Using UML and Maude for Writing and Reasoning about ODP Policies

Francisco Durán, Javier Herrador, and Antonio Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{duran,av}@lcc.uma.es

Abstract

In this paper we present a graphical UML-based notation for writing ODP actions and policies, which can be directly mapped to Maude specifications. Our approach may introduce important benefits to the (usually ambiguous) UML specifications, such as formal support, provision for rigorous specifications, and easy access to Maude's toolkit. We have developed a tool for automating the translation process and for giving access to Maude's analysis tools. In this way we try to bridge the current gap between graphical and formal notations, by providing an easy-to-use environment for modeling enterprise business systems with UML, but still with formal support.

1. Introduction

So far, most notations and tools for capturing and modeling business requirements tend to be either graphical and easy-to-use (but informal and with vague semantics), or formal and with tool support for reasoning about the specifications produced (although unappealing and hard-to-use for the average software engineer). Graphical notations are intuitive and easy to learn and to use, and do not require users to have a deep and specialized knowledge of complex concepts, formalisms, and mechanisms. For instance, the adoption and widespread use of UML for describing and modeling business systems has greatly helped involve stakeholders of diverse backgrounds in the system specification process. However, these notations usually have a weak and imprecise semantics, which hinders the formal analysis processes required in any "real" engineering discipline. Even worse, their apparent simplicity may artificially hide the real complexity of the systems being modeled.

On the contrary, formal notations provide precise and unambiguous system specifications. More importantly, formal notations also allow the rigorous analysis of the systems, with tools for quick-prototyping, model checking, or theorem proving.

These two worlds (graphical notations and formal methods) have usually lived apart. This paper tries to provide a bridge between them in the context of the ODP enterprise viewpoint, focusing on the specification of actions and the behavioral policies conditioning them. In particular, this proposal presents a tool for the graphical specification of business systems, based on the corresponding formal specifications in Maude, aiming at getting all the benefits that they individually provide.

For modeling business requirements and systems we will use the concepts provided by the RM-ODP enterprise viewpoint [9]. 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, or the technology used to implement the system.

As formal notation for specifying the ODP enterprise concepts we will use Maude [4], an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In a previous work [7] we showed a simple and natural way of modeling the enterprise viewpoint concepts. In addition to the nice properties of the Maude specifications obtained following that approach, the use of Maude provides additional advantages. The fact that rewriting logic specifications are executable, allows us to apply a flexible range of increasingly stronger formal analysis methods and tools, such as runtime verification, model checking, narrowing analysis, or theorem proving. Maude offers a comprehensive toolkit for the analysis of specifications, including an inductive theorem prover; an LTL model checker; tools to check the Church-Rosser property, coherence, and termination; tools to perform Knuth-Bendix and coherence completion; and a tool to specify, analyze and model check real-time specifications.

However, the use of Maude and its toolkit—as it happens with most formal notations—is not easy for most system analysts and developers, which demand simpler, more appealing, and more user-friendly notations for handling business systems requirements and specifications. In this respect, the contribution we present in this paper provides a graphical way for specifying the structure and behavior of enterprise systems, with a direct translation to Maude. Our goal is to provide users with a "friendly" UML-based environment in which the system specifications can be written and automatically translated into the corresponding Maude (formal) specifications. Then, the Maude system and its analysis tools may be accessed from the UML environment, freeing the system analyst from most formal technicalities.

The structure of this document is as follows. First, Sections 2 and 3 serve as brief introductions to the ODP enterprise viewpoint and Maude, respectively. Then, Section 4 describes our proposal to write enterprise specifications in UML, and Section 5 discusses how to map these UML specifications into their corresponding Maude specifications. Section 6 is dedicated to a small example that illustrates our approach, while Section 7 briefly describes the tool we have built for the translation of the UML specifications into Maude and vice-versa. Finally, Section 8 draws some conclusions and describes some future research activities.

2. The Enterprise Viewpoint

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. In particular, the Reference Model of Open Distributed Processing (RM-ODP) framework [8] provides five generic and complementary viewpoints on the system and its environment: *enterprise*, *information*, *computational*, *engineering* and *technology* viewpoints. They enable different abstraction viewpoints, allowing participants to observe a system from different suitable perspectives [10].

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, or the technology used to implement the system.

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 [9]. An enterprise specification describes the behavior assumed by those who interact with the ODP system, explicitly including those aspects of the environment that influence its behavior—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 behavior, and that has been formed for a particular objective.

The scope of the system is defined in terms of its intended behavior, 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 behavior by means of (partially ordered) sets of *actions*, which are related to achieving some particular subobjective within the community. Finally, policies are the rules that constrain the behavior and membership of communities in order to make them achieve their objectives. A policy can be expressed as an obligation, an authorization, a permission, or a prohibition. Actions contrary to rules are known as violations.

