Specifying the ODP Information Viewpoint using Maude

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

Abstract

Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In this paper we explore the possibility of using Maude as a formal notation for writing and reasoning about RM-ODP information specifications.

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 participants to observe a system from a suitable perspective, and at a suitable level of abstraction [14].

The viewpoints are sufficiently independent to simplify reasoning about the complete specification. 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. The use of the object paradigm provides abstraction and encapsulation, two important properties for the specification and design of complex systems.

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. In any case, certain need to develop appropriate notations for ODP specifications has been identified by different authors [1, 4, 14, 18, 12] in order to increase the applicability of the ODP framework. For example, formal description techniques (FDTs) such as LOTOS or SDL have been proposed for the computational viewpoint [12], and Object-Z for the information and enterprise viewpoints [20].

The *information* viewpoint is concerned with information modeling. An information specification defines the semantics of information and the semantics of information processing in an ODP system, without having to worry about other system considerations, such as particular details of its implementation, or the technology used to implement the system. This viewpoint distinguishes between instantaneous views of information (static schemas), statements of information which is necessarily unchanged by the system (invariant schemas), and the description of information reflecting the behavior and evolution of the system (dynamic schemas).

Z and Object-Z have been traditionally considered as highly appropriate notations for information modeling. Static, dynamic and invariant schemas can be directly mirrored into Z and Object-Z specifications. Moreover, Object-Z provides a good basis for relating specifications in other viewpoints and languages [3]. On a different arena, UML or Fusion [6] are also well suited for ODP information modeling. Although they are not really formal, they have been used because of their appealing graphical syntax and because they are part of fully integrated development methodologies (see, e.g., [19, 2, 13]).

In this paper we would like to explore another alternative for specifying the information viewpoint. We propose Maude, an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems [17]. Maude has also been used for modeling the ODP enterprise viewpoint, and our main contribution in this paper is to show how Maude, and in particular rewriting logic and its underlying *membership equational logic* [16], provides the expressiveness required for modeling the information viewpoint in a clean and clear manner.

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

2 The Information Viewpoint

The information viewpoint is concerned with information modeling. It focuses on the semantics of information and information processing. By factoring an information model out of the individual system components, it provides a common view which can be referred to by the specifications of information sources and sinks and the information flows between them.

In general, the information language helps answer the questions "what kind of information is managed by the system?" and "what constraints and criteria need to be applied to access the information?".

In the ODP Reference Model [12], prescription in the information viewpoint is restricted to a small basic set of concepts and structuring rules. The three basic concepts are the following:

- **Invariant schema**: A set of predicates on one or more information objects which must always be true. The predicates constrain the possible states and state changes of the objects to which they apply.
- **Static schema**: A specification of the state of one or more information objects, at some point in time, subject to the constraints of any invariant schemata.
- **Dynamic schema**: A specification of the allowable state changes of one or more information objects, subject to the constraints of any invariant schemata.

An information specification defines the semantics of information and the semantics of information processing in an ODP system in terms of a configuration of information objects, the behavior of those objects, and environment contracts for the objects in the system.

- An information object template references static, invariant and dynamic schemata. The relationships between information objects can be modeled as part of the state of those information objects.
- Information objects are either atomic or are represented as a composition of component information objects. When an information object is a composite object, the schemas are composed as well.

- Allowable state changes specified by a dynamic schema can include the creation of new information objects and the deletion of information objects involved in the dynamic schema. Allowable state changes can be subject to ordering and temporal constraints.

- Finally, the configuration of information objects is independent from distribution, i.e., there is no sense or focus on distribution is this viewpoint. Likewise, information object interfaces cannot themselves be reference points; there is no commitment to the interfaces appearing in an implementation.

3 Rewriting Logic and Maude

Maude [17] 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 [16]. 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 [17].

Rewriting logic [15] 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` $C \mid a_1 : S_1, ..., a_n : S_n$", where C is the name of the class, a_i are attribute identifiers, and S_i are the corresponding sorts of the attributes. Objects of a class C are then record-like structures of the form `<` $O : C \mid a_1 : v_1, ..., a_n : v_n$ `>`, where O is the name of the object, and the v_i are the current values of its attributes. Objects can interact in a 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* between some objects and messages. The general form of such rewrite rules is

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

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 '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.

