Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Computer Standards & Interfaces 25 (2003) 83–102

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Formalizing ODP enterprise specifications in Maude

Francisco Durán, Antonio Vallecillo*

ETSI Informática, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga 29071, Spain

Received 16 July 2002; received in revised form 26 November 2002; accepted 12 December 2002

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. Maude offers a simple, natural and accurate way of modeling the enterprise viewpoint concepts, which provides interesting benefits over previous modeling approaches, allows overcoming some of their limitations, and offers good tool support.

Keywords: Rewriting logic; Maude; RM-ODP; Enterprise 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. In particular, the Reference Model of Open Distributed Processing (RM-ODP) framework [12] 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 [16].

These viewpoints are sufficiently independent to simplify reasoning about the complete specification of the system. The architecture defined by RM-ODP tries to ensure the mutual consistency among the viewpoints, and the use of a common object model provides the glue that binds them all together. In addition, the use of the object paradigm provides abstraction and encapsulation, two important properties for the specification and design of complex systems.

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, without having to worry about other system considerations, such as particular details of its

* Corresponding author. Tel.: +34-952-13-2794; fax: +34-952-13-1397.
E-mail address: av@lcc.uma.es (A. Vallecillo).

0920-5489/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0920-5489(02)00121-6

implementation, or the technology used to implement the system.

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 notations have been proposed for the different viewpoints by different authors, which seem to agree on the need to represent the semantics of the ODP viewpoints concepts in a precise manner [2,5,12,16,25]. For example, formal description techniques (FDTs) such as LOTOS and SDL have been proposed for the computational viewpoint [12], and Z and Object-Z for the information and enterprise viewpoints [27]. On a different arena, object-oriented modeling languages, such as UML or Fusion [8], have also been proposed for ODP information and enterprise modeling. Although they are not formal notations, they have been used because of their appealing graphical syntax, and because they are part of fully integrated development methodologies (see, e.g. Refs. [1,3,17]). However, their lack of precise semantics represents an impediment for achieving any sort of formal analysis of the systems.

In this paper, we explore a new alternative for specifying the enterprise viewpoint. We propose Maude [6], an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems [22]. As we shall see, this choice not only offers new benefits over the previous approaches for formalizing ODP enterprise specifications (in particular, over the Object-Z approach, probably the most widely accepted proposal), but also allows to overcome some of their limitations. We propose a simple and natural way of modeling the enterprise viewpoint concepts, where, for example, roles are modeled as Maude classes, enterprise objects as Maude objects, communities as Maude configurations, and the system behavior is modeled by Maude rules, whose form is determined by the different kinds of policies. We make special emphasis on the capability of Maude for avoiding overspecification, allowing us to produce specifications faithful to the current state of knowledge, and to the desired level of detail. This approach facilitates a process of refinement in which we can uncover underspecified details at one level, which are covered in successive iterations. The object-oriented nature and simplicity of the Maude specifications make them easily understandable, helping involve stakeholders of diverse backgrounds in the system specification process.

In addition to the nice properties of the Maude specifications obtained following our 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 [11], model checking [10], or theorem proving [7]. Maude offers a comprehensive toolkit for automating such kinds of formal analysis of specifications. In this paper, we will discuss just the controlled execution of the specifications.

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 presents our proposal, describing how to write enterprise specifications in Maude. Section 5 is dedicated to a small case study that illustrates our approach. The execution of the Maude specification produced is discussed in Section 6. Finally, Section 7 compares our work to other similar approaches, and Section 8 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 [13]. 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.

An enterprise specification also identifies those actions that involve *accountability* of a party, where a party represents a natural person or any other entity considered to have some of the rights, powers, and duties of a natural person [13]. Authority or functions can be delegated. Principal parties are responsible for the acts of any parties acting as their delegated agents, including their possible commitments, prescriptions, evaluations, declarations, and further delegations.

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 [13].

The first step towards building the enterprise specifications of a system is to identify all those elements. Although the ODP Enterprise Language specification does not prescribe any particular method for that, we propose the following tasks for producing such specifications:

(1) Identify the communities, the roles in such communities, and the relationships among those roles and among those communities.
(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).
(4) Identify the policies that rule the actions (permissions, obligations, authorizations, prohibitions), and the effects of the possible violations of those policies.
(5) Identify the behavior that may change the structure or the members of each community during its lifetime, and the policies that rule such a behavior.
(6) Identify the behavior that may change the rules that govern the system, and the policies that rule such a behavior—changes in the structure, behavior or policies of a community can occur only if the specification includes the behavior that can cause those changes [13].
(7) Finally, identify the actions that involve accountability of a party.

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. Points 5 and 6 define the rules that govern the allowed changes in the structure and policies of the system during its lifetime. Finally, point 7 defines the accountability rules. Of course, the order of these activities needs not necessarily be linear.

3. Rewriting logic and Maude

Maude [6] is a high-level language and a high-performance interpreter and compiler that supports equational and rewriting logic specification and programming of systems. Rewriting logic is parameterized by its underlying equational logic, which can be unsorted, many-sorted, order-sorted, or membership equational logic. The underlying equational logic chosen for Maude is membership equational logic [21]. 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 [22].

Rewriting logic [20] 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 preconditions, 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, a_i the attribute identifiers, and S_i the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form $\langle$O : C|a_1:v_1, ..., a_n:$v_n\rangle$, where O is the name of the object, and v_i 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

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

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, $C_1 \ldots C_n$, $C'_{i1} \ldots C'_{ik}$ 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 $O_{i1} \ldots O_{ik}$ 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 a rule `debit` which 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 that rule, the message is consumed, and the balance of the account is modified.

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

Maude distinguishes two kinds of inheritance, namely *class inheritance* and *module inheritance*. Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration $C < C'$, indicating that C is a subclass of 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. ODP's notion of subtyping—

A is a subtype of B if every <X> that satisfies A also satisfies B—corresponds to Maude's class inheritance. On the other hand, the ODP's notion of inheritance that allows the suppression and modification of the attributes and methods of the base class [12] corresponds to Maude's module inheritance. Throughout the paper, by inheritance we will mean Maude's notion of *class* inheritance, i.e. ODP's subtyping. Multiple inheritance is supported in Maude [6].

4. Writing enterprise specifications in Maude

In this section, we present our proposal for modeling in Maude all the concepts described in Section 2, which constitute the enterprise specification of a system.