In general, ODP systems are modeled in terms of objects. An object is a model of an entity; 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.

Summing up, an enterprise specification is composed of specifications of the elements previously mentioned, i.e. the system's communities (sets of enterprise objects), roles (identifiers of behavior), processes (sets of actions leading to an objective), policies (rules that govern the behavior and membership of communities to achieve an objective), and their relationships [9].

3. Rewriting Logic and Maude

Maude [4] is a high-level language and high-performance interpreter and compiler that supports equational and rewriting logic specification and programming of systems. The instance of rewriting logic for Maude is parameterized by membership equational logic [13], although its underlying equational logic can also be unsorted, many-sorted, or order-sorted equational logic. 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.

Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Conditional equations (resp., membership assertions) are written as "$t = t' \ if\, P$" (resp., "$t : S \ if\, P$"), stating the equality of terms t and t' (resp., that a given term t has a sort S) if a certain boolean condition P holds[1].

Rewriting logic [12] 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 sorts (types) and operations, and E is a set of (conditional) equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite *rules* of the form $t \rightarrow t'$, where t and t' are Σ-terms. These rules 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 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 `|` a_1`:`S_1`, ...,` a_n`:`S_n, where C is the name of the class, the a_i are attribute identifiers, and the S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form `<` O `:` C `|` 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 number of different ways, including message passing.

In a concurrent object-oriented system the concurrent state, which is called a *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 of objects and messages. The general form of such rewrite rules is

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

where r is the rule label, $M_1 \ldots M_m$ and $M'_1 \ldots M'_q$ are messages, $O_1 \ldots O_n$ and $Q_1 \ldots Q_p$ are object identifiers, $C_1 \ldots C_n$, $C'_{i_1} \ldots C'_{i_k}$, and $C''_1 \ldots C''_p$ are classes, $i_1 \ldots i_k$ is a subset of $1 \ldots n$, and *Cond* is a boolean condition (the rule's 'guard'). The result of applying such a rule is that: (a) messages $M_1 \ldots M_m$ disappear, i.e. they are consumed; (b) the state, and possibly the classes of objects $O_{i_1} \ldots O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1 \ldots Q_p$ are created; and (e) new messages $M'_1 \ldots M'_q$ are created, i.e. they are sent. Rule guards can be omitted if not needed.

For instance, the following Maude definitions specify a class `Account` with an attribute `balance` of sort integer, a message `withdraw` with an object identifier and an integer as arguments, and two rules describing the behavior of the objects belonging to this class. The rule `debit` specifies a local transition of the system when there is an object `A` of class `Account` that receives a `withdraw` message with an amount smaller or equal than the balance of `A`; as a result of such a rule, the message is consumed, and the balance of the account is modified. The rule `transfer` models the effect of receiving a money transfer message.

```
class Account | balance : Int .
msg withdraw : Oid Int -> Msg .
msg transfer : Oid Oid Int -> Msg .
crl [debit] :
  withdraw(A, M)
  < A : Account | balance : Bal >
  => < A : Account | balance : Bal - M >
  if M <= Bal .
crl [transfer] :
  transfer(A, B, M)
  < A : Account | balance : Bal >
  < B : Account | balance : Bal' >
  => < A : Account | balance : Bal - M >
     < B : Account | balance : Bal' + M >
  if M <= Bal .
```

When several objects or messages appear in the left-hand side of a rule, they need to synchronize in order for such a rule to be fired. These rules are called *synchronous*, while rules involving just one object and one message in their left-hand sides are called *asynchronous* rules.

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 [14].

4. Modeling Enterprise Specifications in UML

Different authors have proposed the use of UML for modeling the enterprise viewpoint concepts [1, 2, 11, 16].

At the structural level (e.g. when defining the communities, the roles, and the relationships among them) UML proves itself to be expressive enough, offering a natural way for modeling enterprise concepts at this level—which is the

[1]Membership equational logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains.

approach followed by most authors. However, at the behavioral level the situation is not so bright. Typical UML diagrams for modeling behavior (such as use case, sequence, and collaboration diagrams) prove to be inadequate and insufficiently expressive for modeling ODP actions and policies (cf. [16]).

In this Section we will describe our proposal for modeling in UML the concepts described in Section 2, which constitute the enterprise specification of a system.

4.1. Structural Concepts

For modeling the structural concepts we will follow the approach commonly used by most authors (see, e.g., [1, 2, 11, 16]). Each **role** will be modeled by a class, whose members are the objects exhibiting a behavior compatible with the one identified by the role. The name of the class modeling a role is the same as the role name, and the class attributes describe the properties that characterize the objects fulfilling such a role. The fact that a role *A* specializes other role *B* is modeled by class *A* inheriting from class *B*.