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. Thus, such rules are called *synchronous*, while rules involving just one object and one message in their left-hand side 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 [17].

On the other hand, 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. Such a 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. Conditional membership equations are written as "t`:`S `if` P", stating that a given term t has a sort S if a certain boolean condition P holds.

4 Writing Information Specifications in Maude

In general, ODP systems are modeled in terms of objects. An object is a representation of an entity in the real world; it contains information and offers services. A system is therefore composed of interacting objects.

In the case of the information viewpoint we talk about *information objects*, which model the entities defined in an information specification. Information objects are abstractions of entities that occur in the real world, in the ODP system, or in other viewpoints [12]. Basic information elements are represented by atomic information objects. More complex information is represented as composite information objects expressing relationships over a set of constituent information objects. An ODP system can be represented in terms of information objects and their relationships.

An information specification is composed of specifications of the elements previously defined in Section 2. That is, it consists of the specification of

(a) a configuration of information objects,

(b) the behavior of those objects, and

(c) environment contracts for the objects in the system.

In RM-ODP, the environment of an object is the part of the model which is not part of that object. The environment contract is an agreement between an object and its environment. It states two-way expectations and constraints by one on the other. It may well include expectations of the objects in the system from their environments, such as quality of service statements, availability of resources, etc.

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

Information objects are modeled by Maude objects. In Maude, each object belongs to a class, which is defined by a set of attributes (of certain sorts), the specification of the signature of the operations accepted by the object, and a set of rules that define the behavior and allowable state changes of the object. Thus, **roles** that objects play in the information specification are modeled by Maude classes, whose members are the objects exhibiting a behavior compatible with the ones identified by each 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 that role. A role A that specializes other role B is modeled by class `A` inheriting from class `B`.

Each **relationship among roles** is modeled by a class with the name of the relationship as its name, and whose attributes are the identifiers of the participants plus the relationship's attributes.

Communities are compositions of information objects, and therefore are naturally modeled by Maude's *configurations*, which are multisets of objects and messages.

Although there is no notation for **invariant schemas** in Maude, there are several ways of modeling them. Basically, an invariant is a predicate that a specification always requires to be true, and that may restrict the possible behaviors given in such a specification. One way of modeling them is by using Maude's membership equational logic [16]. 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. For example,

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

Note that this approach is completely systematic and therefore may be automated easily, although doing it by hand we have a much greater flexibility. For example, we may have a much more complex sort structure, and different membership axioms or conditions on the rules.

Static schemas will be directly mirrored to Maude object specifications. A static schema was defined as the "specification of the state of one or more information objects, at some point in time, subject to the constraints of any invariant schemata" [12]. This specification of the state of the objects is precisely the one provided by the Maude specifications of these objects, while the guaranty that the invariant constraints are fulfilled is provided by the corresponding membership axioms.

Environment contracts have usually been modeled by observer objects, responsible for observing and arbitrating the collective behavior of configurations of objects, notifying faulty objects or violating behavior [10]. This is due to the fact that it is difficult to impose obligations on actions that are due to external agents of the system (e.g., a video server which does not provide the agreed quality or performance). In Maude, environment contracts admit a more homogeneous treatment, modeling them as normal rules that monitor the conditions of the contract, and detect possible violations of such conditions. Those '*watchdog*' rules will react to these violations and determine the appropriate corrective (penalty or incentive) actions.

Once the system specifications are written using this modeling approach, what we get is a rewrite logic specification of the system, which can be executed in Maude, thus simulating its behavior. This means that, in addition to being able to formally reason about such a system, we will have a running prototype of it. Maude implements a default top-down rule-fair strategy for the execution of rewrite systems, although it also provides the possibility of defining our own strategies for controlling the execution, which will allow us to carry out simulations, model checking, and other dynamic analysis of the system [17].

5 A Case Study

In order to illustrate our proposal we will specify in Maude a simple example, an extended version of one used by different authors to illustrate the use of Object-Z [20] and Maude [7] for the specification of the enterprise viewpoint.