4.1. Structural concepts

Each **role** will be modeled by a Maude 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 Maude objects. In Maude, each object belongs to a class, which may be changed during its lifetime. The class of an object is obtained by composing all the Maude classes that model the different roles that the object fulfills, which may be realized in Maude by multiple class inheritance.

A **community** is a composition of enterprise objects and, therefore, it can be naturally modeled by Maude's *configurations*. 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 a Maude object (belonging to some class).

4.2. Relationships among roles

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 ISO General Relationship Model may be useful here.

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 model relationships, we will distinguish between generalizations, and the rest of the ways in which roles can be related. First, generalizations can be modeled in Maude by using inheritance, as mentioned earlier. Now, for modeling the rest of the relationships (*usage* and other *dependencies*, different kinds of *associations*, etc.), we propose using a Maude class with the name of the relationship as its name, and whose attributes are the identifiers of the participants and such a 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 modeled in a simpler way. We model 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 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.

4.3. Behavioral concepts

Actions will be modeled by rewrite rules. The left-hand side and guard of a rewrite rule expresses the conditions that must be satisfied by 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.

A **process** is a collection of steps taking place in a prescribed manner, and leading to an objective [13]. The collective behavior of a community may be reported as a set of processes. Maude's internal strategies are a natural and general form of controlling the evolution of the systems specified (see Section 6).

4.4. Policies

Policies (both membership and behavioral) determine the form of the rewrite rules, stating the conditions for the action to happen (either by restricting the pattern of the left-hand side of the rule, or by explicitly stating a condition with an if guard). The way to model the different policies that govern the behavior of a system will depend on the kind of policy:

- **Permissions** allow state transitions. Therefore, a permission will be modeled by a rule whose left-hand side and guard determine the scenario of the permitted action(s) and their participants, and whose right-hand side 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 (i.e. its scope [13]). These will be modeled as normal 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 customer that does not return a hired car). 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' left-hand sides and guards for explicitly banning such actions. In this way, the system will automatically prevent the prohibited action to happen. For actions whose occurrence escape 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 modeled by more than one rewrite rule, and be controlled by different policies. Therefore, a policy may be modeled by more than one 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 in Maude by individually applying those policies to each of the rules 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 rules for modeling both actions and policies. Of course, there are other alternatives for specifying business systems and business rules using Maude. For instance, we could have modeled ODP actions by Maude messages, and ODP policies by Maude 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 Maude rules, with guards that determine when the rules are enabled. 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 to use Maude's *strategies* for controlling the execution of the system (see Section 6). 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).

In addition, it ought to be emphasized how the use of Maude's *configurations* (multiset of objects whose collective behavior is determined by the rewrite rules) 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.

4.5. Rules for changing the structure and policies of a community

RM-ODP considers the possibility of changes in the structure and behavior of a community. The kind of changes cover the introduction of new policy rules, new roles into the communities, and changes in the existing rules.

In general, the reflective capabilities of rewriting logic and Maude could have been considered for specifying change. However, it does not seem to be necessary, since the ODP Enterprise Language Standard clearly states that changes in the structure or behavior of a community can occur only if an enterprise specification includes behavior that can cause such changes [13].

This can be seen as a strong assumption in some situations, since the behavior of the environment of an open system is generally unpredictable. However, in an enterprise specification, the possible kinds of changes allowed for a system need to be considered when building the system's specifications, and the behavior that can cause the changes need to be specified in advance. Possibly, this is to avoid the scope of a system from drifting into unspecified or unwanted behaviors. In any case, the requirement of having to specify the kinds of changes allowed and their enabling behaviors simplifies their specification in Maude, since they can be treated as possible (structural, behavioural, or policy) alternatives. The occurrence of a behavior that causes such a change will enable the new structure, behavior, or rules.

5. A case study

In order to illustrate our proposal, we will specify in Maude a simple example, a rental car store named *VStore*. The regulations (i.e. business rules) of the system, especially those that rule the rental processes, are as follows:

(1) Cars are rented for a specific number of days, after which they should be returned to the store.
(2) A car can be rented only if it is available.
(3) No credit is allowed. Customers must pay cash.
(4) Customers must make a deposit of the estimated rental charges at pick-up time.
(5) Rental charges depend on the car class. *VStore* stores define three categories: economy, midsize, and full-size cars.
(6) When a rented car is returned, the deposit is used to pay the rental charges, which are calculated in accordance to the conditions at pick-up time.
(7) If a car is returned before the due date, the customer is charged only for the number of days the car has been used. In this case, the rest of the deposit is reimbursed to the customer at return time.
(8) Customers who return a rented car after its due date are charged for all the days the car has been used, with an additional 20% for each day after the due date.
(9) Failure to return the car on time or to pay a debt may result in the suspension of renting facilities.
(10) *VStore* staff members can also rent cars.
(11) Staff members and regular customers benefit from special discounts in all rentals.
(12) A customer qualifies as "regular" by *VStore* when the accumulated amount of money paid to the company by the customer is above certain threshold.

Although not explicitly mentioned as such, these business rules define the permissions, obligations and prohibitions for the people, systems, and artifacts fulfilling roles in the virtual store community.

We have left many details in the textual description of the system above intentionally unspecified. Our purpose is to illustrate that sometimes we cannot, or simply do not want, to make explicit certain details. Our (formal) specification must of course conform to this description, giving the specification of the system at the same level of abstraction, thus, leaving these details also unspecified. Our point is that more than overspecifying, our aim should be *detecting* and *recognizing* the missing details, but never hiding them. In fact, this is a difficult task. Very often, we make unintentional decisions based on what can be seen as "common sense" to us, but which fail to be true in the business context. Helping make explicit such kind of assumptions already supposes a great benefit. Of course, the specification produced may be further refined as many times as wished, making the corresponding decisions, until the right level of detail is reached.

5.1. The structure of the system

Let us begin with the static aspects of this community, i.e. its structure. From the text above, we can identify three main roles, namely the store, customers, and cars. There are three special kinds of customers (staff, casual customers, and regular customers), and three kinds of cars (economy, mid-size, and full-size cars). A rental car community can be seen as composed by objects fulfilling these roles.

Customers may rent cars. This relationship may be represented by a `Rental` class which, in addition to references to the objects involved in the relationship, have some additional attributes. The system also requires some control over time, which we get with a class representing calendars that provide the current date and simulate the passage of time.