Enterprise objects will be modeled by UML objects. Every object will always belong to a class, which may be changed during its lifetime. The class of an object is obtained by composing all the classes that model the different roles that the object fulfills, which may be realized by multiple class inheritance.

A **community** is a composition of enterprise objects, and therefore it can be modeled by a UML subsystem. However, a community may also be expressed as a composite object when considered at a more abstract level of detail and, dually, an enterprise object may itself be refined as a community at a more concrete level. Thus, when abstracted as an enterprise object, a community will be modeled by an object (belonging to some class).

A **relationship among roles** establishes a semantic connection among them. The concept of role relationship is not explicitly defined in the Enterprise Language Standard, although the concepts defined in ISO's General Relationship Model can be used here. UML relationships (despite their vague semantics) will be used for modelling relationships. Association classes [15] will be used in case of relationships with attributes.

Three stereotypes will identify enterprise structural concepts. Stereotypes ≪role≫ and ≪relationship≫ associated to UML classes will model roles and relationships, respectively. Stereotype ≪community≫ associated to a UML subsystem will model a community.

For expressing *membership* policies, which constraint the structure of the community and the assignment of enterprise objects to roles, we will also use the standard mechanisms provided by UML, namely constraints attached to the modeling elements, and relationships' multiplicity.

4.2. Behavioral Concepts

Most authors that have tried to model the enterprise viewpoint behavioral concepts using the standard UML diagrams and mechanisms for modeling behavior have found serious difficulties. For instance, Steen and Derrick [16] propose the use of use cases for representing enterprise actions—that in theory could be further refined into interaction (sequence or collaboration) diagrams. But when it comes to expressing behavioral policies, the main problem is that the interaction model of UML is based on message exchange between objects, whereas interactions in the enterprise viewpoint can be seen as pieces of shared behavior. As policies constrain the behavior of roles, OCL could be another alternative. However, OCL is not expressive enough for that purposes either (e.g., it does not provide powersets or timing constraints, uses the UML model of interaction, and lacks a formal semantics). As a consequence, different authors have proposed different proprietary languages for expressing ODP policies, usually with formal support (e.g. Object-Z) but with no graphical syntax—hence losing one of the advantages of using UML.

We propose modeling the *actions* by specifying transitions between object diagrams, with *behavioral* policies shaping the form of such transitions. These transitions may be conditional, where the conditions are given as expressions in a notation close to OCL. We may see these transitions as rewrite rules, where the source of the transition—left-hand side of the rule—and its guard express the conditions that must be satisfied by a particular set of enterprise objects for such a transition to take place on it, that is, what has to happen for an action to take place. The target of the transition—its right-hand side—represents the effect of such an action on such a subsystem. Policies (both membership and behavioral) determine the form of the transitions, stating the conditions for the action to happen—either by restricting the pattern of the source of the transition, or by explicitly stating a condition with an `if` guard—or its effects—by shaping the target of the transition.

ODP actions and behavioral policies can then be modeled as (possibly conditional) transitions between object diagrams. The source and target of a transition are depicted as packages, containing the object diagrams of the corresponding subsystems, linked by a dependency relationship. The dependency relationship is labeled with the name of the transition, and it may have associated a note with the condition that describes its guard. Figure 1 shows an example of a simple transition. The left diagram (labelled Before in the picture) represents the conditions that need to be in place for the action to be fired, while the right one (labelled After) represents the effects of such an action on the system.

The use of "before" and "after" instance diagrams is a well known technique for writing pre- and postconditions.

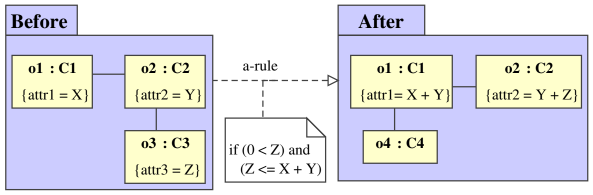

Figure 1. Object diagrams modeling business rules.

These instance diagrams are referred to as *snapshots* in Catalysis [5], and are also used by, e.g., Cheesman and Daniels in [3]. In these cases, snapshots are used to help clarify state changes and write the pre- and postconditions for the involved methods. The "before" state helps to define the precondition, and the "after" state helps to define the postcondition of an action. In our case, we are interested in using such representations, not just as a help, but as models of the actions, using them for generating the corresponding Maude specification in an automatic and transparent way.

