation. For instance, in [2] the authors propose the translation of LOTOS into Object-Z. However, many important aspects of the specification are usually lost in these translations, since the underlying logic of Object-Z is not expressive enough. We think that Maude can greatly help in this point, and is something that we want to explore further. Rewriting logic is such that faithful translations from other FTDs into Maude can be obtained [11].

7 Concluding Remarks

Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In this paper, we have explored the possibility of using Maude for specifying the enterprise viewpoint, and we are now in a position to formally reason about the specifications produced.

Once we make sure that the specifications of a particular viewpoint satisfy certain properties, we need to address two additional issues, namely, the composition and the consistency checking of specifications of different viewpoints, so that we get a specification of the complete system. By establishing the consistency of different viewpoints we simply mean that the specifications of the different viewpoints do not impose contradictory requirements.

Different notations have been proposed for specifying the different viewpoints, and some of them may be even used for specifying several or even all of them. We plan to study the adequacy of Maude for being used in the specification of all of them. However, there is a general belief that no formal method applies well to all problem domains. It is not only about being expressive enough, but on the fact that each formalism is more appropriate than others for a particular viewpoint. One may prefer, for example, Object-Z for the information viewpoint and LOTOS or SDL for the computational viewpoint.

Checking the consistency of the specifications of different viewpoints is a difficult task, and it is even harder checking it if such viewpoints are specified in different formalisms. Thus, we have two options: either we write all viewpoints specifications in Maude, or use different formalisms for the different viewpoints and then translate them into Maude.

It has been shown that rewriting logic and Maude has very good properties as a logical framework, in which representing many different languages and logics, and as a semantic framework, in which giving semantics to them [11]. Formalisms such as CCS, LOTOS, SDL, and many others can be represented in rewriting logic, thus allowing the possibility of bringing very different models under a common semantic framework. Such a framework makes much easier to achieve the integration and interoperation of different models and languages in a rigorous way. Thus, Maude seems to be a promising option as a unifying framework for the specification of RM-ODP viewpoints in which consistency checks can be rigorously studied.

Another interesting topic of research is the use of the reflective capabilities of rewriting logic and Maude for specifying and reasoning about different system properties, such as QoS (as in Najm's works) or reliability, or about the dynamic reconfiguration of systems.

Acknowledgments

We are very thankful to Ambrosio Toval and to the anonymous referees for their comments on a previous version of this paper.

References

1. C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and consistent semantics for ODP viewpoints. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, Canterbury, 1997. Chapman & Hall.
2. E. A. Boiten, H. Bowman, J. Derrick, P. F. Linington, and M. W. Steen. Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537, August 2000. Available at `http://www.cs.ukc.ac.uk/pubs/2000/1026`.
3. H. Bowman, J. Derrick, P. F. Linington, and M. W. A. Steen. FDTs for ODP. *Computer Standards and Interfaces*, 17:457–479, September 1995. Available at `http://www.cs.ukc.ac.uk/pubs/1995/178`.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.
5. ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing. Rec. ISO/IEC 10746-1 to 10746-4, ITU-T X.901 to X.904, ISO/ITU-T, 1997.
6. ISO/IEC. RM-ODP Enterprise Language. FCD Rec. ISO/IEC 15414, ITU-T X.911, ISO/ITU-T, 2000.
7. P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2-3):225–246, 1997.
8. P. F. Linington. RM-ODP: The architecture. In K. Milosevic and L. Armstrong, editors, *Open Distributed Processing II*, pages 15–33. Chapman & Hall, Feb. 1995.
9. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Comput. Sci.*, 96:73–155, 1992.
10. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.
11. J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. F. Smith and C. L. Talcott, editors, *Proc. of FMOODS'2000*, pages 89–117, Stanford, CA, Sept. 2000. Kluwer Academic Publishers.
12. J. Meseguer and C. L. Talcott. A partial order event model for concurrent objects. In *Proceedings of CONCUR'99: Concurency Theory*, volume 1664 of *LNCS*, pages 415–430. Springer-Verlag, 1999.
13. E. Najm and J.-B. Stefani. A formal operational semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.
14. E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, Canterbury, 1997. Chapman & Hall.
15. P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 361–383. Elsevier, 2000.
16. M. W. Steen and J. Derrick. ODP Enterprise Viewpoint Specification. *Computer Standards and Interfaces*, 22:165–189, September 2000. Available at `http://www.cs.ukc.ac.uk/pubs/2000/1122`.

17. A. Toval-Álvarez and J. L. Fernández-Alemán. Formally modeling UML and its evolution: A holistic approach. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*. Kluwer Academic Publishers, 2000.