The customer role is modeled by the `Customer` class, which has two attributes, namely, `cash` and `debt`, for keeping record, respectively, of the amount of cash that the customer currently has, and his/her debt with *VStore*. Classes `Staff`, `CasualCust`, and `RegularCust` are subclasses of `Customer`, and do not have any additional attributes. Such classes may be defined by the following Maude declarations:

```
class Customer | cash : Int,
                 debt : Int .
classes Staff CasualCust RegularCust .
subclasses Staff CasualCust
             RegularCust < Customer .
```

The role car is modeled by class `Car`, whose attribute `available` indicates whether the car is currently available or not. We model the different types of cars for rent in *VStore* by three different subclasses, namely `EconomyCar`, `MidSizeCar`, and `FullSizeCar`.

```
class Car | available : Bool .
classes EconomyCar MidSizeCar
         FullSizeCar .
subclasses EconomyCar MidSizeCar
               FullSizeCar < Car .
```

As part of the *VStore* rental car store community, we find a `VStore` store object, which represents the community when considered as a composite object. Class `VStore` models this role, whose attributes represent the information concerning the general parameters of such stores: the rates applicable to each type of car, the discounts for each type of customer renting each type of car, the identifiers of the customers who are suspended, the amount of money above which casual customers are qualified as regular, the record with the amount of money spent in the store by each of the customers, and the daily penalty for late return (20%). In addition, attributes `customers`, `cars`, `rentals`, and `calendar` store the identifiers of the objects participating in the relationships with the `VStore` composite object (see Fig. 1 for a UML class diagram with the structure of our example). Please note that those are directed binary relationships and, therefore, we only store the identifiers of the subordinate objects as attributes of the object that reference them.

```
class VStore |
  discounts : PFun(Tuple(Cid, Cid), Int),
  payments  : PFun(Oid, Int),
  penalty   : Int,
  threshold : Int,
  suspended : Set(Oid),
  rates     : PFun(Cid, Int),
  customers : Set(Oid),
  cars      : Set(Oid),
  rentals   : Set(Oid),
  calendar  : Oid .
```

The information on rates, discounts and money spent is modeled by attributes of sort `PFun` of partial functions, associating the appropriate values to each of the different actors involved. This sort has two main operators available: a consult operator `_[_]`, which takes a partial function and a domain element and returns its image, or a default value if it has no associated image; and the operator `_[_->_]`, that modifies the value associated to the domain element given as second argument in the partial function given as first argument with the image element given as third argument.

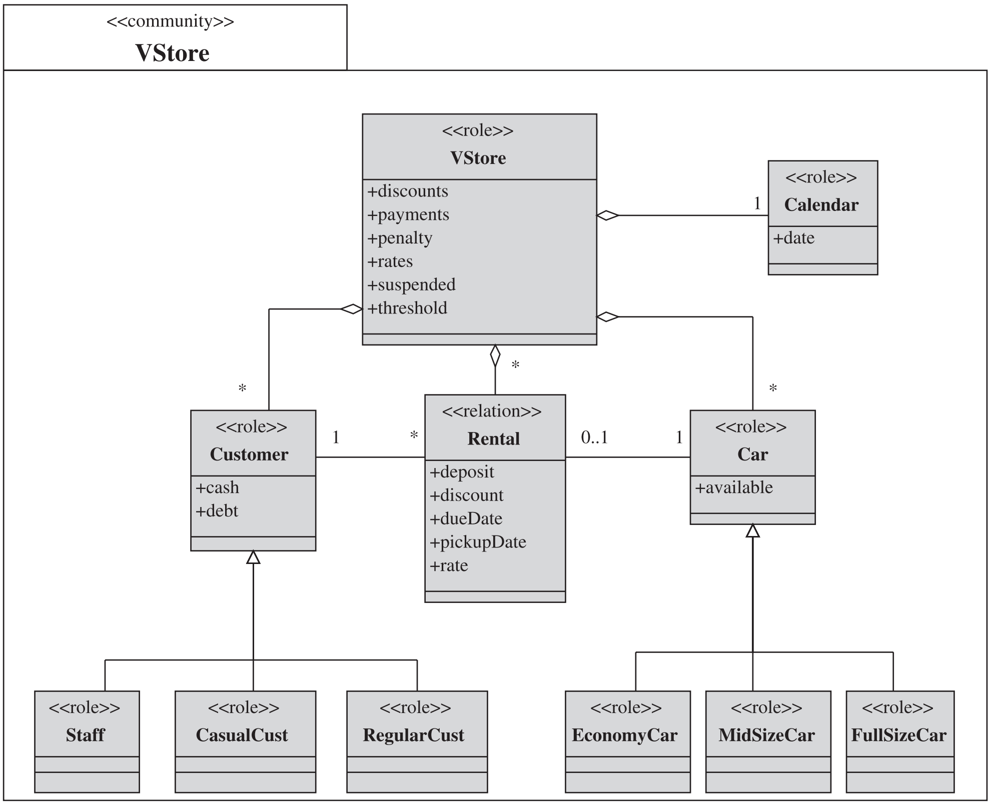

Fig. 1. Structure of the *VStore* community.

The rates for the different cars are stored in the attribute `rates`, of sort `PFun(Cid, Int)`, that associates the per-day rate to be charged to a customer for renting a given type of car. Thus, supposing that `Rts` is a variable of sort `PFun(Cid, Int)`, with value the partial function assigning the appropriate rates to each type of car, we have that `Rts[FullSizeCar]` is the per-day rate for renting a full size car. If we want to increase this rate in, say, 20%, we can use the expression `Rts[FullSizeCar->Rts[FullSizeCar]*120/100]`. The discounts applied to each customer on each type of car and the amount of the purchases of each customer are stored, respectively, in attributes `payments` and `discounts`. The set of the identifiers of the customers who are suspended is stored in an attribute `suspended` of sort `Set(Oid)`. In addition, notice the use of the predefined sorts `Oid` and `Cid` for object identifiers and class identifiers, respectively.

Please note that in this way we are modeling *VStore* communities as objects of class `VStore`, which represent the communities when considered at this level of detail. This specification will allow us, for instance, to easily 'compose' communities with different particular details (e.g. discounts may change from one store to another), allowing them to easily co-exist. Moreover, there may be objects fulfilling roles in different communities.