The example is based on a library, the Templeman Library at the University of Kent at Canterbury, but in this occasion we will concentrate on its information specification. For space reasons we do not include all the details required in a real library, just some of them: all academic staff, and postgraduate and undergraduate students of the university have borrowing rights; there are prescribed periods of loan and limits on the number of items allowed on loan to a borrower at any given time—for example, undergraduates may borrow 8 books, for four weeks, and may not borrow periodicals; borrowers who fail to return an item when it is due will become liable to a charge at the rates prescribed until the book or periodical is returned to the library; failure to pay charges may result in suspension by the librarian of borrowing facilities.

5.1 Informal Specification

We will start by an informal specification, and then illustrate how it can be formalized in Maude.

From the text above we can identify four main roles, namely **borrowers**, library **items**, **calendars** and **librarians**. There are three special kinds of borrowers (academic staff, undergrats, and postgrats), and two kinds of items (books and periodicals). A library community can be seen as composed by a calendar, one or more librarians, items, and borrowers. Borrowers may loan items, which may be represented as a `Loan` class. Figure 1 shows a UML class diagram with the structure of the information specification of the library, in terms of the roles that the information objects may play in the community. Note that, in this case, each information object fulfills only one role, although we might consider, for example, academics or students acting as librarians just by introducing some additional inheritance relations. The use of the UML class diagrams for the description of the structure of the communities has been proposed by different authors, e.g. [13, 20].

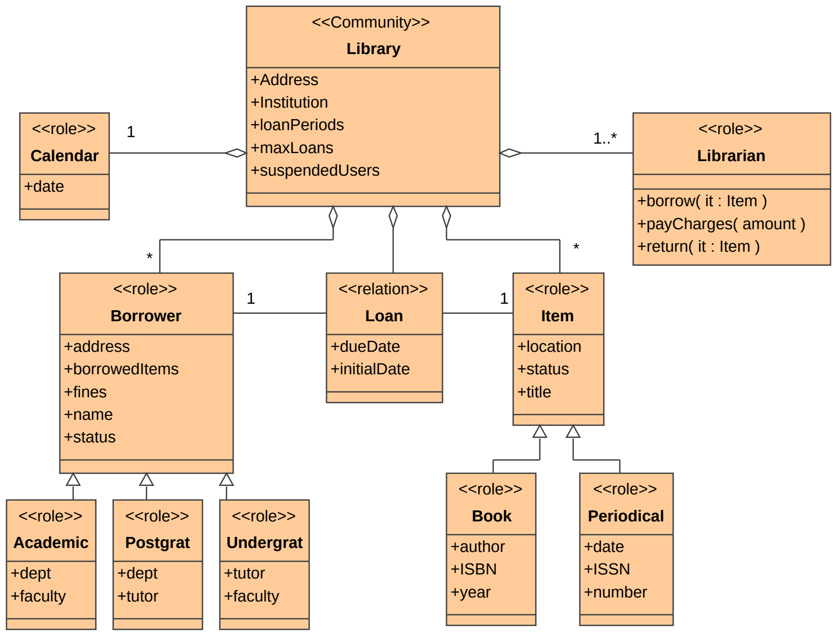

Figure 1: Structure of the library community.

The attributes and operations of each class define the information to be captured by this information specification. Apart from the obvious attributes (title, author, and ISBN of books; or name and address of borrowers), other attributes need some explanation:

- The prescribed loan periods and limits on the number of items allowed on loan to a borrower at one given time are given, respectively, by attributes `maxLoans` and `loanPeriods` of the class `Library`.

- Borrowers who fail to return an item when it is due may be fined until the book or periodical is returned to the library. The amount of their current fines, together with the number of borrowed items, is kept, respectively, in the attributes `fines` and `borrowedItems` of class `Borrower`.
- Borrowers may be suspended by the librarian if they do not pay their fines. This information is stored by attributes `status` of class `Borrower` and `suspendedUsers` of class `Library`. Of course, the values of these attributes should be kept consistent in the system. This constraint will be considered as an invariant in Section 5.3.
- The attribute `status` of class `Item` defines the status of an item, with possible values "on loan", "free", or "disposed", if the book has been destroyed or thrown away. Attribute `location` of class `Item` describes the location of the corresponding item when it is in the library.
- Finally, we have defined operations `borrow`, `return` and `payCharges` in the `Library` objects to illustrate how operations are modeled in our approach.