Note that we have also added a "guard" to the pair. Although the action's condition could be expressed as a restriction in the Before snapshot, we have found useful having this explicit condition in the representation of ODP policies, and it simplifies the generation of the corresponding Maude specification. The transformation consists in mapping each of the Before-After transitions into a Maude rewrite rule, which we keep straightforward by restricting the form of the UML object diagrams used. In fact, our starting point is the use of Maude in the formalization of the ODP enterprise concepts (see [7]), being the present work an attempt to offer a graphical notation to support it. Maude rewrite rules provide a semantics for the Before-After transitions, and therefore, given their close relationship, we will refer to them as "UML rules", or "rules" for short.

The way in which the different policies that govern the behavior of a system are modeled will depend on the kind of policy:

– *Permissions* allow state transitions. Therefore, a permission is expressed by a rule whose Before object diagram and guard determine the scenario of the permitted action(s) and their participants, while its After diagram describes the effects of such action(s).

– To model *obligations* we need to differentiate between *internal* and *external* ones. By internal obligations we mean those actions that the system is forced to undertake as part of its intended behavior. These actions will be modeled as paired object diagrams (rules) that determine the behavior of the system, perhaps restricting any other behavior with appropriate guards. However, it is difficult to impose obligations on actions that are due to external agents of the system (e.g. a borrower that does not return a book). 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. 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' Before object diagram and 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, if possible.

Note that an action may be expressed by more than one UML rule, and be controlled by different policies. Therefore, a policy may be modeled by more than one UML rule, each of which may itself model more than one policy. Likewise, a policy or a collection of policies may apply to more than one action, which can be modeled by individually applying those policies to each of the paired object diagrams modeling such actions, or by characterizing the actions and then applying the policies to such a characterization.

An interesting issue worth pointing out here is the use of UML paired object diagrams for modeling both actions and policies. Of course, there are other alternatives for specifying business systems and business rules. For instance, we could have modeled ODP actions by messages (following the UML standard interaction model), and ODP policies by UML rules. In general, object-oriented modeling approaches may be perceived to require using message-oriented communication models, while the enterprise viewpoint (and RM-ODP in general) does not require so. In our approach both ODP actions and policies are modeled by paired object diagrams, with guards that determine when the actions are enabled—and thus can happen. We think that 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 second, UML messages naturally correspond to ODP messages, that model interactions between objects—but in the computational viewpoint, where they naturally belong (in ODP, a messages is a computational viewpoint concept).

In addition, it ought to be emphasized how the use of object diagrams (representing sets of objects) and our UML rules (paired object diagrams that specify collective behavior) allows a natural representation of the collective state and collective behavior of a system (that is, state and behavior not owned by a specific object), in contrast to other (object-oriented) modeling approaches in which each action needs to be assigned to just one actor, and where there is no explicit representation of the collective state.

The use of UML rules for modeling both actions and behavioral policies greatly facilitates the UML modeling of these two enterprise concepts, unifying the notation and simplifying the modeling elements and mechanisms used. Furthermore, there is a clear and direct correspondence between the UML and Maude representation of ODP actions and policies.

5. Translating the UML Enterprise Specifications into Maude

In this Section we will discuss how the UML enterprise specifications can be translated into Maude (formal) enterprise specifications, and vice-versa. A detailed description of the use of Maude for modeling enterprise concepts can be found in [7].

Although possible (see, e.g., the work by Toval-Álvarez and Fernández-Alemán in [17]), we do not intend getting an automatic translation of all UML elements into Maude, but only of those that we found relevant for our purposes, and that have been described in Section 4. They are enough for writing the ODP enterprise specifications of a system. Any other UML element—class methods, tagged values, sequence diagrams, etc.—will be ignored in the translation process. Of course, they can appear in the UML description of the system for illustration or clarification purposes, but will not be considered in this formalization of the system in Maude.

5.1. Structural Concepts

We have found that there is a strong correspondence between the UML model classes and the Maude classes, which allows an easy translation between both models. UML classes modeling roles and relationships will be directly mapped to the corresponding Maude classes, and UML subsystems modeling communities will be directly mapped to Maude configurations. UML class attributes will be mapped to Maude class attributes.

Roles can be related in different ways, including *generalizations* (which are defined by role subtyping relationships), *dependencies* (such as *usage* and other kinds of interactions), *compositions* (e.g. "is part of" relationships and aggregations), and *associations* (such as conceptual relationships among roles that involve a connection).

In order to translate UML relationships, we will distinguish between generalizations and the other ways in which roles can be related. First, generalizations can be modeled in Maude by using inheritance, as mentioned earlier. The rest of the relationships (*usage* and other *dependencies*, different kinds of *associations*, etc.) can be mapped to Maude classes, with the name of the relationship as the Maude class name, and whose attributes are the identifiers of the participants and the relationship's attributes. Instances of a relationship are therefore objects of the class that models the relationship.

