Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Standards & Interfaces 27 (2005) 597–620

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Using Maude to write and execute ODP information viewpoint specifications

Francisco Durán, Manuel Roldán, Antonio Vallecillo*

Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, ETSI Informatica, Campus de Teatinos, 29071, Malaga, Spain

Received 6 October 2004; accepted 22 October 2004
Available online 18 November 2004

Abstract

The aim of the open distributed processing (ODP) information viewpoint is to describe the semantics of the information and of the information processing in a system, from a global point of view, without having to worry about other considerations, such as how the information will be finally distributed or implemented or the technology used to achieve such implementation. Although several notations have been proposed to model this ODP viewpoint, they are not expressive enough to faithfully represent all the information concepts, or they tend to suffer from a lack of (formal) support, or both. In this paper, we explore the use of Maude as a formal notation for writing ODP information specifications. Maude is an executable rewriting logic language especially well suited for the specification of object-oriented open and distributed systems. We show how Maude offers a simple, natural, and accurate way of modeling the ODP information viewpoint concepts, allows the execution of the specifications produced, and offers good tool support for reasoning about them.

Keywords: Rewriting logic; Maude; RM-ODP; ODP information viewpoint; Collective behavior; Invariants

1. Introduction

One of the common ways of dealing with the inherent complexity of specifying distributed systems is by dividing the design activity into a number of areas of concern, each one dealing with a specific aspect of the system. Current software architectural practices define several distinct viewpoints of systems to accomplish such specification decomposition. Examples include the viewpoints described in IEEE standard 1471 [20], the "4+1" view model [26], the Zachman's framework [37], or the Open Distributed Processing (ODP) Reference Model [22]. In particular, we are interested in the Reference Model of Open Distributed Processing (RM-ODP) framework, which provides five generic and complementary viewpoints on the system and its environment: *enterprise, information, computational, engineering*, and

* Corresponding author. Tel.: +34 9 5213 2794; fax: +34 9 5213 1397.
E-mail addresses: duran@lcc.uma.es (F. Durán), mrc@lcc.uma.es (M. Roldán), av@lcc.uma.es (A. Vallecillo).

0920-5489/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2004.10.008

technology. They enable different abstraction viewpoints, allowing participants to observe a system from different suitable perspectives [27]. One of the main benefits of RM-ODP is that, as opposed to other approaches, it provides precise definitions of a system of interrelated concepts rather than some, often imprecise, descriptions of isolated ones.

The ODP 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. Although separately specified, the viewpoints are not completely independent; key items in each of them are identified as related to items in the other viewpoints. Furthermore, all RM-ODP viewpoints have a common foundation defining concepts used in all of them (and therefore, for example, the information viewpoint may and should use such foundational concepts as composition, type, subtype, etc.).

One of these viewpoints, the information viewpoint, is concerned with information modeling. An information specification defines the semantics of information and 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 schemata), statements about the information that must always hold (invariant schemata), and the description of information reflecting the behavior and evolution of the system (dynamic schemata).

To represent the different viewpoints, the ODP reference model provides abstract languages for the relevant concepts of each viewpoint. However, such languages are abstract, in the sense that they define what concepts should be supported, but not how they should be represented. Several notations have been proposed for the different viewpoints by different authors, which nevertheless seem to agree on the need to represent the semantics of the ODP viewpoints concepts in a precise manner [1,4,22,24,25,27,33]. For example, formal description techniques such as LOTOS and SDL have been proposed for the computational viewpoint [22], and Z and Object-Z for the information and enterprise viewpoints [36]. Object-oriented modeling languages such as UML or Fusion [8] have also been proposed for ODP information and enterprise modeling.

Z and Object-Z have been traditionally considered as highly appropriate notations for information modeling, since static, dynamic, and invariant schemata can be directly mirrored into Z and Object-Z specifications [25]. Furthermore, Object-Z is object oriented, as ODP is, and provides a good basis for relating specifications in other viewpoints [2]. On a different arena, UML and Fusion [8] are also well suited for ODP information modeling. Although they are not formal, they have been used because of their appealing graphical syntax and because they are part of fully integrated development methodologies, such as RUP [23] or Catalysis [9]. However, their loose semantics represent an impediment for achieving the precise specification and analysis of systems. In particular, the semantics of some essential UML constructs (especially those describing relationships and types) is not well defined, and thus, the understanding of UML specifications has to rely on tacit assumptions, which may be different for different writers and readers of the specifications.

In this paper, we explore a new alternative for specifying the information viewpoint. We propose Maude [5], an executable formal language based on rewriting logic specially well suited for the specification of object-oriented open and distributed systems [31]. Maude has already been used for modeling some of the ODP viewpoints: A proposal for modeling the enterprise viewpoint was presented in Ref. [13], a first attempt for modeling the information viewpoint in Ref. [14], and an approach for specifying the computational viewpoint has been recently proposed in Ref. [34]. Rewriting logic has also been used by Najm and Stefani [32,33] to formalize the computational viewpoint. In this paper, we shall show how rewriting logic and its underlying membership equational logic [30] and, in particular, Maude provide the expressiveness required for modeling the ODP information viewpoint.

As we shall see, this choice not only offers new benefits over the previous approaches for formalizing ODP information specifications (and, in particular, over the Object-Z approach) but also allows to overcome some of their limitations. Moreover, 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. The use of Maude provides additional advantages: The fact that rewriting logic specifications are executable will allow us to apply a flexible range of increasingly stronger formal analysis methods and tools, such as run-time verification [18], model checking [16], or theorem proving [7]. Maude offers a comprehensive toolkit for automating such kinds of formal analysis of specifications. Furthermore, the reflective capabilities of Maude will be used to drive the system execution according to the dynamic and invariant schemata defined for the system, as we will also show.

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 for writing information specifications in Maude. Section 5 is dedicated to a case study that illustrates our approach. Finally, Section 6 compares our work to other similar approaches, and Section 7 draws some conclusions and describes some future research activities.

2. The information viewpoint

The information viewpoint is concerned with information modeling. By factoring an information model out of the individual system components, it provides a common view that can be referred to by the rest of the specifications. 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 [22], 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 that 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. Other considerations about these schemata include the following:

- 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.
- The configuration of information objects is independent from distribution; that is, there is no sense or focus on distribution is this viewpoint.

3. Rewriting logic and Maude

Maude [5,6] is a high-level language and a high-performance interpreter and compiler in the OBJ [17] algebraic specification family that supports membership equational logic and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute three million rewriting steps per second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments of various logics, models of computation, theorem provers, or even programming languages. We informally describe those Maude features necessary for understanding the paper in this section; the interested reader is referred to its manual [6] for more details.

Rewriting logic [29] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational

theory describing its set of states as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E), and R is a collection of rewrite rules. Maude's underlying equational logic is membership equational logic [30], 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 the definition of partial functions with equationally defined domains.

For example, the following Maude functional module defines the natural numbers (with sorts `Nat` of natural numbers and `NzNat` of nonzero natural numbers), using the Peano notation, with the zero (`0`) and successor (`s_`) operators as constructors (note the `ctor` attribute). Then, the addition operation (`_+_`) is defined, being its behavior specified by two equational axioms. The operators `s_` and `_+_` are defined using the *mixfix* syntax (underscores indicate places for arguments).

```
fmod MY-NAT is
 sort NzNat Nat .
 subsort NzNat < Nat .
 op 0 : -> Nat [ctor] .
 op s_ : Nat -> NzNat [ctor] .
 op _+_ : Nat Nat -> Nat
    [assoc comm] .
 vars M N : Nat .
 eq 0 + N = N .
 eq s M + s N = s (M + N) .
endfm
```

If a specification is confluent, terminating, and sort decreasing, then it can be executed. Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is found. Some equations, like the one expressing the commutativity property, are not terminating, but nonetheless they are supported by means of *operator attributes*. Maude performs simplification modulo the equational theories provided by such attributes, that can be associative (`assoc`), commutativity (`comm`), identity (`id`), and idempotence (`idem`). The above properties must therefore be understood in the more general context of simplification modulo such equational theories.