Each object of class `Rental` will establish a relationship between a customer and a car, whose identifiers are kept in attributes `customer` and `car`, respectively. In addition to these, the class `Rental` is also declared with attributes `deposit`, `pickUpDate`, `dueDate`, `rate`, and `discount` to store, respectively, the amount of money left as deposit by the customer, the date in which the car is picked up by the customer, the date in which the car should be returned to the store, and the rate and discount applied at pick-up time.

```
class Rental |
  deposit : Int, discount   : Int,
  dueDate : Int, pickUpDate : Int,
  rate    : Int, customer   : Oid,
  car     : Oid .
```

Given the simple use that we are going to make of dates, we can represent them, for example, as integer numbers. Then, we may have a calendar object that keeps the current date, which gets increased by a rewrite rule as follows.

```
class Calendar | date : Int .
rl [new-day] :
  < O : Calendar | date : F >
  => < O : Calendar | date : F + 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 works by Kosiuczenko and Wirsing [15] or by Ölveczky and Meseguer [26] on the specification of real-time systems in rewriting logic and Maude for a discussion on these issues.

As suggested by different authors [1,3,17,27], the structure of the communities can be specified using UML, as a first step towards formalizing it. We have shown in Fig. 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.

5.2. Actions and policies governing the system's behavior

Five actions can be identified in the example: (1) a customer rents a car, (2) a customer returns a rented car, (3) a customer is suspended for being late in paying his/her debt or for being late in returning a rented car, (4) a customer pays (part of) his/her debt, and (5) the qualification of a customer is changed (from casual to regular). Each of the following subsections is devoted to one of these five actions and to the policies that rule them.

5.2.1. Action 1: car rental

In the first place, renting a car needs the customer not to be suspended, the car to be available (i.e. not currently rented), and that the customer has enough money to make the deposit. The enterprise policies conditioning the rental of a car are the following:

(1) Any customer is *permitted* to rent a car if (s)he has enough cash to pay the deposit and the car is available.
(2) Suspended customers are *forbidden* to rent cars.
(3) Staff members and regular customers are *authorized* to have a discount in all car rentals, according to the virtual store discount list.
(4) Customers are *obliged* to make a deposit of the estimated rental charges.

The rental of a car by a customer is specified by the rule `car-rental` below, which involves the customer renting the car, the car itself, the store, and a calendar object supplying the current date. The rental can take place if the customer is not suspended, that is, if its identifier is not in the set of identifiers of suspended customers of the store, and if the customer has enough cash to make the corresponding deposit. Notice that a customer could rent a car for less time (s)he really is going to use it on purpose because either (s)he does not have enough money for the deposit, or prefers making a smaller deposit. According to the description of the system, the payment takes place when returning the car, although with an extra charge for the days the car was not reserved. The rate and discount to be applied are however those at pick-up time, which need to be part of the rental information so that they can then be used. To make explicit the fact that the rental is taking place in the context of the store, an object of class `VStore` is involved in the rule.

```
crl [car-rental] :
 < U : Customer | cash : M >
 < I : Car | available : true >
              *** the car is available
 < V : VStore | suspended : US,
       rates : Rts, discounts : Dscnts,
       calendar : C, cars : I IS,
       customers : U SS, rentals : RS >
 < C : Calendar | date : Today >
 => < U : Customer | cash : M - Amnt >
    < I : Car | available : false >
    < V : VStore | rentals : A RS >
    < C : Calendar | >
    < A : Rental | pickUpDate : Today,
          dueDate : Today + NumDays,
          car : I, deposit : Amnt,
          customer : U, rate : Rt,
          discount : Dscnt >
 if not U in US
     *** the customer is not suspended
    /\ Rt := Rts[class(< I : Car | >)]
    /\ Dscnt :=
        Dscnts[
           (class(< U : Customer | >),
            class(< I : Car | >))]
    /\ Amnt := (Rt - Dscnt) * NumDays
    /\ M >= Amnt .
     *** enough cash to make a deposit
```

In Maude, 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 that the variables `A` and `NumDays` appear in the right-hand side or condition of the rule but not in its left-hand side. Note as well the use of attributes `customer` and `car` in objects of class `Rental`, which make explicit that a rental relationship is between the customer and the car specified by these attributes. Likewise for attributes `customers`, `cars`, and `calendar` of object `V` of class `VStore`, which indicate that the customer, car and calendar appearing on the rule should be known to the store. After the action, the rental is added to the set of rentals kept by the store.

Rules may be applied on objects of the classes specified in the rules or of any of their subclasses. The function `class` takes an object as argument and returns its actual class. Thus, if the `Customer` object to which the above rule applies is, for instance, ⟨`'john:RegularCust`|...⟩, then the class function applied to it returns `RegularCust`, and not `Customer`.

Finally, note the use of *matching equations* of the form `t:=t′` in the condition. Matching equations are interpreted as ordinary equations; however, operationally the pattern term on the left `t` is matched against the normal form of the subject term on the right `t′`. Hence, the variables in `t` not appearing in the left-hand side of the rule get instantiated as a result of the matching. The satisfaction of the conditions is attempted sequentially from left to right and, thus, variables that get instantiated in matching equations can be used in the rest of the condition or in the right-hand side. In the rule above, it is used as a let or where section in conventional functional languages.

5.2.2. Action 2: car return

The enterprise policies conditioning the return of a car that can be extracted from the description are the following:

(1) Customers are *obliged* to return the rented cars within their specified rent period.
(2) Customers are *obliged* to pay the charges corresponding to the number of days they have used the car.
(3) Customers are *obliged* to pay a 20% extra for the days they have kept the car after the due date.
(4) The store is *obliged* to reimburse the part of the deposit exceeding the due charges when returning a car before the due date.

A customer returning a car late cannot be forced to pay the total amount of money due at return time. Perhaps (s)he does not have such an amount of money at that time. In fact, the description just says that the customer must pay, but not when. If we impose that having enough money for paying the rental fee is a condition for returning a car, we may be overspecifying the system by forcing customers to keep cars more time than they would do otherwise.

The return of a rented car is specified by the rules below. The first rule handles the case in which the car is returned on date, that is, the current date is smaller or equal than the due date and, therefore, the deposit is greater or equal than the amount due. Notice that the rate and discount to be used in the calculation of the amount due are those at pick-up time, which are stored as attributes of the `Rental` object.