The particular case of *binary* relationships without attributes can be mapped in a simpler way. We model in Maude this kind of relationships by using an additional attribute in each of the classes modeling the roles involved in the relationship. These attributes will hold the identifier(s) of the object(s) at the other end of the relationship. In the case of directed binary relationships without attributes (e.g. simple composition or dependency relationships) it is enough to store the identifiers of the managed objects as attributes of the managing objects.

Each constraint can be seen as a static invariant on the system. Constraint expressions on the model elements will be translated using membership assertions. Thus, if we want to express that a given predicate *P* is an invariant over a configuration of information objects, we may specify a subsort `CorrectConfig` of `Configuration`, such that only those configurations represented by terms of sort `Configuration` satisfying *P* are in `CorrectConfig`.

```
op P : Configuration -> Bool .
subsort CorrectConfig < Configuration .
var C : Configuration .
cmb C : CorrectConfig if P(C) .
```

These declarations make the invariants always true: a term of sort `Configuration` is in `CorrectConfig` if and only if it satisfies the invariant predicate. However, it does not constrain the possible states and state changes of the objects to which rules apply. To get this, we need to make sure that the configurations on which the rules apply satisfy the invariant, that is, that are of the right type.

```
crl [r] : C => C' if C : CorrectConfig .
```

Note that this approach is completely systematic and therefore can be easily automated. The invariant predicate *P* contains all the UML constraints on the model elements. Currently, we assume that these constrains are expressed in the Maude syntax.

5.2. Behavioral Concepts

As mentioned earlier, the way we model actions and behavioral policies in UML directly reflects the way we modeled them in Maude. Hence, there is again a direct translation between these two specifications.

Both *actions* and their corresponding *policies* are modeled in UML by paired object diagrams, joined by a dependency relationship with a condition attached. The two object diagrams (Before and After) can then be mapped into the left- and right-hand sides of the rule, respectively. The condition attached to the dependency relationship directly maps to the rule guard.

With all this, Maude's *configurations* (multisets of objects whose collective behavior is determined by the rewrite rules) clearly correspond to UML's object diagrams, whose behavior is now defined by our UML rules.

6. An Example

In order to illustrate our proposal we will use 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, and then by ourselves for validating our proposal and for comparing both approaches [6]. The example is loosely 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.*
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.*
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.

6.1. 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, undergrads, and postgrads), and two kinds of items (books and periodicals). There is also a possible **Loan** relationship between borrowers and items. A library community can be seen as composed by objects playing these roles. There is also a **Library** role which represents the community when considered as a composite object. Finally, role **Calendar** models the passage of time and provides the current date.

Figure 2 shows a UML class diagram with a possible model of the structure of our example. The corresponding Maude specifications describing this structure can be obtained as a direct translation from that diagram, and are as follows.

```
class Library | borrowers  : Set(Oid),
                calendar   : Oid,
                items      : Set(Oid),
                librarians : Set(Oid),
                loans      : Set(Oid) .
class Calendar | date : Date .
class Librarian .
class Borrower | bookLoanPeriod : Int,
                 borrowedItems  : Int,
                 fines          : Money,
                 loans          : Set(Oid),
                 maxLoans       : Int,
                 periodicalLoanPeriod : Int,
                 suspended      : Bool .
classes Academic Undergrad Postgrad .
subclasses
   Academic Undergrad Postgrad < Borrower .
class Item | free : Bool,
             loan : Default(Oid) .
classes Periodical Book .
subclasses Periodical Book < Item .
class Loan | borrower  : Oid,
             dueDate   : Date,
             issueDate : Date,
             item      : Oid .
```

Please notice how attributes modelling the different relationships have been added to the classes modelling the roles involved in them. Predefined sorts `Oid` and `Cid` are used

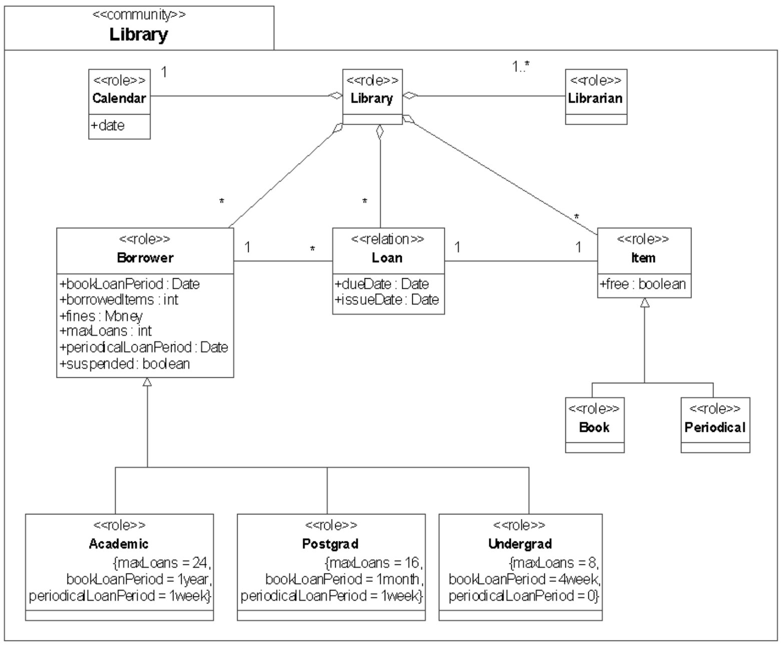

Figure 2. Structure of the library community.

for representing object identifiers and class identifiers, respectively.