Appendix 1: The Complete Library Specification in Maude

This appendix contains the complete enterprise specification of the Library in Maude.

```
*************** DEFAULT ******************
*** file: ~/Maude/Library/default.fm

(fmod DEFAULT[Y :: TRIV] is
  sort Default[Y] .
  subsort Elt.Y < Default[Y] .
  op null : -> Default[Y] .
 endfm)

*************** SET ******************
*** file: ~/Maude/Library/set.fm

(fmod SET[X :: TRIV] is
  sort Set[X] .
  subsort Elt.X < Set[X] .

  op mt : -> Set[X] .
  op __ : Set[X] Set[X] -> Set[X] [assoc comm id: mt] .
  op _in_ : Elt.X Set[X] -> Bool .

  vars E E' : Elt.X .
  var S : Set[X] .

  eq E E = E .
  eq E in mt = false .
  eq E in (E' S) = E == E' or (E in S) .
 endfm)

*************** MAP ******************
*** file: ~/Maude/Library/map.fm

(fth FUNCTION is
  sorts Domain Codomain .
  op f : Domain -> Codomain .
 endfth)

(view Domain from TRIV to FUNCTION is
  sort Elt to Domain .
 endv)

(view Codomain from TRIV to FUNCTION is
```

```
   sort Elt to Codomain .
  endv)

in ~/Maude/Library/set.fm

 (fmod MAP[F :: FUNCTION] is
   protecting (SET[Domain] * (op __ to _;_))[F] .
   protecting SET[Codomain][F] .

   op map : Set[Domain][F] -> Set[Codomain][F] .

   var A : Domain.F .
   var S : Set[Domain][F] .

   eq map(mt) = mt .
   eq map(A ; S) = f(A) map(S) .
  endfm)

 *************** PFUN *****************
 *** file: ~/Maude/Library/pfun.fm

 in ~/Maude/Library/map.fm

 (fmod PFUN[U :: TRIV, V :: TRIV] is
   protecting BOOL .
   protecting DEFAULT[V] .

   sort Pair[U, V] .
   op <_;_> : Elt.U Default[V] -> Pair[U, V] .
   op 1st : Pair[U, V] -> Elt.U .
   op 2nd : Pair[U, V] -> Default[V] .

   var A : Elt.U .
   var B : Default[V] .

   eq 1st(< A ; B >) = A .
   eq 2nd(< A ; B >) = B .

   sorts PairSet[U, V] NePairSet[U, V] .
   subsorts Pair[U, V] < NePairSet[U, V] < PairSet[U, V] .

   op mt : -> PairSet[U, V] .
   op __ : PairSet[U, V] PairSet[U, V] -> PairSet[U, V]
         [assoc comm id: (mt).PairSet`[U`,V`]] .
   op __ : NePairSet[U, V] NePairSet[U, V] -> NePairSet[U, V]
         [assoc comm id: (mt).PairSet`[U`,V`]] .
   op _in_ : Pair[U, V] PairSet[U, V] -> Bool .

   vars E E' : Pair[U, V] .
   var S : PairSet[U, V] .
```

```
  eq E E = E .
  eq E in (mt).PairSet`[U`,V`] = false .
  eq E in (E' S) = (E == E') or (E in S) .

  protecting SET[U] .
  protecting SET[V] .

  sort PFun[U, V] .
  subsorts Pair[U, V] < PFun[U, V] < PairSet[U, V] .

  op mt : -> PFun[U, V] .
  op _`[_`] : PFun[U, V] Elt.U -> Default[V] .
  op _`[_->_`] : PFun[U, V] Elt.U Default[V] -> PFun[U, V] .

  vars C : Default[V] .
  var F : PFun[U, V] .

  op dom : PairSet[U, V] -> Set[U] .  *** domain
  eq dom((mt).PairSet`[U`,V`]) = (mt).Set`[U`] .
  eq dom(< A ; B > S) = A dom(S) .

  op im : PairSet[U, V] -> Set[V] .   *** image
  eq im((mt).PairSet`[U`,V`]) = (mt).Set`[V`] .
  eq im(< A ; B > S) = B im(S) .

  cmb < A ; B > F : PFun[U, V] if not(A in dom(F)) .
  eq (< A ; B > F)[ A ] = B .
  ceq F [ A ] = null if not(A in dom(F)) .
  eq (< A ; B > F)[ A -> C ] = < A ; C > F .
  ceq F [ A -> C ] = < A ; C > F if not(A in dom(F)) .
 endfm)

*************** MAIN MODULE ******************
*** file: library.fm

*** Note that:
*** - The spec below requires strategies to be executed, since there
***   are rules with variables in their rhs or conditions not apearing
***   in their lhs.
*** - It assumes a function class : Object -> Cid returning the class
***   of an object which is not currently available in Maude.

in ~/Maude/Library/pfun.fm

******************  CALENDAR *************************
(omod Calendar is
  pr MACHINE-INT .
  class Calendar | date : MachineInt .
  var O : Oid .
```

```
   var D : MachineInt .
   rl [new-day] :