```
crl [on-date-car-return] :
  < U : Customer | cash : M >
  < I : Car | >
  < A : Rental | customer : U, car : I,
    rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt,
    deposit : Dpst >
  < V : VStore | payments : Pmnts,
    cars : I IS, customers : U SS,
    calendar : C, rentals : A RS >
  < C : Calendar | date : Today >
  => < U : Customer |
       cash : M + Dpst - Amnt >
     < I : Car | available : true >
     < V : VStore | rentals : RS,
       payments :
         if Pmnts[U] == null
            *** no record for this customer
         then Pmnts[U -> Amnt]
         else Pmnts[U -> Pmnts[U] + Amnt]
         fi >
     < C : Calendar | >
  if (Today <= Ddt)
   /\ Amnt :=(Rt - Dscnt)*(Today - Ppdt) .
```

In this case, the deposit is greater than the amount due and, therefore, part of this deposit needs to be reimbursed. Note also that the `VStore` object keeps record of the amount of money spent by each customer in the store and, thus, it must be updated accordingly. We can see how the `Rental` object disappears in the right-hand side of the rules, it is removed from the set of rentals known by the store, and the availability of the car is restored.

The second rule deals with the case in which the car is returned late. The amount to be paid is calculated at drop-off time, but the rate and discount to be used, those at pick-up time, may have changed when returning the car.

```
crl [late-car-return] :
  < U : Customer | debt : M >
  < I : Car | >
  < A : Rental | customer : U, car : I,
    rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt,
    deposit : Dpst >
  < V : VStore | payments : Pmnts,
    penalty : Pnlt, rentals : A RS,
    cars : I IS, customers : U SS,
    calendar : C >
  < C : Calendar | date : Today >
  => < U : Customer |
       debt : M + Amnt - Dpst >
     < I : Car | available : true >
     < V : VStore | rentals : RS,
       payments :
         if Pmnts[U] == null
         then Pmnts[U -> Dpst]
         else Pmnts[U -> Pmnts[U] + Dpst]
         fi >
     < C : Calendar | >
  if Ddt < Today   *** it is returned late
     /\ Amnt := (Rt - Dscnt) * (Ddt - Ppdt)
             + (Rt - Dscnt) * (Today - Ddt)
             * (100 + Pnlt) / 100 .
```

In this case, the customer's debt is increased by the part of the amount due not covered by the deposit.

5.2.3. Action 3: debt payment

The description of the system says nothing about when or how customers should pay their debts. They are obliged to do it however. We have included a Maude rule with which customers may pay their due charges. Notice that the amount of the debt paid is left unspecified in this rule, since it may be paid either all at once or in several settlements.

```
crl [pay-debt] :
  < V : VStore | payments : Pmnts,
    customers : U SS, calendar : C >
  < U : Customer | debt : M, cash : N >
  < C : Calendar | date : Today >
  => < V : VStore |
       payments :
         Pmnts[U -> Pnmts[U] + Amnt] >
     < U : Customer |
       debt : M - Amnt, cash : N - Amnt >
     < C : Calendar | >
  if 0 < Amnt /\ Amnt <= N /\ Amnt <= M .
```

We are assuming that if there is a debt then there has been a previous payment and, therefore, that there is already a record for that customer.

5.2.4. Action 4: customer suspension

The enterprise policies conditioning the suspension of a customer that can be extracted from the text are the following:

(1) Customers are *obliged* to pay their debts.
(2) Customers are *obliged* to return rented cars on the due date.
(3) A *violation* of any of the previous rules may result in a suspension action to the customer.

The text says that customers who are late in returning a rented car or in paying their debts “may” be suspended. However, nothing is said about the reasons for taking such a decision, or when they should be suspended, that is, a customer could be suspended right after the car is returned without having paid all the charges, after some days of grace, or never. In most cases there will be a fixed criteria, as for example, suspending customers that are two days late, or two months. However, since no indications are given in the text, these details are left unspecified in the rules.

These policies can be faithfully modeled by the following two rewrite rules. The first rule deals with the case in which a customer has a pending debt, and the second one handles the case in which a customer is late in returning a rented car.

```
crl [suspend-late-payers] :
  < V : VStore |
    suspended : US, customers : U SS >
  < U : Customer | debt : M >
  => < V : VStore | suspended : U US >
     < U : Customer | >
  if (not U in US) and M > 0 .

crl [suspend-late-returns] :
  < V : VStore |
    suspended : US, cars : I IS,
    customers : U SS, calendar : C >
  < U : Customer | >
  < I : Car | >
  < A : Rental |
    customer : U, car : I, dueDate : F >
  < C : Calendar | date : Today >
  => < V : VStore | suspended : U US >
     < U : Customer | >
     < I : Car | >
     < A : Rental | >
     < C : Calendar | >
  if (not U in US) and F < Today .
```

5.2.5. Action 5: qualification change

The description of the system also explains that customers may be requalified according to the amount of money they have spent at the store. We can extract only one enterprise policy conditioning the change in the qualification of a customer from the text: The store is *obliged* to requalify customers, from casual to regular, when the amount of money they spent in the store is above the store’s threshold.

The upgrade of a customer can then be modeled with the following rule:

```
crl [upgrade-to-regular] :
  < U : CasualCust | cash : M, debt : N >
  < V : VStore |
    threshold : Thrshld, payments : Pmnts,
    customers : U SS, calendar : C >
  < C : Calendar | date : Today >
  => < U : RegularCust |
         cash : M, debt : N >
     < V : VStore | >
     < C : Calendar | >
  if Pmnts[U] >= Thrshld .
```

In this rule, an object of class `CasualCust` becomes of class `RegularCust` when the accumulated amount of its purchases exceeds the store's threshold. The partial function stored in the attribute `payments` gives us the amount of money spent by every customer. In Maude, objects changing their classes must show all their attributes in the right-hand sides of the rules.

Note that nothing is said about whether a suspended customer or a customer with a pending debt can be upgraded and, therefore, this information is not taken into account in the rule modeling the action. Again, the fact of writing the rules helps uncovering these unspecified situations, which need to be properly addressed in successive refinements of the enterprise specifications.

5.3. Further issues

It is important to notice that there are still many other details left unspecified in the original system description. For example, nothing was said about the restoration of the renting rights of a suspended customer, or the possible "demotion" of a regular customer to casual if, for example, (s)he stops spending enough money in the store and, therefore, no rules are given for those actions. As we mentioned before, our intention was to produce a specification faithful to the description given, and hence we have restricted ourselves to the details provided. What we have tried to show here is how the specification process followed has allowed us to: (a) uncover many underspecified details in the system description; and (b) still produce faithful specifications, without incurring in overspecification.

The specifications produced with such a level of detail could be then refined with the information provided by the stakeholders of the system, trying to solve the ambiguities and gaps detected in them. The object-oriented nature and simplicity of the Maude rules make them easily understandable, which helps involving stakeholders of diverse backgrounds in the system specification process. In addition, the object-oriented nature of Maude perfectly fits with the object-oriented nature of ODP, facilitating the conceptual mapping between these two "worlds".

Notice also that communities modeled in this way are easily composable. It is possible, for example, having several stores sharing cars and customers, or cars being returned in stores different to the one they were rented in. We can also compose communities of different nature very easily, and having objects fulfilling different roles in different communities.

Finally, there is the issue of accountability. At this level of abstraction, the responsibilities for each of the actions have not been made explicit. Responsibilities could be specified in different ways, for example, by having simple comments on the rules, or metalevel functions returning any information on the actions. However, our current thought is that, 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) subactions. This is the way in which several Requirements Engineering proposals work (such as KAOS [28]), in which actions are refined until a responsible agent can be identified. Refinements are based on questions, such as *what*, *how*, *why*, and *by whom*. Thus, in our example it will be made explicit at an appropriate (lower) level of abstraction the fact that it is the customer who initiates the renting action, for instance.

6. Executing the enterprise specifications

Once the system specifications are written using this modeling approach, what we get is a rewrite logic specification of the system, which can be used for formally reasoning about it. The fact that under reasonable assumptions, rewriting logic specifications are executable, allows us to apply a flexible range of increasingly stronger formal analysis methods, such as runtime verification, model checking, narrowing analysis, or theorem proving. Each analysis method has its own complexity, so we could start with the most efficient ones, and leave the most complex methods for the analysis of systems that have already passed all the previous "filters".

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; Knuth–Bendix and coherence completion tools; and a tool to specify, analyze, and model check real-time specifications [7]. In this section, we will concentrate on the controlled execution of the specifications.

Maude specifications as the one presented in this paper are rewrite theories that do not need to be either Church–Rosser or terminating. That is, in principle, the inference process could not terminate, or could go in many different directions. Maude implements a default strategy for the execution of rewrite systems [6], which may be enough in some cases. But it is with the possibility of specifying user-defined *strategies* with which Maude provides absolute control over the rewriting inference process. Moreover, in most cases we will not only want to follow one of the possible paths, but we would like to explore different alternatives, or even considering all the possible ones.

In Maude, thanks to its reflective capabilities, strategies are made *internal* to the logic, that is, strategies are defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module. In fact, there is great freedom for defining many different types of strategies, or even many different strategy languages inside Maude. This can be done in a completely user-definable way, so users are not limited by a fixed and closed particular strategy language [6].

Let us illustrate some of the possibilities with a strategy for controlling the execution of the rules modeling the *VStore* system, which give rise to nondeterministic and nonterminating computations. A simple but interesting strategy may be one that allows us to execute a given sequence of rules, that is, to accomplish sequentially a series of actions from a particular initial state.

In this case, a strategy can be defined as a sequence of pairs (L, S), where L is the label of the rule to be applied and S is a partial substitution to be used in the application of that rule. Notice that in order to specify a particular action, it is not enough to give the name of the rule that models the action—given a configuration of objects, a rule may be applied in different ways. For instance, rule `suspend-late-payers` may be applied to different customers. To make sure that the customer being suspended when applying that rule is `'john`, we need to state the partial substitution `'U <-` $\overline{\texttt{john}}$ when applying such a rule. In addition, note that the strategies are to be defined at the metalevel: overlined terms stand for the meta-representation of these terms.

Suppose then that we start with the following initial configuration of objects:

```
< O1 : VStore |
   discounts :
     ((CasualCust, MidSizeCar), 0)
     ((CasualCust, FullSizeCar), 0)
     ((RegularCust, MidSizeCar), 20)
     ((RegularCust, FullSizeCar), 30),
   rates     :
     (MidSizeCar, 150) (FullSizeCar, 200),
   payments  : mt,
   suspended : mt,
               *** No suspended customers
   threshold : 450,
   penalty   : 20,
   customers : 'C1 'C2,
   calendar  : 'C,
   cars      : 'A1 'A3 'A5 >
< C1 : CasualCust |
  cash : 5000, debt : 0 >
< C2 : CasualCust |
  cash : 5000, debt : 0 >
< A1 : EconomyCar | available : true >
< A3 : MidSizeCar | available : true >
< A5 : FullSizeCar | available : true >
< C : Calendar | date : 0 >
```

This configuration consists of a *VStore* store `O1`, two clients `C1` and `C2`, three cars `A1`, `A3`, and `A5`, and a calendar object `C`. Let us call it *VStoreConf*.

Now, let *VStoreStrat* be the following sequence of pairs (rule label-substitution) that define the strategy—by means of a sequence of actions in this case:

```
('car-rental,                                 *** client C1 rents the mid-
 (('U <- C1); ('I <- A3);                     *** size car A3 for 2 days
  ('NumDays <- 2); ('A <- 'a0)))
('new-day, none)                              *** two days pass
('new-day, none)
('on-date-car-return, none)                   *** car A3 is returned
('new-day, none)
('car-rental,                                 *** client C1 rents the full
 (('U <- C1); ('I <- A5);                     *** size car A5 for 1 day
  ('NumDays <- 1); ('A <- 'a1)))
('new-day, none)                              *** two days pass
('new-day, none)
('late-car-return, none)                      *** car A5 is returned
('new-day, none)
('suspend-late-payers, none)                  *** client C1 is suspended
('new-day, none)
('upgrade-to-regular, none)                   *** client C1 is upgraded
('new-day, none)
('pay-debt, ('Amnt <- 100))                   *** client C1 pays 100$
```

Comments on the right side explain the sequence of rules defining the strategy. Basically, the execution trace specified consists of client `C1` renting two cars, one of which is returned on time and the other one late. After the second car is returned, the client is suspended for being late in his payments. However, this does not prevent him from being upgraded. The client then pays part of his debt. Note how the passage of time is modeled by the application of the rule `new-day`.

Now, in order to execute the system specifications using such a strategy, we just need to use a function (`rewSeq`) that takes a module, a term, and the list of pairs that define the strategy, and applies the rules in the strategy sequentially, using their corresponding partial substitutions. The actual way to rewrite the above initial configuration *VStoreConf* following the strategy *VStoreStrat* is as follows.