In Maude, specifications may be generic; that is, they may be defined with other specifications as parameters. The requirements that a data type must satisfy are described by *theories*. For example, lists can be constructed on top of any data, which means that its parameter could be a theory requiring only the existence of a sort.

```
fth TRIV is
 sort Elt .
endfm

fmod LIST(X :: TRIV) is
 sort List(X) .
 subsort X@Elt < List(X) .
 op nil : -> List(X) [ctor] .
 op _ _ : List(X) List(X) -> List(X)
    [ ctor assoc id: nil] .
endfm
```

`X::TRIV` denotes that `X` is the label of the formal parameter and that it must be instantiated with modules satisfying the requirements expressed by the theory `TRIV`. The sorts and operations of the theory are used in the body of the parameterized module, but sorts are qualified with the label of the formal parameter. Thus, in this case, the parameter `Elt` becomes `X@Elt` in the `LIST` module.

The way to express instantiations is by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations in the target module, in such a way that the induced axioms are provable in the target module. The following view `Nat` maps the theory `TRIV` to the predefined module `NAT` of natural numbers.

```
view Nat from TRIV to NAT is
 sort Elt to Nat .
endv
```

Then, the module expression `LIST(Nat)` denotes the instantiation of the parameterized module `LIST` with the above view `Nat`. Notice that the name of the sort `List(X)` makes explicit the label of the parameter. In this way, when the module is instantiated with a view, like, e.g., `Nat` above, the sort name is also instantiated, becoming `List(Nat)`.

For more information on parameterization and how it is implemented in the Maude system, the reader is referred to Ref. [6].

We illustrate the use of membership axioms with the following ORD-NAT-LIST module, in which we define ordered lists of natural numbers. In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of ordered lists, e.g., defined by a property over lists. Membership equational logic allows subsort definition by means of conditions involving equations and/or sort predicates. In the following example, we use this technique to define a subsort OrdNatList, containing sorted lists of natural numbers, of List(Nat). Notice the way the sort is defined: The empty and singleton lists are always ordered (first membership axiom and subsort declaration Nat < OrdNatList), and any other list is ordered if the first element is less than or equal to the second and the list without the first element is also ordered (last membership axiom).

```
fmod ORD-NAT-LIST is
 protecting LIST(Nat) .
 sort OrdNatList .
 subsort Nat < OrdNatList
             < List(Nat) .
 vars N M : Nat .
 var  L : List(Nat) .
 mb nil : OrdNatList .
 cmb N M L : OrdNatList
   if N <= M /\ M L : OrdNatList .
endfm
```

We illustrate the reduction of terms (using equations and membership axioms) in Maude with the command reduce.

```
Maude> reduce in ORD-NAT-LIST :
         0 1 2 3 4 .
result OrdNatList : 0 1 2 3 4
```

Notice that, although no equation has been applied, the system has calculated the right sort using the membership 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; that is, 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.

Let us consider the following system module (mod...endm), in which a rule switches elements out of place.

```
mod SORTING is
 protecting ORD-NAT-LIST .
 vars N M : Nat .
 var  L : List(Nat) .
 crl N L M => M L N if N > M .
endm
```

The command rewrite reduces terms using rules, equations, and membership axioms.

```
Maude> rewrite in ORD-NAT-LIST :
         4 3 2 1 0 .
result OrdNatList : 0 1 2 3 4
```

The module SORTING is confluent and terminating. However, system modules need not be confluent nor terminating, and thus, some general ways to control the execution of rules may be required. These are called *rewriting strategies*, and we will come back to them in Section 4.5.

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

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

```
crl[r] :
 < O1 : C1 | atts1> ... <On : Cn | attsn >
 M1 ... Mm
 => < Oi1 : Ci1' | atts i1' > ... < Oik : Cik' | atts ik' >
 < Q1 : C1'' | atts1'' > ... < Qp : Cp'' | attsp'' >
 M1' ... Mq'
 if Cond .
```

where r is the rule label (rule labels are optional), $M_1 \ldots M_m$ and $M_1' \ldots M_q'$ are messages, $O_1 \ldots O_n$ and $Q_1 \ldots Q_p$ are object identifiers, $C_1 \ldots C_n$, $C_{i_1}' \ldots C_{i_k}'$, and $C_1'' \ldots C_p''$ are classes, $i_1 \ldots i_k$ is a subset of $1 \ldots n$, and *Cond* is a Boolean condition (the rule's guard). The result of applying such a rule is that: (a) messages $M_1 \ldots M_m$ disappear; that is, 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; that is, they are sent. Rule labels and guards are optional.

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

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

When several objects or messages appear in the left-hand side of a rule, they need to synchronize 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, which allows the suppression and modification of the attributes and methods of the base class (Ref. [22], Part 2-9.21), corresponds to Maude's module inheritance. Throughout the paper, by Maude inheritance, we shall mean Maude's notion of *class* inheritance, i.e., ODP's subtyping. Multiple inheritance is also supported in Maude [5].

4. Writing information specifications in Maude

In the ODP Reference Model, the information language uses a basic set of concepts and structuring rules, including those from ITU-T Recommendation X.902, ISO/IEC 10746-2, and three concepts specific to the information viewpoint: invariant schema, static

schema, and dynamic schema. The different schemata may apply to the whole system, or they may apply to particular domains within it.

Then, an ODP information specification consists of the specification of information objects, the behavior of those objects, and environment contracts for the system.

The concepts used in an information specification are explained below, together with their corresponding representation in Maude and a justification of the mapping rules used. To distinguish between the ODP and the Maude concepts, we will use names within their naming context whenever confusion is a danger (e.g., ODP class vs. Maude class).

4.1. Structural concepts

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

In 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 [22].

Information objects can be either atomic or represented as a composition of information objects. 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. When an information object is a composite object, the associated schemata are composed as well.

Information objects have types. A *type* of an information object is a predicate characterizing a collection of information objects. ODP objects may have several types. Some of these types may be dynamically acquired and lost [24].

Out of the multiple types of an object, the *template* type is essential because it expresses the requirements that the instantiations of a template are intended to fulfil; a template of an object specifies the common features of a collection of objects in sufficient detail that an object can be instantiated using it.

Information objects, as any other ODP objects, exhibit behavior, state, identity, and encapsulation. The behavior of an object is determined by the collection of *actions* in which the object can take part, together with a set of constraints on when they may occur. The set of actions associated with an object can be partitioned into *internal actions* and *interactions*. An internal action always takes place without the participation of the environment of the object. An interaction takes place with the participation of the environment of the object. A *communication* is defined in ODP as the conveyance of information between two or more objects as a result of one or more interactions, possibly involving some intermediate objects.

The traditional object model imposes a mindset of using a single hierarchy of subclasses of isolated objects exchanging messages and typically underestimates relationships between objects. It also requires properties of collections of objects (both collective state and collective behavior) to be expressed in terms of their refinement using attributes of individual objects. In contrast, a more general object model, such as the one followed by ODP, does not require invariants and operations to be "owned" by a single object; rather, it uses *collective state* for invariants, and *collective behavior* for operation and interaction specifications [25]. For instance, an ODP action is associated with at least one object and, thus, is not necessarily a message; an ODP contract is defined as an agreement governing part of the collective behavior of a set of objects; enabled behavior is defined in ODP as behavior characterizing a set of objects; and so on. Thus, a classical object model may not be the best option for specifying ODP constructs [24]. In this sense, Maude's object model is far more general and, therefore, more suitable for modeling ODP concepts, as we shall later see.

The following mapping rules will be used for representing such ODP concepts in Maude.