There is also an invariant predicate capturing the restrictions imposed by the constraints in the model, namely the maximum allowances of each kind of borrower, the loan periods (e.g., 7 days for an academic to have a periodical, etc.). This invariant predicate can be expressed as follows:

```
op P : Configuration -> Bool .
eq P(none) = true .
eq P(< B : Academic | maxLoans : M > C)
  = (M == 24) and P(C) .
eq P(< B : Undergrad | maxLoans : M > C)
  = (M == 16) and P(C) .
...
```

6.2. 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. For the sake of brevity, in this paper we will concentrate just on particular cases two of these actions (the borrowing and the return of a book). See [6] for a discussion on the other actions.

Associated to these two actions there is a set of policies that govern their behavior, as for instance: (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 Undergrads, etc.) and is not suspended; (2) an `Undergrad` is *forbidden* to borrow a `Periodical` item; and (3) any borrower is *permitted* to borrow an item for a given period of time.

Let us see how these actions and policies can be specified in a graphical way, and then its corresponding translation to Maude rules.

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 UML diagram shown in Fig. 3, 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. Please notice how a new object (the loan), appears on the After object diagram, and the appropriate attributes of the other objects are updated. In our approach (following the Maude convention [4]) attributes of an object that are not relevant for a rule do not need to be mentioned, while attributes not appearing in the After object diagram will maintain their previous values unmodified.

Now, the Maude rule that corresponds to such a UML paired object diagram can be directly obtained:

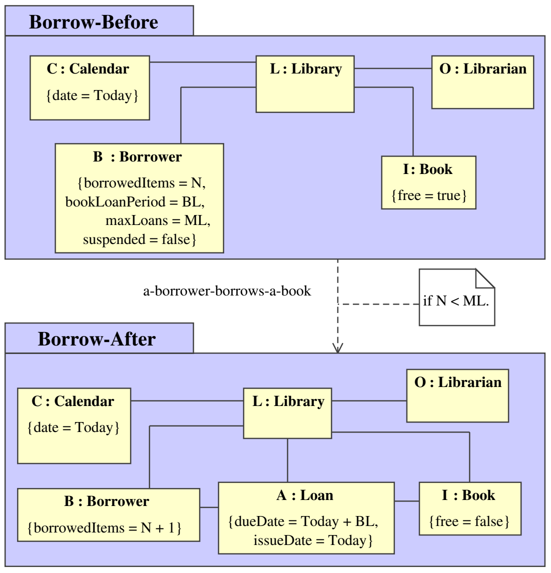

Figure 3. Graphical specification of the rule `a-borrower-borrows-a-book`**.**

```
crl [a-borrower-borrows-a-book] :
 < B : Borrower | borrowedItems : N,
                  maxLoans : ML,
                  bookLoanPeriod : BL,
                  loans : BLS,
                  suspended : false >
 < I : Book | free : true, loan : null >
 < L : Library | items : I IS,
                 borrowers : B BS,
                 librarians : O OL,
                 calendar : C,
                 loans : LLS >
 < O : Librarian | >
 < C : Calendar | date : Today >
 => < B : Borrower | loans : A BLS,
           borrowedItems : N + 1 >
    < I : Book | free : false, loan : A >
    < L : Library | loans : A LLS >
    < O : Librarian | >
    < C : Calendar | >
    < A : Loan | borrower : B, item : I,
          dueDate : Today + BL,
          issueDate : Today >
    if N < ML .
```

The second business rule is concerned with the return of an item, whose graphical representation is shown in Figure 4. We can see how the `Loan` object disappears in the After object diagram of the rule. The corresponding Maude rule is as follows.