```
rewrite rewSeq(V-STORE,
               VStoreConf,
               VStoreStrat).
```

where `V-STORE` is the name of the module containing the specification of the *VStore* system. The configuration resulting from the above rewrite is the following.

```
< O1 : VStore |
   discounts :
    (((CasualCust, MidSizeCar), 0)
     ((CasualCust, FullSizeCar), 0)
     ((RegularCust, MidSizeCar), 20)
     ((RegularCust, FullSizeCar), 30)),
   rates     :
    ((MidSizeCar, 150)
     (FullSizeCar, 200)),
   payments  : ('C1, 600),
                      *** C1 has paid 600
   suspended : 'C1,  *** C1 is suspended
   threshold : 450,
   penalty   : 20,
   customers : 'C1 'C2,
   calendar  : 'C,
   cars      : 'A1 'A3 'A5 >
< C1 : RegularCust | cash : 4400,
   debt : 140 >     *** 140$ debt
< C2 : CasualCust |
   cash : 5000, debt : 0 >
< A1 : EconomyCar | available : true >
< A3 : MidSizeCar | available : true >
< A5 : FullSizeCar | available : true >
< C : Calendar | date : 8 >
                       *** 8 days later
```

We can see in this configuration how 8 days have passed, after which the client `C1` has been upgraded to regular—it is now an object of the `RegularCust` class—and is suspended. The client `C1` has paid a total of \$600 ($2 \times 150 + 200 + 100$), and has still a debt of \$140 ($200 + 20\%(200) - 100$).

This simple application of the rules may not have much interest by itself, but shows how it can be used for building more interesting strategies. For example, we may also find all possible one-step rewrites of a term using one by one the rules in a module on this term. Once we have this, we may find all possible computations for an initial state, or we may be interested in studying the weaknesses of the system, etc. The full expansion tree of the execution model of a system may also be built in that way, which is a possible approach for model-checking the system. Reachability analysis can also be realized in that way, exploring the execution tree of a system checking for given states. However, the Maude model checker [10] offers far more possibilities that these kinds of strategies for exploring the execution tree of a system, and with a greater performance.

7. Related work

Formal description techniques are being extensively employed in ODP and have proved valuable in supporting the precise definition of the reference model concepts [5]. 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 [24,25] use rewriting logic to formalize the computational model of RM-ODP. In Ref. [24], a formal operational semantics of the ODP computational model is presented, which is extended in Ref. [25] to deal with reflection and Quality of Service (QoS) contracts using failures. Although Najm does not consider executability issues for his specifications, their formalization may be directly specified in Maude and then executed following our proposal.

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 enterprise specifications [27]. 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 forthcoming 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 presents some shortcomings from our point of view:

(1) In the Object-Z approach, 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.
(2) Analogously, Maude 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), which cannot be represented so easily in Object-Z.
(3) 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 (cf. Ref. [27]), which is not directly accessible from the Object-Z specifications.
(4) 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 overspecifications in many cases.
(5) Another disadvantage of the use of Object-Z appears when modeling enterprise roles by Object-Z classes, which is the natural way of doing so in that approach. 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 objects cannot change their roles during their lifetime. In some applications, models which assume fixed roles may be adequate; but in others, there may be a need to represent objects which fulfill 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.

(6) Maude offers far more tool support than Object-Z does. Even if some animation can be obtained with Object-Z, it does not reach the level that can be obtained with Maude's execution facilities and strategies. Additionally, tools for model checking, theorem proving, and other behavioral analysis of specifications are available [6].

(7) Other notations (such as Z, LOTOS, or CSP) have been proposed for other viewpoints. A common way of dealing with consistency between specifications written in different notations is by translating them into one single notation. For instance, in Ref. [4] 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 FDTs into Maude can be obtained [19,22].

An interesting line of work that may also be related to ours is the use of deontic logic [23] for specifying systems, using the theory of norms and normative systems to specify the policies that rule the behavior of a system. However, the paradoxes of this logic and its complexity may hinder its practical utility, as discussed in Refs. [14,27].

Finally, we would like to mention the Ponder declarative language for the specification of security and management policies of distributed systems. Being a policy-based language, it allows the realization of many of the ODP enterprise policy concepts [18], and it is "down to earth" and implementable. However, Ponder does not cover all the enterprise viewpoint concepts (just policies), and it is too low-level when compared to our approach. Nevertheless, both approaches could be somehow complementary if bridges between the Maude specifications and the Ponder language were defined, allowing the refinement of the Maude rules into implementable Ponder instructions, or the other way round, whenever this is possible.

8. 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, showing how to build enterprise specifications of systems using Maude concepts and rules. With them we do not only obtain a high-level enterprise description of the system, but also are in a position to formally reason about the specifications produced and to quick-prototype them.

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. Rewriting logic and Maude have also been proposed for specifying the information viewpoint [9] and the computational viewpoint [25], and we plan to study their adequacy for being used in the specification of the others. 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.

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

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 the same FDT, or use different formalisms for the different viewpoints and then translate them into such an FDT. It has been shown that rewriting logic 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 [22]. 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.