The way the library works (from the perspective of the information viewpoint) needs to be defined in terms of how the information is processed. Invariant, static and dynamic schemas are the mechanisms defined for that purpose.

Again, for space reasons we are not concerned in this paper about providing the complete set of schemas that define the behavior of the library, but on selecting a representative collection of schemas that illustrate the use of Maude to model them.

5.2 Static Schemas: Modeling the System Structure with Maude

The strong correspondence between the UML model classes and the Maude classes allows an easy translation between both models. This process is 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 first, and then translate it into Maude classes. Special care should be taken here, since the semantics of UML is often weak and imprecise, as opposed to the semantics of Maude. An automatic translation of UML class diagrams into Maude, inside a more ambitious project trying to give semantics to UML, has been proposed in [21].

In our example, the borrower role is modeled by the `Borrower` class.

```
sort BorrowerStatus .
ops suspended released : -> BorrowerStatus .
class Borrower | address : String, borrowedItems : Nat, fines : Money,
                 status: BorrowerStatus, name : String .
```

Please note how we make use of Maude facilities for defining abstract data types, in this case an enumerated type that represent the status of a borrower, which has two values: `suspended` and `released`. The subclasses of `Borrower` are naturally defined by inheritance:

```
class Academic  | dept : String, faculty : String .
class Postgrat  | dept : String, tutor : String .
class Undergrat | tutor : String, faculty : String .
subclasses Academic Undergrat Postgrat < Borrower .
```

Class `Item` and its subclasses are defined in a similar way.

```
sort ItemStatus .
ops on-loan free disposed : -> ItemStatus .
class Item | location : String, status: ItemStatus, title : String .
```

The subclasses of `Item` are naturally defined by inheritance:

```
class Book | author : String, ISBN : String, year : Nat .
class Periodical | date : Date, ISSN : String, number : Nat .
subclasses Book Periodical < Item .
```

A library role is modeled by a class containing the information concerning the institution to which the library belongs, its address, and the deadlines and number of items that a borrower is allowed to borrow simultaneously.

```
class Library | institution : String, address : String,
                loanPeriod : PFun(Tuple(Cid, Cid), Nat),
                maxLoans : PFun(Cid, Nat), suspendedUsers : Set(Oid) .
```

Partial functions are used for specifying the `loanPeriod` and `maxLoans` attributes. They associate the appropriate values to each of the different actors involved, with an operator `_[_]` which takes a partial function and a domain element and returns its image, or a default value if it has no associated image[1]. Thus, class `Library` has an attribute `maxLoans`, of sort `PFun(Cid, Nat)`, that returns the maximum number of items allowed to borrow for a given class name (e.g., 24 for an academic, etc.). This function is not defined for a class name other than `Academic`, `Undergrat`, and `Postgrat`. Similarly, there is a `loanPeriod` attribute such that, given the names of a borrower class and an item class, it gives back the number of days that a borrower is allowed to hold an item (e.g., 7 days for an academic to have a periodical, etc.). Library objects also have an attribute `suspendedUsers`, which is a set with the identifiers of the borrower objects who are suspended.

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

In the example, class `Librarian` does not have any attributes, but it offers three operations. In our approach, Maude messages will be used for modeling method definitions, which produces the following specification:

```
class Librarian .
msgs borrow return : Oid Oid Oid -> Msg .
msg payCharges : Oid Money Oid -> Msg .
```

Each message will be named after the operation it models, and the first and last arguments of the message will be the identifiers of the called and caller objects, respectively. The rest of the arguments correspond to the arguments declared for the operation. In the case of messages `borrow` and `return`, the middle argument is the identifier of the item being referenced. In the case of message `payCharges`, it is the amount to be paid.

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

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

[1]Maude supports user-definable mixfix syntax, with underscores '_' indicating the position of the different arguments.

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

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

5.3 Invariant Schemas in Maude

Once we have specified the static schemas that determine the structure of the system, let us formally specify in this section some invariant schemas. For illustration purposes we will consider the following ones:

I1. There should be unique `Library` and `Calendar` objects in the system.

I2. The number of `Loan` objects should be equal to the sum of the values of the `borrowedItems` attributes of all `Borrower` objects in the system.