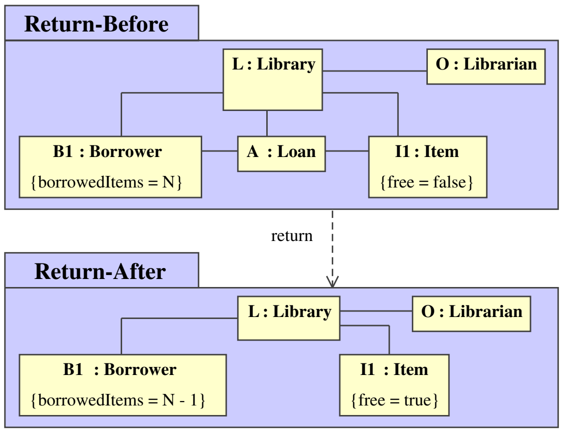

Figure 4. Graphical specification of the rule `return`**.**

```
rl [return] :
 < B : Borrower | borrowedItems : N,
                  loans : A BLS >
 < I : Item | loan : A >
 < A : Loan | borrower : B, item : I >
 < L : Library | items : I IS,
                 borrowers: B BS >
 => < B : Borrower | borrowedItems : N - 1,
                     loans : BLS >
    < I : Item | free : true, loan : null >
    < L : Library | > .
```

Note that our goal is to keep this formal specification hidden. Users of our tool work only at the graphical level, i.e., with the UML more-intuitive representation of the business rules modelling the actions and policies, being completely unaware of the corresponding formal specifications supporting them.

7. From UML to Maude and vice-versa

In this section we will discuss some of the features and implementation details of the tool we have developed for the automatic translation of the UML enterprise specification into the corresponding (formal) Maude specifications. The main goal of this tool is to help the software engineer capture the system requirements and model the behavior of the system using a friendly UML environment, while being supported by a formal method. In this way the software engineer can be unaware of all the complexity associated to the use of a formal notation such as the one provided by Maude, and its corresponding analysis toolkit. Despite this apparent simplicity and appealing interface, with this tool

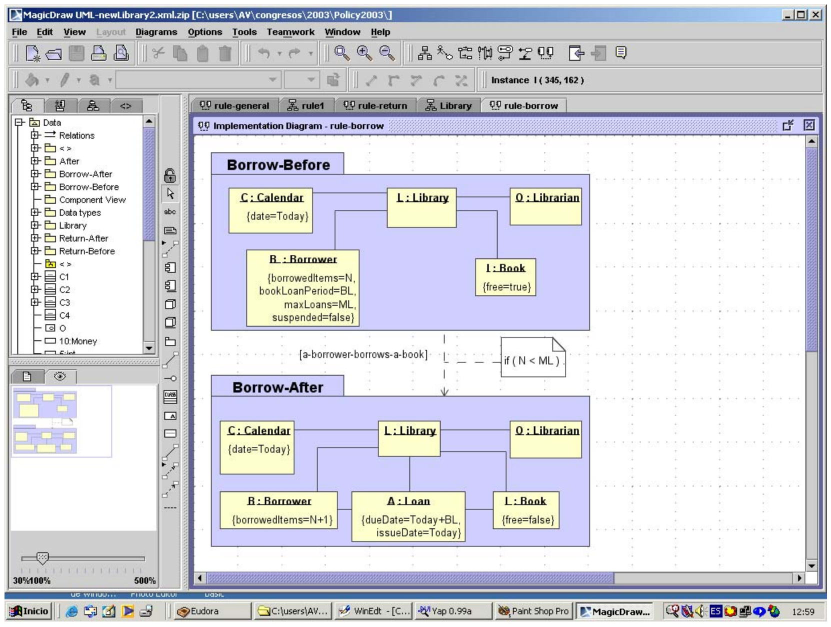

Figure 5. Writing rules in UML with MagicDraw.

we can still make use of many of the benefits and useful utilities that formal methods may provide.

The way our tools work is as follows. In the first place, the user draws the UML diagrams using any of the many software tools available in the market, with the only restriction that it should store the UML information using the XML tags defined in XMI 1.1 for UML. We have experienced the real benefits of using XMI, which has really provided the interoperability we needed, and has allowed us to accomplished the drawing-tool independence that we were pursuing. We have used MagicDraw as a drawing tool (`http://www.magicdraw.com`). Figure 5 shows the look and feel of the working environment. Once the diagrams are drawn, the user has just to invoke our tool, which then extracts all the information it needs from the XML descriptions of the diagrams, and builds the Maude class definitions from the UML class diagram, and the Maude rules specifying the system business rules from the paired object diagrams. We have strongly relied on the one-to-one relationship between the UML diagrams and the Maude classes and rules for automating the translation process.

Once the Maude specifications are written, they are passed on to the Maude interpreter. The translation process may help uncovering some common mistakes and under-specifications, such as inconsistent or missing data types of attributes, or missing details. They are communicated to the user so they can be fixed.