- *Information object types* will be represented by Maude classes. In Maude, each class is defined by a name and a set of attributes (of certain sorts) that describe the state of the objects of such a class.
- *Information objects* will then be represented by Maude objects. In Maude, each object belongs to a class, although it may change during the object's lifetime. In ODP, an object may have several types. Maude's multiple inheritance can be used in that case.

- In ODP, a *class* of information objects is the set of all information objects satisfying a given type. Notice, however, that the Maude concept of class is different. A Maude class is a "description" of a set of objects, while an ODP class is the set of objects itself.
- ODP *action types* will be represented by Maude rules. The left-hand side and guard of the rule describe the configuration of objects involved in the action and the conditions for the action to take place, while the right-hand side of the rule specifies the effects of such an action. This allows the representation of the collective behavior of a set of objects in a very natural way.
- Finally, Maude messages can be useful for representing the information exchanged in ODP *communications*, i.e., the information conveyed in object interactions.

4.2. Relationships between information objects

A relationship between information objects establishes a semantic connection between them. Some relationships, such as subtyping and composition, are defined in the RM-ODP foundational part, and invariant schemata are natural and appropriate for specifying them (see Section 5.4). Subtyping relationships between objects can be modeled in Maude by using Maude inheritance.

There is also the General Relationship Model (GRM [21]), the ISO standard that precisely defines and deals with relationships. Following the GRM, a relationship between information objects is a collection of information objects together with an invariant referring to the properties of those objects. The relationship invariant defines the "collective state" of the relationship participants. Although not part of RM-ODP, some system designers could may well like to use the GRM concepts for modeling relationships. In this case, one natural way for representing such kind of relationships in the ODP information viewpoint is in terms of invariant schemata (see Sections 4.5 and 5.4).

Finally, according to the ODP information viewpoint, a relationship among information objects can be modeled as part of the state of such objects (Ref. [22], Part 3-6.2). To represent such relationships in Maude, we can use a Maude class with the name of the relationship as its name and whose attributes are the identifiers of the participants and the relationship's attributes.

4.3. Static schemata

A static schema is 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" [22]. Examples of static schemata include the specification of the initial state of the system (i.e., the initial state of its objects and their initial relationships) or any other specific state of the system of particular relevance to the specifier or to any of the system stakeholders.

Such specification of the state of the objects in the system and their relationships is provided by the Maude configurations representing the information objects at each moment in time. Therefore, static schemata will be represented by Maude object configurations.

Static schemata should always fulfill the system constraints imposed by the invariant schemata. This is guaranteed because the execution of the system is driven by Maude rewriting strategies (see Section 4.5), which will check that the initial state satisfies the invariants and will not allow invalid system states to be reached.

4.4. Dynamic schemata

A dynamic schema describes the allowable state changes of the system. Dynamic schemata will be represented by making use of the fact that behavior in an information system can be modeled as transitions from one static schema to another (Ref. [22], Part 3-6.1.3). Thus, dynamic schemata will be represented by Maude rules, whose left- and right-hand sides represent, respectively, the configuration of objects and messages before and after the state change of the system. They describe the allowed state changes of one or more information objects.

4.5. Invariant schemata

In the information viewpoint, an invariant schema establishes constraints that must always hold for the information objects.

Basically, an invariant is a predicate that a specification or program always requires to be true.

However, we can distinguish two kinds of invariants, depending on whether they can be proved (or inferred) from the specifications or they form part of the specifications themselves (hence restricting the behavior of the system). We shall refer to the first kind of invariants as *deducible* or *external* invariants, and to the second kind as *constraining* invariants.

External invariants are usually used to verify specifications and programs against them. This verification process may be achieved in different ways—and using different techniques and tools—depending on the logic in which the invariant predicates are expressed and the logic supporting the specification notation used. Notice that these two logics may not coincide; in fact, they do not coincide in most cases—we may be interested, for example, in verifying whether a certain program written in C satisfies a given property expressed in some temporal logic.

In general, not all specifications need to be executable (e.g., declarative specification such as business models), but in the case of executable specifications of systems, we can also use a dynamic approach to verify a given specification against an external invariant, by checking that the execution traces of the specification satisfy such an invariant. Then, we talk about *model checking*, if we study and check all the possible system execution traces, or about *monitoring*, if we just consider the actual execution trace of the system, checking that the invariant always holds for such a trace.

On the other hand, constraining invariants cannot be deduced from the specification of the system; on the contrary, they are part of the specification themselves, constraining the behavior of the system. Therefore, dealing with such invariants implies a completely different approach.

In the case of executable specifications, the fact that constraining invariants restrain the possible configurations and behavior of the system forces to take them into account when executing or simulating the system being specified. In this case, we do not want to stop the execution of the program when we reach a state in which the invariant is not satisfied, as we can do when monitoring. On the contrary, what we want is to drive the system execution using such invariants, hence preventing the system from getting into any invalid state.

The execution or simulation of specifications with constraining invariants is typically based on integrating somehow the invariants into the system code. However, such an integration is sometimes unsatisfactory, specially if the invariants get lost amidst the code and become difficult to locate, trace, and maintain. If the specification language is not directly executable, as happens, e.g., with Object-Z and UML, the specifications and the invariants or constraints on them must be translated to (and combined in) a different language, which make things worst. Moreover, as we have mentioned above, the programs and the invariants to be satisfied are usually expressed in different formalisms and live at different levels of abstraction. Having support for expressing invariants over our specifications, being able to directly execute them, and having the possibility of using different formalisms for expressing invariants are, therefore, important requirements.

Finally, we see the convenience of expressing some of the constraining invariants in a modular way. In this way, the system specifier could define those invariants without having to hard-wire them into the actions specification, hence improving the readability, modularity, traceability, and maintainability of the system specification. In any case, the decision on whether to specify an invariant as part of the specification of an action, or independently from it, is a design decision (usually very difficult) that falls beyond the scope of this paper. Our main concern in this article is to explain the Maude mechanisms that will allow system designers to specify the invariants in either way; now, it is up to them to make the decision in each case and for each invariant.

More precisely, in this section, we will discuss the two main issues involved in the specification of constraining invariants in Maude: (a) how to express them and (b) how to take them into account when executing the Maude specifications.

Let us first say that Maude does not provide direct support for expressing invariants, as, e.g., Z does. However, Maude has very good properties as a logical and semantic framework in which to express many different languages, logics, and models of computation [28]. Thus, it turns out to be a very good candidate for giving support to different types of invariants, which may be expressed in different formalisms, as shown in Section 4.5.2.

Second, Maude specifications can be executed, and, thanks to its reflective capabilities, the Maude system provides very powerful and flexible ways of controlling the executions. Our proposal is then simple: Let us use rewriting strategies that take into account the constraining invariants representing the invariant schemata of the information viewpoint specifications. Section 4.5.1 proposes a general method for executing specifications, which is parameterized by the specification that we want to execute, the logic in which the invariants are expressed, and the invariants themselves. Although it could have been done in a monolithic way, this offers, among others advantages, the possibility of decoupling the specification of the systems and the invariants on them, which may, in addition, be given in different formalisms.

Section 4.5.1 shows how to define execution strategies for dealing with invariants, and Section 4.5.2 shows how to express such invariants in Maude.

4.5.1. Defining execution strategies that deal with invariants

Maude provides two built-in strategies for executing rewrite theories, namely, a top–down lazy rule-fair strategy and a position-fair bottom–up strategy [6]. However, since we may need different ways of execution in some cases, it also provides facilities for defining new rewriting strategies [5], thus guiding the rewrites in the desired direction, depending on our specific needs.