I3. The list `suspendedUsers` of the `Library` object should contain exactly the identifiers of all `Borrower` objects whose `status` attribute is equal to `suspended`.

The invariant schema I1 limits the number of objects of a given class in the system (i.e., the number of objects that may simultaneously fulfill the given roles). The following predicate `thereIsOneAndOnlyOne` decides whether there is a unique object of the given class in a given configuration.

```
op thereIsOneAndOnlyOne : Cid Configuration -> Bool .
op numberOfOccurrences : Cid Configuration -> Nat .
var M : Message . var Ats : Attributes . var O : Oid . vars A A' : Cid .

eq thereIsOneAndOnlyOne(A, C) = numberOfOccurrences(A, C) == 1 .
eq numberOfOccurrences(A, < O : A' | > C) = if < O : A' |> : A
                                            then 1 + numberOfOccurrences(A, C)
                                            else numberOfOccurrences(A, C) fi .
eq numberOfOccurrences(A, M C) = numberOfOccurrences(A, C) .
eq numberOfOccurrences(A, none) = 0 .
```

The specification of the predicate for the invariant I2, which states that the number of `Loan` objects should be equal to the sum of the values of the `borrowedItems` attributes of all `Borrower` objects in the system, assuming the declarations above, may be written as follows:

```
op consistentNumberOfLoans : Configuration -> Bool .
op borrowedItemsSum : Configuration -> Nat .
var N : Nat .
eq  consistentNumberOfLoans(C) = numberOfOccurrences(Loan, C) == borrowedItemsSum(C) .
eq  borrowedItemsSum(< O : Borrower | borrowedItems : N > C) = N + borrowedItemsSum(C) .
ceq borrowedItemsSum(< O : A | > C) = borrowedItemsSum(C) if not < O : A |> : Borrower .
eq  borrowedItemsSum(M C) = borrowedItemsSum(C) .
eq  borrowedItemsSum(none) = 0 .
```

Invariant I3, which states that the list `suspendedUsers` of the `Library` object should contain exactly the identifiers of all `Borrower` objects with `suspended status`, may be specified in a similar way, assuming by I1 that there is one and only one library object:

```
op consistentSuspendedUsers : Configuration -> Bool .
eq consistentSuspendedUsers( < O : Library | suspendedUsers : O' OS >
                             < O' : Borrower | status : BS > C )
   = (BS == suspended) and
      consistentSuspendedUsers(< O : Library | suspendedUsers : OS > C) .
eq consistentSuspendedUsers( < O : Library | suspendedUsers : none >
                             < O' : Borrower | status : BS > C )
   = (BS =/= suspended) and consistentSuspendedUsers(< O : Library | > C) .
ceq consistentSuspendedUsers(< O : Library |> < O' : A | > C)
   = consistentSuspendedUsers(C) if not < O : A |> : Borrower .
eq consistentSuspendedUsers(M C) = consistentSuspendedUsers(C) .
eq consistentSuspendedUsers(< O : Library | suspendedUsers : OS >) = OS == none .
```

Although this is not necessarily the case for all invariants, there may be cases in which we need to establish a condition on the whole system, and not only on an arbitrary part of it. This happens for instance when requiring the uniqueness of the object identifiers in a system. To be able to capture the whole configuration of objects and messages, we may define an operation "`{_}`":

```
op {_} : Configuration -> WholeConfig .
```

as a constructor of sort `WholeConfig`. Thus, we may discriminate between the whole configuration and any part of it, and apply rules and equations to just the whole, or to any of its parts, depending on our needs. With this, the invariant may be specified by the following membership equation:

```
var C : Configuration .
subsort ValidLibrary < WholeConfig .
cmb {C} : ValidLibrary if thereIsOneAndOnlyOne(Calendar, C)
                       and thereIsOneAndOnlyOne(Library, C)
                       and consistentNumberOfLoans(C)
                       and consistentSuspendedUsers(C) .
```

5.4 Dynamic Schemas: Description of the System's Behavior

The dynamic schemas describe the allowed state changes. In our particular case we have defined the following:

D1. On invocation of a `borrow` operation, a new `Loan` object is created, that relates both the item and the borrower issuing the operation. The borrower and item attributes `borrowedItems` and `status` are updated to reflect this change. The values of the loan attributes are calculated according to the date shown in the `Calendar` object, and the library policies on period limitations. Borrowers can only borrow items according to the limitations on the number of items and loan periods imposed by the library; suspended borrowers cannot borrow more books, and only "free" items can be borrowed.