     < O : Calendar | date : D > => < O : Calendar | date : D + 1 > .
  endom)

 *****************  VIEWS ************************
 (view Oid from TRIV to OID is
   sort Elt to Oid .
  endv)

 (view Cid from TRIV to OID is
   sort Elt to Cid .
  endv)

 (view MachineInt from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
  endv)

 (view Tuple`[Cid`,Cid`] from TRIV to TUPLE(2)[Cid, Cid] is
   sort Elt to Tuple[Cid, Cid] .
  endv)

*****************  LIBRARY ************************

 (omod LIBRARY is
   pr MACHINE-INT .
   pr SET[Oid] .
   pr CALENDAR .
   pr PFUN[Cid, MachineInt] .
   pr PFUN[Tuple`[Cid`,Cid`], MachineInt] .

   sort ItemStatus .
   ops free onloan : -> ItemStatus .

   class Borrower | borrowedItems : MachineInt, fines : MachineInt .
   class Academic | .
   class Undergrat | .
   class Postgrat | .
   subclasses Academic Undergrat Postgrat < Borrower .

   class Item | status : ItemStatus .
   class Periodical | .
   class Book | .
   subclasses Periodical Book < Item .

   class Library | maxLoans : PFun[Cid, MachineInt],
                   loanPeriod : PFun[Tuple`[Cid`,Cid`], MachineInt],
                   suspendedUsers : Set[Oid] .

   class Librarian | .
```

```
class Loan | dueDate : MachineInt, borrower : Oid, item : Oid .

subsort Qid < Oid .

vars B I A L O C : Oid .
vars N M Amount Today : MachineInt .
var ML : PFun[Cid, MachineInt] .
var LP : PFun[Tuple`[Cid`,Cid`], MachineInt] .
vars OS BS : Set[Oid] .

crl [a-borrower-borrows-a-book] :
  < B : Borrower | borrowedItems : N >
  < I : Book | status : free >
  < L : Library | maxLoans : ML, loanPeriod : LP, suspendedUsers : OS >
  < O : Librarian | >
  < C : Calendar | date : Today >
  => < B : Borrower | borrowedItems : N + 1 >
     < I : Book | status : onloan >
     < L : Library | >
     < O : Librarian | >
     < C : Calendar | >
     < A : Loan | dueDate : Today + LP[(class(B), Book)],
                  borrower : B, item : I >
     if not B in OS and N < ML[class(B)] .

crl [a-non-undergrat-borrows-a-periodical] :
  < B : Borrower | borrowedItems : N >
  < I : Periodical | status : free >
  < L : Library | maxLoans : ML, loanPeriod : LP, suspendedUsers : OS >
  < O : Librarian | >
  < C : Calendar | date : Today >
  => < B : Borrower | borrowedItems : N + 1 >
     < I : Periodical | status : onloan >
     < L : Library | >
     < O : Librarian | >
     < C : Calendar | >
     < A : Loan | dueDate : Today + LP[(class(B), Book)],
                  borrower : B, item : I >
     if not B in OS and class(B) =/= Undergrat and N < ML[class(B)] .

rl [return] :
  < B : Borrower | borrowedItems : N >
  < I : Item | >
  < A : Loan | borrower : B, item : I >
  < O : Librarian | >
  < L : Library | >
  => < B : Borrower | borrowedItems : N - 1 >
     < I : Item | status : free >
     < O : Librarian | >
     < L : Library | > .
```

```
  crl [fine] :
    < B : Borrower | fines : M >
    < I : Item | status : onloan >
    < A : Loan | dueDate : D, borrower : B, item : I >
    < O : Librarian | >
    < L : Library | >
    < C : Calendar | date : Today >
    => < B : Borrower | fines : M + Amount >
       < I : Item | >
       < A : Loan | >
       < O : Librarian | >
       < L : Library | >
       < C : Calendar | >
    if D < Today .

  crl [suspend] :
    < O : Librarian | >
    < L : Library | suspendedUsers : BS >
    < B : Borrower | fines : M >
    => < O : Librarian | >
       < L : Library | suspendedUsers : B BS >
       < B : Borrower | >
       if M > 0 .

  rl [pay-charges] :
    < B : Borrower | fines : M >
    < L : Library | >
    < O : Librarian | >
    => < B : Borrower | fines : M - Amount >
       < L : Library | >
       < O : Librarian | >
       if M > 0 and 0 < Amount and Amount <= M .
endom)
```