Maude provides key metalevel functionality for metaprogramming and for writing execution strategies, so that strategies can be defined using statements in a normal module. In general, strategies are defined in extensions of the predefined module `META-LEVEL` by using predefined functions in it, like `metaReduce`, `metaApply`, `metaXapply`, etc., as building blocks. `META-LEVEL` also provides sorts `Term` and `Module`, so that the representations of a term T and of a module M are, respectively, a term $\overline{T}$ of sort `Term` and a term $\overline{M}$ of sort `Module`.

Of particular interest for our current purposes are the partial functions `metaReduce` and `metaXapply`.[1]

[1] We have simplified the form of these functions for presentation purposes, since we do not need here their complete functionality. See Ref. [5] for the actual descriptions.

```
op metaReduce :
     Module Term ~> Term .
op metaXapply :
     Module Term Qid ~> Term .
```

metaReduce takes a module $\overline{M}$ and a term $\overline{T}$ and returns the metarepresentation of the normal form of T in M, i.e., the result of reducing T as much as possible using the equations in M. Partial function `metaXapply` takes, as arguments, a module $\overline{M}$, a term $\overline{T}$, and a rule label L and returns the metarepresentation of the term resulting from applying the rule with label L in M on the term T. Note that these operators are declared using ~>, meaning that they will return an error term in the kind `[Term]`, associated to the sort `Term`, if the term is not reduced. One can think of a kind as an error "supersort", where, in addition to correct well-formed terms, there are undefined or error terms.

Let us suppose first that we are interested in a strategy that rewrites a given term by applying on it the rules in a given module, in any order. The strategy should just try to apply the rules one by one on the current term until it gets rewritten. Once a rule can be applied on it, the term resulting from such an application becomes the current term, and we start again. If none of the rules can be applied on a term, then it is returned as the result of the rewriting process. Such a strategy can be specified as follows:

```
op rew : Module Term -> Term .
op rew : Module Term QidList
     QidList -> Term .
eq rew(M, T)
  = rew(M, T, labels(M) , nil) .
eq rew(M, T, L LL, LL')
  = if metaXapply(M, T, L) :: Term
    then rew(M, metaXapply(M, T, L),
             LL' LL L, nil)
    else rew(M, T, LL, LL' L)
    fi .
eq rew(M, T, nil, LL) = T .
```

In these equations, we assume a function `labels` that takes a module and returns a list with the labels (quoted identifiers, of sort `Qid`) of the rules defined in such a module. We also assume a sort `QidList` of lists of quoted identifiers, with `nil` as the empty list and _ _ (empty syntax) as the concatenation operator.

Note also the use of the `metaXapply` function. A rewriting step $T \xrightarrow{L} T'$ is accomplished only if the rule labeled L is applicable on the term T, being T' the term returned by `metaXapply`(M, T, L). The membership assertion "`metaXapply`(M, T, L)`::Term`" is used to check whether the result of the application of the rule is of sort `Term` or not. In case the rule cannot be applied, operation `metaXapply` returns an error term in the kind `[Term]`. Then, given (the metarepresentation of) a module M and (the metarepresentation of) a term T, the `rew` operation tries to rewrite T by applying the rules in M one by one, until one is found that can be successfully applied to T. If no rule can be applied, the original term is returned (as specified by last equation). In case a rule L of M is found that can be successfully applied to T (resulting in term T'), the `rew` operation is recursively invoked on T'.

Now, if we want to deal with constraining invariants, we just need to change the previous strategy to check an additional invariant condition before taking a rewriting step. This check will guarantee that every new state satisfies the specification invariant.

To implement this new strategy, we shall consider a satisfaction Boolean predicate `_|=_`. Given a configuration of objects C (that represents a state of the system) and an invariant I, the expression $C|=I$ will evaluate to `true` or `false`, depending on whether the configuration C satisfies invariant I. The only difference with respect to the previous strategy is that, now, it must check whether the initial state satisfies the invariant and then take a rewriting step only if the term can be rewritten using a particular rule, and it yields to a next state that satisfies the invariant. Invariant I is checked by evaluating the expression $T'|=I$ for a given candidate transition $T \xrightarrow{L} T'$. Note, however, that the rewriting process takes place at the metalevel, and therefore, it is convenient to have the invariant also metarepresented. Thus, we use `metaReduce` for evaluating the satisfaction of the property. The new rewriting strategy, which we have called `rewInv`, is shown in Fig. 1. Note that `rewInv` takes four arguments: the module specifying the system, the module defining the satisfaction relation for the logic in which the invariants are expressed, the initial term to be rewritten, and the invariant to be satisfied. Now, the auxiliary function `rewInvAux` is invoked if the initial state satisfies the invariant; otherwise, the initial term is left as an error term.

In this way, the rules describing the system can be written independently from the invariants applied to them, and the module specifying the system is independent of the logic in which the invariants are expressed, thus providing the right kind of independence and modularity between the system actions and the system invariants. In fact, the strategy is parameterized by the module to be executed (M), the invariant to be preserved (I), and the module defining the satisfaction relation (M'). This allows the use of different logics (e.g., propositional logic, linear temporal logic, etc.) to express the invariant without affecting the strategy or the system to execute. We

```
op rewInv : Module Module Term Term ~> Term .
op rewInvAux : Module Module Term Term QidList QidList -> Term .
ceq rewInv(M, M', T, I)
   = rewInvAux(M, M', T, I, labels(M), nil)
   if metaReduce(M', '_|=_[T, I]) = 'true.Bool .
eq rewInvAux(M, M', T, I, L LL, LL')
   = if metaXapply(M, T, L) :: Term
         and-then metaReduce(M', '_|=_[metaXapply(M, T, L), I])
      then rewInvAux(M, M', metaXapply(M, T, L), I, LL' LL L, nil)
      else rewInvAux(M, M', T, I, LL, LL' L)
      fi .
eq rewInvAux(M, M', T, I, nil, LL) = T .
```

Fig. 1. Rewriting strategy with invariants.

shall illustrate this in the case of propositional logic in the next section. Note that the mechanism is very expressive and powerful: A simple extension of the strategy would allow us, for instance, to simultaneously consider invariants expressed in different logics, use heuristics to guide the process, etc.

4.5.2. Expressing invariants

Invariant predicates can be expressed in different logics. We have already experimented with invariants expressed in propositional logic [11] and linear temporal logic [12]. In this section, we shall illustrate the definition of the satisfaction relation in the case of propositional logic. A detailed discussion on how LTL invariants may drive the system execution, together with their complete Maude implementation, several examples, and additional extensions, such as the use of backtracking, can be found in Ref. [12]. The temporal logic we considered there is the same that Maude uses in its model checker [16] and the

```
fmod PROPOSITIONAL-CALCULUS is
   sort Formula Proposition .          ---- propositional formulae
   subsort Proposition < Formula .
   ops true false : -> Formula .
   op _and_ : Formula Formula -> Formula [assoc comm prec 55] .
   op _or_ : Formula Formula -> Formula [assoc comm prec 59] .
   op _xor_ : Formula Formula -> Formula [assoc comm prec 57] .
   op not_ : Formula -> Formula [prec 53] .
   op _implies_ : Formula Formula -> Formula [prec 61] .
   op _iff_ : Formula Formula -> Formula [assoc prec 63] .
   vars A B C : Formula .
   eq true and A = A .
   eq false and A = false .
   eq A and A = A .
   eq false xor A = A .
   eq A xor A = false .
   eq A and (B xor C) = A and B xor A and C .
   eq not A = A xor true .
   eq A or B = A and B xor A xor B .
   eq A implies B = not(A xor A and B) .
   eq A iff B = A xor B xor true .

   sort State .
   op _|=_ : State Formula -> Bool .  ---- satisfaction relation
   var S : State .
   eq S |= (A and B) = (S |= A) and (S |= B) .
   eq S |= (A xor B) = (S |= A) xor (S |= B) .
   eq S |= true = true .
   eq S |= false = false .
endfm
```

Fig. 2. Maude specification of propositional formulae.

approach used to deal with it is similar to the one proposed by Havelund and Roçu in [18] for monitoring Java programs. Note, however, that although the strategy `rewInv` given in Section 4.5.1 is valid for logics like propositional logic, it does not work in the case of temporal logics. For example, the satisfaction of a linear time logic (LTL) formula cannot be decided considering particular states, but we need to look at future states, and, perhaps, even to complete traces.