Once the Maude specifications are accepted by the Maude interpreter, we are ready for running some of the analysis tools provided by Maude. The simplest example is quick-prototyping. Maude specifications are executable, and therefore we can provide an initial configuration of objects, and ask the system to use the default strategy for the execution of the rewrite rules on them. In our case, it is just a matter of drawing a paired object diagram with the initial configuration on the Before diagram, its After diagram empty, and a number instead of the business rule label. That number indicates the maximum number of rewrite steps that the Maude engine should perform before returning the resulting configuration. This number is important, since the specifications of most business systems are neither Church-Rosser nor terminating. Hence, an upper limit of rewrite steps should be specified if we want the rewrite process to end. The initial configuration of objects and the number of rewrites is then passed to Maude, which processes it using the rules already defined. The result from Maude is another configuration of objects, which our tool translates and fits into the corresponding After object diagram (that was previously empty) so the user can visualize it.

Our tool is still at a preliminary stage. Current work tries to give access to other formal analysis tools from the UML environment. Our first goal is to be able to specify user-defined *strategies* for guiding the rewriting inference process. After that we plan to give support to stronger analysis methods, such as model checking [4]. The results obtained so far are promising and encouraging.

8. Concluding Remarks

In this paper we have presented a proposal for the graphical modeling of some of the ODP concepts, in particular actions and policies, for which no natural representation in UML existed as yet. The UML diagrams produced have a direct translation into Maude, therefore providing formal support.

One of the major benefits of our contribution is that it allows the stakeholders of the system to use a more user-friendly graphical notation like UML to express the system's structure, requirements and behavior, and then translate them into Maude specifications. The fact that the semantics of UML is often weak and imprecise, as opposed to the semantics of Maude, is taken care by the way the ODP specifications are written in UML, with a few stereotypes and object diagrams which guide the development of the specifications. Moreover, the tool that automates the translation process helps the production of correct specifications, asking for any missing details found in the UML specifications, but which are required by Maude (e.g. types of the attributes).

Apart from extending our tool for a smoother integration with Maude's analysis toolkit, there is much work ahead. For instance, there is the issue of how the individual ODP policies are combined, or how to deal with overlapping or inconsistent policies. Besides, in our approach polices are not explicitly stated, they just "shape" the rules that express the ODP actions. A possible alternative would be using a policy language for expressing the ODP policies, and then try to automate such a shaping of the actions. Identifying this policy language and integrating it in our proposal is a matter of further research.

Finally, representing enterprise policies is a valuable exercise in its own right, but it would be even more useful if they could be related to the computational and engineering mechanisms implementing them. In this sense, we are working in modeling other ODP viewpoints in Maude, and then using UML for providing a graphical access to these Maude specifications. Enhancing our tool for representing other viewpoints in UML is therefore another of our short-term goals.

Acknowledgements This work has been partially supported by Spanish Projects TIC2002-04309-C02-01, TIC2000-0701-C02, and TIC2001-2705-C03.

References

[1] J. Aagedal and Z. Milošević. ODP enterprise language: UML perspective. In *Proc. of EDOC'99*, pages 60–71, Germany, Sept. 1999. IEEE CS Press.

[2] X. Blanc, M.-P. Gervais, and R. L. Delliou. Using the UML language to express the ODP enterprise concepts. In *Proc. of EDOC'99*, pages 50–59, Germany, Sept. 1999. IEEE CS Press.

[3] J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software*. Addison-Wesley, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Comput. Sci.*, 285:187–243, Aug. 2002.

[5] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.

[6] F. Durán and A. Vallecillo. Writing ODP Enterprise specifications in Maude. In J. Cordeiro and H. Kilov, editors, *Proc. of WOODPECKER'01*, pages 55–68, Setubal, Portugal, July 2001.

[7] F. Durán and A. Vallecillo. Formalizing ODP Enterprise specifications in Maude. To appear in *Computer Standards & Interfaces*, 2003.

[8] 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, 1997.

[9] ISO/IEC. RM-ODP Enterprise Language. Draft International Standard ISO/IEC 15414, ITU-T X.911, ISO, 2001.

[10] P. Linington. RM-ODP: The architecture. In K. Milosevic and L. Armstrong, editors, *Open Distributed Processing II*, pages 15–33. Chapman & Hall, Feb. 1995.

[11] P. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In *Proc. of EDOC'99*, pages 72–82, Germany, Sept. 1999. IEEE CS Press.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Comput. Sci.*, 96:73–155, 1992.

[13] 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.

[14] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C. Talcott, editors, *Proc. of FMOODS 2000*, pages 89–117, Stanford, CA, Sept. 2000. Kluwer Academic Publishers.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[16] M. W. Steen and J. Derrick. ODP Enterprise Viewpoint Specification. *Computer Standards & Interfaces*, 22:165–189, Sept. 2000.

[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, *Proc. of FMOODS 2000*, pages 183–206, Stanford, CA, Sept. 2000. Kluwer Academic Publishers.