Finally, tool support is another essential issue. Maude's intuitive style for specifying classes, objects, and rules greatly simplifies the understandability of the specifications produced. Furthermore, the process shown here for writing the Maude enterprise specifications of a system does not require users to have a deep knowledge of rewriting logic. Thus, it is our belief that Maude specifications could provide a useful vehicle for allowing stakeholders of a system to easily share and discuss about its purpose, scope, and policies, i.e. its enterprise specifications. Having said that, we feel that some graphical tool support may also be needed for the adoption of our proposal. We have already mentioned that the strong correspondence between the UML model classes and the Maude classes allows an easy translation between both models. In this sense, we are currently investigating some kind of graphical representation of Maude rules in UML. This would allow the stakeholders of the system to use a more user-friendly graphical notation like UML to describe the system structure and policies, and then translate them into the corresponding Maude specifications. Future work include the provision of a more user-friendly interface for these tasks. This is an ambitious goal, but we feel that without this kind of tools, the use of formal methods will never materialize, hindering the important benefits that formal analysis of systems could bring to the software development process of business systems.

Acknowledgements

The authors would like to thank José Meseguer and Narciso Martí-Oliet for their comments on previous versions of this paper, and the anonymous referees for their insightful comments and suggestions which greatly helped improve the contents and readability of this paper. This work has been partially supported by Spanish Projects TIC2002-04309-C02-01, TIC2000-0701-C02, and TIC2001-2705-C03.

References

[1] J. Aagedal, Z. Milošević, ODP enterprise language: UML perspective, Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, IEEE Publishing, September 1999, pp. 60–71.

[2] C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, F. Olsen, Transformations and consistent semantics for ODP viewpoints, in: H. Bowman, J. Derrick (Eds.), Proc. of FMOODS'97, Canterbury, Chapman & Hall, 1997, pp. 371–386.

[3] X. Blanc, M.-P. Gervais, R.L. Delliou, Using the UML language to express the ODP enterprise concepts, Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, IEEE Publishing, September 1999, pp. 50–59.

[4] E.A. Boiten, H. Bowman, J. Derrick, P. Linington, M.W. Steen, Viewpoint consistency in ODP, Computer Networks 34 (3) (August 2000) 503–537.

[5] H. Bowman, J. Derrick, P. Linington, M.W. Steen, FDTs for ODP, Computer Standards & Interfaces 17 (September 1995) 457–479.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. Quesada, Maude: specification and programming in rewriting logic, Theoretical Computer Science 285 (August 2002) 187–243.

[7] M. Clavel, F. Durán, S. Eker, J. Meseguer, Building equational proving tools by reflection in rewriting logic, in: K. Futatsugi, A. Nakagawa, T. Tamai (Eds.), CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier, Amsterdam, 2000, pp. 1–31.

[8] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, Object-Oriented Development: The Fusion Method, Prentice Hall, 1994.

[9] F. Durán, A. Vallecillo, Specifying the ODP information viewpoint using Maude, in: H. Kilov, K. Baclawski (Eds.), Proceedings of the Tenth OOPSLA Workshop on Behavioral

Semantics, Florida, Northeastern University, October 2001, pp. 44–57.

[10] S. Eker, J. Mesegur, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gaducci, U. Montanari (Eds.), Proc. of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002), Pisa, Italy, Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier, September 2002, pp. 115–142.

[11] K. Havelund, G. Roşu, Monitoring programs using rewriting, Proc. of Automated Software Engineering 2001 (ASE'01), California, IEEE CS Press, November 2001, pp. 135–143.

[12] 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.

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

[14] A. Jones, M. Sergot, On the Characterization of Law and Computer Systems: the Normative Systems Perspective, Wiley, 1993.

[15] P. Kosiuczenko, M. Wirsing, Timed rewriting logic with an application to object-based specification, Science of Computer Programming 28 (2–3) (1997) 225–246.

[16] P. Linington, RM-ODP: the architecture, in: K. Milosevic, L. Armstrong (Eds.), Open Distributed Processing II, Chapman & Hall, February 1995, pp. 15–33.

[17] P. Linington, Options for expressing ODP enterprise communities and their policies by using UML, Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, IEEE Publishing, September 1999, pp. 72–82.

[18] E. Lupu, M. Sloman, N. Dulay, N. Damianou, Ponder: realising enterprise viewpoint concepts, Proceedings of 4th International Enterprise Distributed Object Computing Conference (EDOC 2000), Japan, IEEE Publishing, September 2000, pp. 66–75.

[19] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, Technical report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993, To appear in: D.M. Gabbay (Ed.), Handbook of Philosophical Logic, Kluwer Academic Publishers.

[20] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.

[21] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, vol. 1376, Springer-Verlag, 1998, pp. 18–61.

[22] J. Meseguer, Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems, in: S.F. Smith, C.L. Talcott (Eds.), Proc. of FMOODS'2000, Stanford, CA, Kluwer Academic Publishers, Boston, September 2000, pp. 89–117.

[23] J.J. Meyer, R.J. Wieringa (Eds.), Deontic Logic in Computer Science: Normative System Specification, Wiley, New York, 1993.

[24] E. Najm, J.-B. Stefani, A formal operational semantics for the ODP computational model, Computer Networks and ISDN Systems 27 (1995) 1305–1329.

[25] E. Najm, J.-B. Stefani, Computational models for open distributed systems, in: H. Bowman, J. Derrick (Eds.), Proc. of FMOODS'97, Canterbury, Chapman & Hall, 1997, pp. 157–176.

[26] P.C. Ölveczky, J. Meseguer, Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems, in: K. Futatsugi (Ed.), Proceedings of the 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00), Electronic Notes in Theoretical Computer Science, vol. 36, Elsevier, 2000, pp. 361–383.

[27] M.W. Steen, J. Derrick, ODP enterprise viewpoint specification, Computer Standards & Interfaces 22 (September 2000) 165–189.

[28] A. van Lamsweerde, R. Darimont, E. Letier, Managing conflicts in goal-driven requirements engineering, IEEE Transactions on Software Engineering 24 (11) (November 1998) 908–926.

Francisco Durán is an Associate Professor at the Department of Computer Science of the University of Málaga, Spain. He received his MSc and PhD degrees in Computer Science from the same University in 1994 and 1999, respectively. He is one of the developers of the Maude system, and his research interests deal with the application of formal methods to software engineering, including topics, such as reflection and metaprogramming, component-based software development, open distributed programming, and software composition.

Antonio Vallecillo has BSc and MSc degrees in Mathematics, and a PhD degree in Computer Science. Most of his professional experience comes from the computer industry, where he has worked for more than ten years for several international companies, both in Spain and in the UK. Since 1996, he has been working at Málaga University, where he is currently an Associate Professor. His research interests include component-based software development, open distributed processing, and the industrial use of formal methods. He is the representative of the Málaga University at ISO and OMG, and a member of several professional organizations, including AENOR (the Spanish National Body for Standardization), the ACM, and the IEEE.