Given a set of atomic propositions of sort `Proposition`, we define the formulae of the propositional calculus and a satisfaction relation for it in the `PROPOSITIONAL-CALCULUS` module shown in Fig. 2.

The first part of the module introduces the sort `Formula` of well-formed propositional formulae, with two designated formulae, namely, `true` and `false`, with the obvious meaning. The sort `Proposition`, corresponding to the set of atomic propositions, is declared as a subsort of `Formula`. Then, the usual conjunction, disjunction, exclusive or, negation, implication, and if and only if operators are declared. These declarations follow quite closely the definition of Boolean values in Maude and OBJ3 [17], which are based on the Church-Rosser and terminating decision procedure proposed by Hsiang [19]. This procedure reduces valid propositional formulae to the constant `true`, and all the others to some canonical form modulo associativity and commutativity, which consists of an exclusive disjunction of conjunctions.

The satisfaction relation is defined in the second part of the module. As mentioned above, the satisfaction relation `_|=_` is a Boolean predicate that, given a state and a formula, evaluates to `true` or `false`, depending on whether the given state satisfies such a formula. Notice that the operator `_|=_` takes a propositional formula as second argument and returns a Boolean value, being `Boolean` a predefined sort in Maude, also with constants `true` and `false`, and with operations `_and_`, `_or_`, `_xor_`, etc. We shall see in Section 5.4 how, once the atomic propositions of interest for a particular problem are defined, and the satisfaction relation is specified for them, we may use this logic to express invariants of systems and to evaluate their satisfaction for the states of such systems.

5. A case study

The following example illustrates the use of Maude for representing the ODP viewpoint specifications of a system. The example is about a computerized system to support the operations of a university library, in particular, those related to the borrowing process of the library items. The system should keep track of the items of the university library, its borrowers, and their outstanding loans. Instead of a general and abstract system, this example is based on the regulations that rule the borrowing process defined at the Templeman Library at the University of Kent at Canterbury, a library that has been previously used by different authors for illustrating some of the ODP concepts (see, e.g., Refs. [3,36]).

The basic rules that govern the borrowing process of that library system are as follows:

(1) Borrowing rights are given to all academic staff and to postgraduate and undergraduate students of the university.
(2) Library books and periodicals can be borrowed.
(3) Each library item has a unique identifier. The library maintains a record for each item, which, in addition to its identifier, contains the relevant information about the item (e.g., title, author, ISBN or ISSN, etc.). This information also includes the item location when it is in the library, and the item status: on-loan, free, or disposed (if the item has been destroyed, lost, or thrown away).
(4) There are prescribed periods of loan and limits on the number of items allowed on loan to a borrower at any time. These limits are:
   – Undergraduates may borrow, at most, eight books. They may not borrow periodicals. They may borrow books for, at most, four weeks.
   – Postgraduates may borrow at most 16 books or periodicals. They may borrow periodicals for up to one week, and books for up to four weeks.
   – Teaching staff may borrow at most 24 books or periodicals. They may borrow periodicals for up to one week, and books for up to one year.

(5) Items borrowed must be returned by the due day and time, which is specified when the item is borrowed.
(6) 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.
(7) Borrowers returning items must hand them in to a librarian at the Main Loan Desk. Any charges due on overdue items must be paid at this time.
(8) Failure to pay charges may result in suspension by the librarian of borrowing facilities.

In the following, we shall refer to these rules as the "textual regulations" of the library system. This section describes a specification of the ODP information viewpoint of such a system, using Maude as formal notation. Such information specification describes the types of information and the relationships between them that are required to define the system. It uses the information language in RM-ODP and, where appropriate, interprets the language in terms of the Maude notation.

The information specification in this section defines both the basic concepts for information used in this specification and the invariant, static and dynamic schemata for it.

5.1. Basic concepts

From the textual regulations of the library, we can identify several main information object types, namely, *borrowers*, library *items*, and *librarians*. These objects represent the information kept in the system about the entities they model. In addition, a *calendar* object should be in charge of representing the passage of time, and *loan* objects will represent the relationships between borrowers and items (see Section 4.2).

These information object types are represented in Maude by the following Maude classes:

```
class Borrower |
  id : Qid,              name : Qid,         address : Qid,
  faculty : Qid,         maxLoans : Int,     loans : Set(Oid),
  borrowedItems : Int,   finesDue : Money,   suspended : Bool,
  bookLoanPeriod : Int, periodicalLoanPeriod : Int.

class Item |
  id : Qid,              title : String,     publisher : Qid,
  location : Qid,        publicationDate : Date,
  status : ItemStatus,   loan : Default(Oid).

class Calendar | date : Date.

class Librarian.

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

The predefined sorts `Oid` and `Qid` are used to represent object identifiers and quoted identifiers (i.e., general identifiers), respectively. We assume parameterized sorts `Set(X)` and `Default(X)`, which define, respectively, sets (of any type used in the instantiation) and sorts with default values. Besides, user-defined can be easily specified in Maude. For instance, enumeration sorts such as `ItemStatus` can be defined as follows:

```
sort ItemStatus .
ops onLoan free disposed other :
      -> ItemStatus .
```

In our information viewpoint specification, another information object will store the general details about the library (`Library`), such as the daily rates to be charged to late-returners, together with the lists of its current members: borrowers, items, librarians, calendar, and the set of outstanding loans.

```
class Library |
  dailyCharges : Money,    borrowers : Set(Oid),
  items : Set(Oid),        librarians : Set(Oid),
  calendar : Oid,          loans : Set(Oid).
```

The textual regulations of the Templeman library also distinguish three special kinds of borrowers (academic staff, undergrads, and postgrads) and two kinds of items (books and periodicals). Their subtyping relationships have been modeled in Maude using class inheritance.

```
class Academic .
class Undergrad .
class Postgrad .
subclasses Academic Undergrad
  Postgrad < Borrower .
class Book | author : Qid,
  edition : Int, ISBN : Qid,
  acquisitionDate : Date .
class Periodical | volume : Int,
  number : Int, ISSN : Qid .
subclasses Book Periodical < Item .
```

Once we have specified the basic information object types, we need to specify the possible interactions among these objects. In the information viewpoint, such interactions will be specified by a set of dynamic schemata, by describing their preconditions and the state changes that they cause in the system.

5.2. Static schemata

Static schemata provide instantaneous views of information, e.g., at system initialization, or in any other specific moment in time that is relevant to any of the system stakeholders. This specification of the instantaneous state of the objects is precisely the one provided by the Maude *configurations* at each moment in time.

One of the benefits of Maude configurations is that they allow to capture not only the individual state of each object, but also the collective state of the system. For example, Maude configurations may also contain messages, which, in this example, will represent pending interactions, i.e., requests that have been sent by the source object but have not yet been consumed by the target object(s).

For instance, the Maude configuration shown in Fig. 3 represents a static schema that models the state of the system at a moment in time (date=1000), in which there are only two borrowers (John and Mary), two librarians (Eve and Pete), two books (Ulysses and Dubliners), and one periodical (yesterday's edition of The Times). There is only one loan (Mary borrowed Ulysses in day 800), and there is a pending request of John to borrow Dubliners. Note the use of dots where nonrelevant information has been omitted for space reasons.

5.3. Dynamic schemata: description of the system behavior

The dynamic schemata describe the allowed state changes of the system or of any subset of its constituent information objects. In our proposal, dynamic schemata are represented by Maude rules, whose left-hand side and guard specify the preconditions of the dynamic schema, and the right-hand side represents the state change caused in the system.

Most approaches for representing the ODP information viewpoint dynamic schemata are based on describing the effect of object interactions only. However, dynamic schemata can be more general than that: State changes can also be caused by other kinds of actions, such as internal actions. For instance, the `Calendar` information object was in charge of