D2. On invocation of a `return` operation, the corresponding `Loan` object is removed. The borrower and item attributes `borrowedItems` and `status` are updated to reflect this change. Only "on-loan" items can be returned.

D3. Fines are calculated when returning an item, adding an amount F for every overdue day.

For instance, the behavior of the `borrow` operation for books as described in D1 can be modeled in Maude as follows:

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

In the synchronous rule above 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. Borrowing a book needs such a book not to be on loan (a), the borrower object not to be suspended (b), and the number of its borrowed items be smaller than its allowance (c).

Note that 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 an equation or rule will maintain their previous values unmodified. Note as well the use of attributes `borrower` and `item` in loan objects, which make explicit that the loan relationship is between the borrower and the item specified by these attributes. Finally, the due date is obtained by adding to the current date the value assigned to the pair borrower-item in the partial function `LP` in the attribute `loanPeriod`.

In addition to the conditions required for the borrowing of a book, the borrowing of a periodical is only allowed if the borrower is not an undergrat. This may be specified by a similar rule.

The return of an item may be specified by the rule below. We can see how the `loan` object disappears in the right-hand side of the rule, and how fines are calculated according to D3.

```
rl [return] :
  return(O, I, B)
  < B : Borrower | borrowedItems : N, fines : M >
  < I : Item | >
  < A : Loan | borrower : B, item : I, dueDate : D >
  < O : Librarian | >
  < L : Library | >
  < C : Calendar | date : Today >
  => < B : Borrower | borrowedItems : N - 1, fines :  M + F * MAX(0, Today - D) >
     < I : Item | status : free >
     < O : Librarian | >
     < L : Library | >
     < C : Calendar | > .
```

Further rules for specifying suspension, release and payments of debts can be defined and specified in a similar way. They have been omitted in this paper for brevity, but the interested reader can find the specification of such rules together with additional details in a separate technical report [8].

6 Related Work

Formal description techniques are being extensively employed in ODP and have proved valuable in supporting the precise definition of reference model concepts [4]. Among all the works, we will focus here on those that try to formalize the information viewpoint, using different notations.

In the first place, in [1] Bernardeschi *et al.* use a class-oriented language for modeling the information viewpoint, which is used for defining a semantically consistent transformation between the information and the computational model—specified using the actors model.

In the same year, Dustzadeh and Najm [9] showed how OMT and Fusion support ODP information modeling, and provide a formal semantics for the object diagrams of these graphical application development methodologies. Along the same lines, many authors consider UML a well suited notation for ODP information modeling because of its appealing graphical syntax and because it is part of a fully integrated development methodology [19, 2, 13]. Moreover, it is widely known and accepted by all kinds of users. However, the loose semantics of UML is a major drawback if precise and unambiguous specifications are required.

Probably the most widely accepted notations for formalizing the information viewpoint are Z and Object-Z. Initially, Z was chosen because the schemas defined in the information viewpoint could be directly mirrored into Z schemas. However, Z is not object-oriented, does not allow modularity, and has some limitations for expressing invariants stating temporal logic properties of the system. Object-Z solves most of the Z drawbacks since it is object oriented, allows modularity, and incorporates a subset of temporal logic for expressing class invariants. Therefore, it became the best candidate language for formalizing the information viewpoint. However, the use of Object-Z for specifying the information viewpoint may also present some shortcomings:

- First, operations in dynamic schemas are assigned to just one object, and included as operations in the object's definition class. How to deal with operations in which there is more than a principal object (e.g., in the case of synchronous actions)? In our approach operations are modeled by messages and rules, and therefore are first-class citizens.
- The treatment of invariant and contract violations is not homogeneous with the rest of the specifications. Violations are not (and cannot be) specified within the Object-Z framework, but at the meta-level [20], which is not easily accessible from the Object-Z specifications.
- Another disadvantage of the use of Object-Z appears when modeling the roles that information objects play in an information specification. Roles are modeled by Object-Z classes, which is the natural way of doing it. This has the initial advantages that information about each role can be encapsulated, and that roles can be composed. However, it has the disadvantage that roles are thereby associated with fixed classes of objects, so that an object cannot change its role during its lifetime. In some applications, models which assume fixed roles may be adequate; but in others, there may be a need to represent objects which play different roles at different times, as it happens for instance in dynamically configurable networks. Again, this is not an issue in Maude, since the class of an object can be changed.
- Other notations (such as Z, LOTOS, or CSP) are used in other viewpoints, because Object-Z does not deal with all aspects. A common way of dealing with consistency between specifications written in different notations is by translating them into one single notation. For instance, in [3] 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.

On the other hand, Object-Z provides notation for the specification of invariants defined by both first-order and temporal logic predicates. Here we have described a possible way for modeling invariants in Maude, focusing on the specification of invariants defined by first-order logic predicates. However, other kinds of invariants—such as temporal logic invariants—may be required in some situations. Although we have not considered such a possibility in this paper, Fiadeiro *et al.* have shown [11] that it is possible to infer and to reason about modal and temporal logic properties of a system specified in rewriting logic.

7 Concluding Remarks

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

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

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 [17]. 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, another alternative for specifying invariants in Maude is through the use of its reflective capabilities, and in particular through the use of strategies for controlling the execution. This may be probably the most general and powerful way of specifying invariants. By using strategies we can both check that invariants are satisfied and restrict the transitions in the system. Wirsing and Knapp also propose the use of strategies for restricting the behavior of object-oriented systems with process expressions [22]. The use of strategies for expressing invariant schemas is the next step in our research, as well as the development of support tools for the construction of Maude specifications of the different viewpoints.

References

[1] C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and consistent semantics for ODP viewpoints. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, Canterbury, 1997. Chapman & Hall.

[2] X. Blanc, M.-P. Gervais, and R. L. Delliou. Using the UML language to express the ODP enterprise concepts. Technical Report LIP6 1999-024, Laboratoire d'Informatique de Paris 6, Paris, France, November 1999.

[3] E. A. Boiten, H. Bowman, J. Derrick, P. F. Linington, and M. W. Steen. Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537, August 2000. Available at `http://www.cs.ukc.ac.uk/pubs/2000/1026`.

[4] H. Bowman, J. Derrick, P. F. Linington, and M. W. A. Steen. FDTs for ODP. *Computer Standards and Interfaces*, 17:457–479, September 1995. Available at `http://www.cs.ukc.ac.uk/pubs/1995/178`.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. To appear in Theoretical Computer Science, 2001.

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

[7] 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. An extended version is available as technical report at `http://www.lcc.uma.es/~av/Publicaciones/01/ITI-2001-8.pdf`.

[8] F. Durán and A. Vallecillo. Writing ODP Information specifications in Maude. Technical Report ITI-2001-10, Departmento de Lenguajes y Ciencias de la Computación, University of Málaga, Aug. 2001. Available at `http://www.lcc.uma.es/~av/Publicaciones/01/ITI-2001-10.pdf`.

[9] J. Dustzadeh and E. Najm. Consistent semantics for ODP information and computational models. In *Proc. of FORTE/PSTV'97*, Osaka, Japan, Nov. 1997. Chapman & Hall.

[10] A. Février, E. Najm, and J.-B. Stefani. Contracts for ODP. In *Proc. of the 4th AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software*, Mallorca, Spain, May 1997.

[11] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. Mosses, editors, *Proceedings of WADT'99*, volume 1827 of *LNCS*, pages 438–458. Springer-Verlag, 2000.

[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/ITU-T, 1997.

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

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

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

[16] 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.

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

[18] E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, Canterbury, 1997. Chapman & Hall.

[19] J. Oyvind Aagedal and Z. Milošević. ODP enterprise language: UML perspective. In *Proceedings of 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*. IEEE Publishing, 1999.

[20] M. W. Steen and J. Derrick. ODP Enterprise Viewpoint Specification. *Computer Standards and Interfaces*, 22:165–189, September 2000. Available at `http://www.cs.ukc.ac.uk/pubs/2000/1122`.

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

[22] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. To appear in Theoretical Computer Science, 2001.