```
< 'Templeman : Library | loans : '223344,
     calendar : 'TheClock,    dailyCharges : 0.02, *** 2 pence
     librarians : 'Eve 'Pete, borrowers : 'John 'Mary,
     items : 'Ulysses 'Dubliners '20031103TheTimes >
< 'TheClock : Calendar | date : 1000 >
< 'John : Undergrad | id : '1,
     maxLoans : 8, borrowedItems : 0, bookLoanPeriod : 28,
     finesDue : 0, suspended : false, periodicalLoanPeriod : 0,
     name : "John Smith", address : "...", ... >
< 'Mary : Academic | id : '2,
     maxLoans : 24, borrowedItems : 1, bookLoanPeriod : 365,
     finesDue : 0,  suspended : false, periodicalLoanPeriod : 7,
     name : "Mary Gordon", address : "...", ... >
< 'Ulysses    : Book | id : 'b1, status : onLoan,
     author : "J. Joyce", location : "Shelf 1", ... >
< 'Dubliners : Book | id: 'b2, status : free,
     author : "J. Joyce", location : "Shelf 1", ... >
< '20031103TheTimes : Periodical | id : 'p1, status : free,
     title : "The Times", location : "Shelf 2",
     publicationDate : 999, ... >
< 'Eve : Librarian >
< 'Pete : Librarian >
< '223344 : Loan | borrower : 'Mary, item : 'Ulysses,
     issueDate : 800, dueDate : 1165 >
borrowRequest('John, 'Dubliners, 'Eve)
```

Fig. 3. An example of a static schema capturing a system state.

providing the current date. Given the use that we are going to make of it, we can assume, for example, that the date gets increased by the following rewrite rule.

```
rl[tic] :
  < O : Calendar | date : D >
  => < O : Calendar |
         date : D + 1 > .
```

Note that 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.

To illustrate the use of Maude rules to represent the dynamic schemata corresponding to object interactions, we show how to specify a user request to a librarian to borrow an item. We model such an interaction with three rules, covering all the possible cases. The rest of the dynamic schemata follow very similar patterns.

Please notice that most restrictions may be considered both in the rules and as constraining invariants. For example, the fact that an undergrad cannot borrow a periodical may be specified as a restriction on the borrowing interaction, but also as a constraining invariant, stating that a periodical can never be borrowed by an undergrad. In general, our preferred practice is to model invariant schemata as rule conditions when they are applied only to specific actions, modeling them as constraining invariants when either they affect several rules or they are expected to be reused or changed. In our example, we consider whether a borrower is suspended, whether he/she is an undergrad, and whether an item is on loan as constraints on the borrowing interaction, since we

find them more "interaction specific". We shall come back to this issue in Section 5.4.

Fig. 4 shows the Maude rule that represents the dynamic schema corresponding to the successful borrowing request for a periodical item. This rule is fired under the presence of a `borrowRequest` message and only if the borrower is not suspended, the item is free, and the user can borrow periodicals (i.e., is not an undergrad). The consequence of such a rule is that the request is accepted, a `Loan` object is created with the corresponding information, and an `acceptLoan` message is produced, which represents the triggering of the corresponding interaction that will happen between the librarian and the borrower when the loan is accepted. The function `class` takes an object as argument and returns its actual class. Thus, if the `Borrower` object to which the above rule applies is, for instance, `< 'John : Postgrad |...>`, then the class function applied to it returns `Postgrad`, and not `Borrower`. Note also 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 a rule will maintain their previous values unmodified.

Likewise, the rule `borrow-request-book-ok` shown in Fig. 5 specifies the dynamic schema that handles the (successful) borrowing of books. Note that, in this case, there is no restriction on the kind of borrower.

Any other case is handled by the rule shown in Fig. 6, which establishes the denial conditions for a borrow request.

5.4. Invariant schemata

Many different invariant schemata can be defined for this specification. In this section, we shall show several of them, which have been chosen for being illustrative examples of how different invariants can be represented in Maude.

According to the discussion in Section 4.5, several kinds of invariant schemata can be identified, depending on how they are expressed. For instance, some invariant schemata that represent the invariants (in GRM terms) defining the associations between information objects can be expressed as part of the state of those objects. In Maude, such a state is stored in the attributes of the Maude objects representing the ODP information objects. Thus, the fact that a loan in

```
crl [borrow-request-periodical-ok] :
  borrowRequest(B, I, A)
  < B :Borrower | suspended : false, borrowedItems : N,
                  loans : BLS, periodicalLoanPeriod : LP >
  < I : Periodical | status : free >
  < L : Library | items : I IS, librarians : A AS,
                  borrowers : B BS, loans : LS, calendar : C >
  < C : Calendar | date : Today >
  < A : Librarian | >
  => < B : Borrower | borrowedItems : N + 1, loans : O BLS >
     < I : Periodical | status : onLoan, loan : O >
     < L : Library | loans : O LS >
     < C : Calendar | >
     < A : Librarian | >
     < O : Loan | borrower : B, item : I,
                  issueDate : Today, dueDate : Today + LP >
     acceptLoan(A, I, B)
  if (class(< B : Borrower | >) =/= Undergrad) .
```

Fig. 4. Maude specification of the dynamic schema corresponding to the successful borrowing request for a periodical item.

```
rl [borrow-request-book-ok] :
  borrowRequest(B, I, A)
  < B : Borrower | suspended : false, borrowedItems : N,
                   loans : BLS, bookLoanPeriod : LP >
  < I : Book | status : free >
  < L : Library | items : I IS, librarians : A AS,
                  borrowers : B BS, loans : LS, calendar : C >
  < C : Calendar | date : Today >
  < A : Librarian | >
  => < B : Borrower | borrowedItems : N + 1, loans : O BLS >
     < I : Book | status : onLoan, loan : O >
     < L : Library | loans : O LS >
     < C : Calendar | >
     < A : Librarian | >
     < O : Loan | borrower : B, item : I,
                  issueDate : Today, dueDate : Today + LP >
     acceptLoan(A, I, B) .
```

Fig. 5. Maude specification of the dynamic schema corresponding to the successful borrowing request for a book.

the library system takes place between exactly one borrower and one item has been represented by the attributes `borrower` and `item` of the `Loan` class, each of which can hold only one object identifier: precisely the identifiers of the `Borrower` and `Item` objects involved in the loan.

Other invariants have been directly incorporated into the Maude rules, such as the one that does not allow an undergrad to borrow periodicals, or the one that requests that an item must be free to be borrowed (see Section 5.3).

Finally, constraining invariants can also be expressed as formulae in a certain logic, and then enforced on the system execution. For illustration purposes, we shall consider the following examples of this kind of invariants:

(1) The information on loans must be consistent. That is, the borrower and the item referenced in a loan object must have such a loan among their loans (the only one in the case of the item).
(2) Borrowers cannot exceed the number of pending loans allowed by the library regulations.
(3) A nonacademic cannot borrow any item if he/she has pending fines.

```
crl [borrow-request-deny] :
  borrowRequest(B, I, A)
  < B : Borrower | suspended : SP > < L : Library | >
  < I : Item | status : ST >        < A : Librarian | >
  => < B : Borrower | > < L : Library | >
     < I : Item | >     < A : Librarian | >
     denyLoan(A, I, B)
  if SP or ST =/= free
     or ((class(< B : Borrower | >) == Undergrad)
         and (class(< I : Item | >) == Periodical)) .
```

Fig. 6. Maude specification of the dynamic schema corresponding to the denial of a borrow request.

Notice that invariant (1) may be deduced from the specification of the system. However, it can also be explicitly stated as a constraining invariant, as we will show here. Invariant (2) could have been enforced as part of the conditions of the rules, as we did for the status of the item to be borrowed, or the category of the borrower in the rules for borrowing in Section 5.3. However, there are some situations in which it is better to specify invariants separately from the Maude rules that specify the interactions they constrain. Decoupling the invariants from the actions that they constrain is currently seen in many cases as a good design practice, since it may help improving the modularity and evolvalibity of the specifications produced [35]. As mentioned before, we try to model invariant schemata as rule conditions when they are applied only to specific actions, modeling them as constraining invariants when either they affect several rules or they are expected to be reused or changed. Finally, invariant (3) did not originally appear in the textual description of the system. With it, we try to illustrate how new invariants not considered initially may be easily added.

To define invariant schema (1), we can specify an atomic proposition `consistentLoans`. As explained in Section 4.5, we must define the satisfaction relation for each atomic proposition. In this case, we define the satisfaction of proposition `consistentLoans` with two equations: one detecting the transgressing case and an otherwise case (note the use of the Maude `owise` attribute of the second equation).

```
op consistentLoans :
     -> Proposition .
ceq < I : Item | loan : L' >
  < L : Loan | borrower : B,
               item : I >
  < B : Borrower | loans : LS > C
    |= consistentLoans
  = false
  if not L in LS or L =/= L' .
eq C |= consistentLoans = true
  [owise] .
```

Invariant schema (2) may be modeled using an atomic proposition `tooManyLoans` on the number of borrowed items as follows:

```
op tooManyLoans : -> Proposition .
eq < O : Borrower |
       borrowedItems : N,
       maxLoans : M > C
    |= tooManyLoans
  = N > M or (C |= tooManyLoans) .
eq C |= tooManyLoans = false
  [owise] .
```

In this case, the satisfaction of the atomic proposition has been defined recursively, checking that each of the borrowers in the configuration has not reached the maximum number of loans.

Invariants can also be specified as formulae over atomic propositions using the connectives defined in Section 4.5. For example, the invariant "`not tooManyLoans`" avoids the application of a borrowing rule if it leads to a configuration where a borrower has more pending loans than allowed, according to the library regulations, thus making unnecessary the use of any guard to check this restriction in the Maude rules that define the borrowing operations.

Invariant schema (3) states that nonacademics cannot borrow an item if they have pending fines. Although not originally included in the textual regulations, we could easily add this restriction by considering an invariant "`not nonAcademicWithPendingFines`", where the atomic proposition can be defined as follows:

```
op nonAcademicWithPendingFines :
    -> Proposition .
ceq < B : Borrower |
        finesDue : N > C
    |= nonAcademicWithPendingFines
  = true
  if class(< B : Borrower | >)
       =/= Academic
    /\ N > 0 .
eq C |= nonAcademicWithPendingFines
  = false[owise] .
```

5.5. The execution of the system

Now, using the module `PROPOSITIONAL-CALCULUS` given in Section 4.5, a module `LIBRARY` containing the specification of the library

```
red rewInv(LIBRARY,
           PROPOSITIONAL-CALCULUS,
           initial-state,
           consistentLoans
             and not tooManyLoans
             and not nonAcademicWithPendingFines) .
```

Fig. 7. Executing the Maude specifications of the Library system.

system, and a module `TEST`, with the declaration of a constant `initial-state`, with value an initial configuration, and the definitions of the above atomic propositions, we can execute the system specification starting in such an initial state constrained by the above invariants, as shown in Fig. 7. (For clarity, overlined $\overline{\mathcal{R}}$; and $\bar{t}$ stand for terms representing, respectively, the module $\mathcal{R}$ and the term t.) The strategy `rewInv` introduced in Section 4.5.1 will drive the execution of the system taking the invariants into account, avoiding those states in which the invariants do not hold.

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, probably the most widely accepted notations for formalizing the information viewpoint are Z and Object-Z. Initially, Z was chosen because the schemata defined in the information viewpoint could be directly mirrored into the Z schemata. Furthermore, Johnson and Kilov [25] have shown how to formalize some of the basic concepts of ODP in Z, with special emphasis on the collective state and behavior of objects, and discussed how to deal with relationships.

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 limitations since it is object oriented, allows modularity, and incorporates a subset of temporal logic for expressing class invariants. Therefore, it seems to be a better candidate language for formalizing the information viewpoint. However, the use of Object-Z for specifying the information viewpoint also presents some shortcomings:

- First, interactions are usually modeled as object operations and assigned to just one object (included as methods in the object's definition class). How to deal with interactions in which there is more than a principal object (e.g., in the case of synchronous interactions)? In our approach, interactions are represented by rules, and therefore, such limitation does not exist.
- Dynamic schemata are only represented in Object-Z in terms of the behavior of the operation that represents the action that causes the state change. As mentioned before, this does not cover the cases of internal actions, spontaneous state changes (due, for instance, to the presence of a given object in the system and not necessarily caused by any interaction), nor those cases of collective behavior, in which the action causing the state change does not belong to a particular object.
- Temporal logic is used to express the invariants, but the fragment of temporal logic included in Object-Z is too limited, as pointed out in Ref. [36]. In our proposal, we have shown how different logics (e.g., propositional logic [11] and linear temporal logic [12]) can be effectively used to express invariants.
- Another disadvantage of the use of Object-Z appears when representing some object types. Object types are represented as Object-Z classes, which is the natural way of doing it. However, Object-Z does not offer any mechanism for the dynamic reclassification of objects, which may be the case under some particular circumstances (e.g., it may be required for representing systems in dynamically configurable networks). Again, this is not an issue in Maude, since the class of an object can be changed during its lifetime.

- As mentioned in Section 4.1, the traditional object model assumes a single hierarchy of subclasses of isolated objects exchanging messages and typically underestimates relationships between objects. By contrast, a more general object model, such as the one followed by ODP and Maude, does not require invariants and operations to be owned by a single object; rather, it uses *collective state* for invariants and *collective behavior* for operation and interaction specifications [25]. Thus, a classical object model may not be the best option for specifying ODP constructs [24]. In this sense, Maude's object model is more general and, therefore, more suitable for modeling ODP concepts.
- 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 [5].
- Finally, other Object-Z issue is related to its expressiveness for representing concepts from other formal languages, to study ODP viewpoint consistency. 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. [2] the authors propose the translation of LOTOS into Object-Z. However, many important aspects of the specification are usually lost in these translations, since the underlying logic of Object-Z is not expressive enough. We think that Maude can greatly help in this point and is something that we want to explore further. Rewriting logic is such that faithful translations from other formal description techniques into Maude can be obtained [31].

Apart from such "formal" approaches, some authors have also considered graphical notations for representing the ODP information concepts. For example, Dustzadeh and Najm [15] showed in 1997 how OMT and Fusion may support ODP information modeling, providing some semantics for the object diagrams of these graphical application development methodologies. Many other authors also consider UML as a well-suited language for ODP information modeling because of its appealing graphical syntax and because it is part of fully integrated development methodologies, such as RUP [23] or Catalysis [9]. Moreover, UML is widely known and easily accepted by all kinds of users. However, the loose semantics of UML is a major drawback if precise and unambiguous specifications are required. This issue is currently being addressed by different research groups and initiatives.

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

There are several research areas that we plan to address in the short term. The first area is related to two important issues in ODP, 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 to check it if such viewpoints are specified in different formalisms. Thus, we have two options: either we write all viewpoints specifications in the same formal notation or we use different formalisms for the different viewpoints and then translate them into a common model. However, there is a general belief that no formal method applies well to all problem domains, which invalidates the first option. 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, Maude for the information viewpoint, and LOTOS or SDL for the computational viewpoint.

It has been shown that rewriting logic has very good properties as a logical framework, in which many different languages and logics can be represented, and as a semantic framework in which semantics can be given to them [31]. Rewriting logic and Maude have also been proposed for specifying the enterprise [13] and the computational viewpoints [33,34], and we plan to study their adequacy for being used in the specification of the others. Formalisms such as CCS, Object-Z, 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.

Finally, tool support is an essential issue for any engineering approach to system specifications. Tool support should cover all the system specification life cycle, providing support for writing and validating them, for reasoning about their properties, and even for executing them, if possible. 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 information 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 information specifications.

Having said that, we also feel that some graphical tool support may be required for the wide adoption of our proposal. We have already mentioned some of the advantages of using graphical notations. In this sense, we are currently working on the smooth integration of our approach with the current proposals for modeling the ODP information viewpoint using UML, as we have done for the Enterprise Viewpoint [10]. This would allow the stakeholders of the system to use a more user-friendly graphical notation like UML to describe the system information viewpoint, and then translate them into the corresponding Maude specifications. Moreover, we are working on the access, from the UML environment to the Maude toolkit, for reasoning about the specification produced. This will allow us to model check the UML specifications or to prove some of their properties using the Maude theorem prover, without forcing the user to have a strong background and knowledge of Maude or of any other formal notation or method. In this way, we could "hide" the complexity involved in writing the formal specifications of the system and reasoning about the system properties using them, making them easily accessible from a UML environment.

Acknowledgements

The authors would like to thank the anonymous referees for their insightful and constructive comments and suggestions, which greatly helped improve the contents and readability of the paper. We would also like to acknowledge the work of many ODP experts who have been involved in investigating and addressing the problems of the information specification of ODP systems. Although the views in this paper are the authors' solely responsibility, they could not have been formulated without many hours of detailed discussions with ISO experts on ODP. Thank also to José Meseguer, Narciso Martí-Oliet, and Raúl Romero for their comments on previous versions of this paper and for their ideas on the specification of invariants. This work has been partially supported by Spanish Projects TIC2002-04309-C02-01, TIC2000-0701-C02, and TIC2001-2705-C03.

References

[1] 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.), Procs. of FMOODS'97, Chapman & Hall, 1997, pp. 371–386.

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

[3] H. Bowman, J. Derrick, Viewpoint modelling, in: H. Bowman, J. Derrick (Eds.), Formal Methods for Distributed Processing. A Survey of Object-Oriented Approaches, Cambridge University Press, 2001, pp. 451–475.

[4] H. Bowman, J. Derrick, P. Linington, M.W. Steen, FDTs for ODP, Computer Standards & Interfaces 17 (1995 (Sept.)) 457–479.
[5] 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 (2002) 187–243.
[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude manual (Version 2.1), March 2004. Available at http://www.maude.cs.uiuc.edu.
[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, 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] D. D'Souza, A. Wills, Objects, Components, and Frameworks with UML. The Catalysis Approach, Addison-Wesley, 1999.
[10] F. Durán, J. Herrador, A. Vallecillo, Using UML and Maude for writing and reasoning about ODP policies, Procs. of Policy 2003, IEEE Computer Society Press, 2003, pp. 15–25.
[11] F. Durán, M. Roldán, A. Vallecillo, Invariant-driven strategies for Maude, Proc. of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004), Aachen, Germany, June 2004, To appear in ENTCS.
[12] F. Durán, M. Roldán, Invariant-based control of the execution of Maude specifications: the LTL case, Proc. of PROLE 2004, Málaga, Spain, November 2004.
[13] F. Durán, A. Vallecillo, Formalizing ODP enterprise specifications in Maude, Computer Standards & Interfaces 25 (2) (2003) 83–102.
[14] F. Durán, A. Vallecillo, Specifying the ODP information viewpoint using Maude, in: H. Kilov, K. Baclawski (Eds.), Procs. of Tenth OOPSLA Workshop on Behavioral Semantics, Northeastern University, 2001, pp. 44–57.
[15] J. Dustzadeh, E. Najm, Consistent semantics for ODP information and computational models, Procs. of FORTE/PSTV'97, Chapman & Hall, 1997.
[16] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gaducci, U. Montanari (Eds.), Procs. of 4th International Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier, 2002.
[17] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: J. Goguen, G. Malcolm (Eds.), Software Engineering with OBJ: Algebraic Specification in Action, Kluwer, 2000.
[18] K. Havelund, G. Roşu, Rewriting-based techniques for runtime verification. To appear in Journal of Automated Software Engineering.
[19] J. Hsiang, Refutational theorem proving using term rewriting systems, Artificial Intelligence 25 (1985) 255–300.
[20] IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard 1471, 2000.
[21] ISO/IEC, General Relationship Model. International Standard ISO/IEC 10165-7, ITU-T Recommendation X.725. 1997.
[22] ISO/IEC, RM-ODP, Reference Model for Open Distributed Processing. International Standard ISO/IEC 10746-1 to 10746-4, ITU-T Recommendations X.901 to X.904. 1997.
[23] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Development Process, Addison-Wesley, 1999.
[24] D.R. Johnson, H. Kilov, Can a notation be used to specify an OO system: using Z to describe RM-ODP constructs, in: E. Najm, J.B. Stephani (Eds.), Procs. of FMOODS'96, Chapman and Hall, Paris, 1996 (March), pp. 407–418.
[25] D.R. Johnson, H. Kilov, An approach to a Z toolkit for the Reference Model of Open Distributed Processing, Computer Standards & Interfaces 21 (5) (1999 (December)) 393–402.
[26] P. Kruchten, Architectural blueprints—the "4+1" view model of software architecture, IEEE Software 12 (6) (1995 (November)) 42–50.
[27] P. Linington, RM-ODP: the architecture, in: K. Milosevic, L. Armstrong (Eds.), Open Distributed Processing II, Chapman & Hall, 1995, pp. 15–33.
[28] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in: D.M. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, vol. 9, Kluwer Academic Publishers, 2002, pp. 1–87.
[29] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.
[30] 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, 1998, pp. 18–61.
[31] J. Meseguer, Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems, in: S. Smith, C. Talcott (Eds.), Procs. of FMOODS 2000, Kluwer Academic Publishers, 2000, pp. 89–117.
[32] E. Najm, J.B. Stefani, A formal operational semantics for the ODP computational model, Computer Networks and ISDN Systems 27 (1995) 1305–1329.
[33] E. Najm, J.B. Stefani, Computational models for open distributed systems, in: H. Bowman, J. Derrick (Eds.), Procs. of FMOODS'97, Chapman & Hall, 1997, pp. 157–176.
[34] R. Romero, A. Vallecillo, Formalizing ODP computational viewpoint specifications in Maude, in: Procs. of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004), Monterey, CA, IEEE Computer Society Press, 2004 (September), pp. 147–159.
[35] R. Ross, Principles of the Business Rules Approach, Addison-Wesley, 2003.
[36] M. Steen, J. Derrick, ODP enterprise viewpoint specification, Computer Standards & Interfaces 22 (2) (2000 (September)) 165–189.
[37] J. Zachman, The Zachman Framework: a primer for enterprise engineering and manufacturing. Zachman International, 1997. http://www.zifa.com.

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.

Manuel Roldán received his MSc in Computer Science from the University of Málaga in 1990. Since 1991 he has been an assistant professor at the Department of Computer Science of the University of Málaga, Spain. He has worked in the areas of logic programming and distributed programming. Now his research interests deal with the application of formal methods to software engineering.

Antonio Vallecillo is Associate Professor at the Department of Computer Science of the University of Málaga. His research interests include model-driven software development, componentware, open distributed processing, and the industrial use of formal methods. He holds the BSc and MSc degrees in Mathematics and the PhD degree in Computer Science from the University of Málaga. He is the representative of the Málaga University at ISO and the OMG, and a member of the ACM, the IEEE and the IEEE Computer Society.