ELSEVIER

Theoretical Computer Science 285 (2002) 407–439

Theoretical Computer Science

www.elsevier.com/locate/tcs

A Maude specification of an object-oriented model for telecommunication networks ☆

Isabel Pita*, Narciso Martí-Oliet

Departamento de Sistemas Informáticos y Programación, Universidad Complutense, Madrid, Spain

Abstract

This paper presents an object-oriented model for broadband telecommunication networks, which can be used both for network management and for network planning purposes. The object-oriented model has been developed using the parallel object-oriented specification language Maude, which allows us to define not only structural aspects of the model but also procedural aspects. The reflective properties of rewriting logic are applied to control the rewriting process, using a strategy language that can be specified internally to the logic. Several modeling approaches are compared, emphasizing the definition of the object relationships and the benefits obtained from using reflection as opposed to the extra effort required to control the process at the object level itself.

Keywords: Rewriting logic; Maude; Object-oriented models; Reflection; Strategy language; Telecommunication networks

1. Introduction

Maude is a specification language based on rewriting logic [16], which integrates equational and object-oriented programming in a satisfactory way. Its logical basis facilitates a clear definition of the object-oriented semantics and makes it a good choice for the formal specification of object-oriented systems.

Rewriting logic was first proposed by Meseguer as a unifying framework for concurrency in 1990 [15]. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [14,19], and on the development of the Maude

☆ Partially supported by CICYT, TIC98-0445-C03-02.

* Corresponding author.

E-mail address: ipandreu@sip.ucm.es (I. Pita).

0304-3975/02/$ - see front matter
PII: S0304-3975(01)00364-4

language [17,4,5] but only in the last few years the application of rewriting logic and Maude to the specification of real systems has started, mainly for applications on distributed architectures and communication protocols [20,9,21]. The work that we report on in this paper was the first application of rewriting logic and Maude in this sense, concerning the specification of a telecommunications network model which can be used both for network management and for network planning purposes. The model can then be used to simulate and analyze the protocols and applications of these systems.

We develop a Maude specification of an object-oriented model for broadband telecommunication networks showing the power of the language to specify this kind of systems, and in particular the well-known inheritance relationship and other object relationships, like containment or symmetric relationships ("member-of", "client–server", ...). We have focused on the impact of these relationships in the object creation and deletion processes, and give a general method for their specification. In particular, we propose using subobjects contained in other objects [17] to specify the containment relationship and compare it with the use of object identifiers. The selected application of modeling broadband telecommunication networks is a good choice to illustrate containment and symmetric relationships because they appear in a natural way between the objects of different layers of the network model and between objects of the same layer.

A lot of research has focused on the reflective properties of rewriting logic [2,3]. Reflection allows a system to access its own metalevel, providing a powerful mechanism for controlling the rewriting process. Some general *strategy languages* have been proposed [6,1,2] to define adequate strategies to control rewriting. The important issue is that, thanks to reflection, these languages are based on rewriting and their semantics and implementation are described in the same logic, which allows us to define the strategies by rewriting rules and to implement them in a reflective rewriting logic language like Maude. In this way, control is not an extra-logical addition to the language but remains declaratively inside the logic.

Reflection is used to better exploit resources at different levels. We combine ideas coming from the field of logical reflection with ideas coming from the field of object-oriented reflection [17] by using a *mediator*, a metaobject living in the metalevel and having access to the configuration of the network, for the management of a network. The application of the reflective properties of rewriting logic is illustrated by a process that modifies the demand of a service between two nodes in a network. The strategy language used for controlling the process is based on the one presented in [6], adapting the syntax to the latest version of the language available at the time of writing [5].

Because the object-oriented classes defined in the model are taken from the standard classes specified by the International Telecommunications Union (ITU) for the interconnection of telecommunication systems [11], the resulting model is very general and can be used for the integration of many different applications. Developed as the information model of the object-oriented management protocols used by the systems, it has become the best choice for the database model of the network management centers, to avoid information model translations. For the same reason it is also very appropriate for applications that can interact with this database, such as network planning or the simulation of the network behavior under different situations.

This paper is organized as follows. First, we present some basic notions of rewriting logic and the Maude language that will be used in the application case. Next, we introduce the system that will be specified by defining the model of the network and describing the object-oriented classes defined in the system, and the relationships between them. We discuss the implementation of the relationships in the Maude language and specify some possible queries to the system that are realized by methods implemented in the classes. Then, we introduce the strategy language, and define a new operation needed for the application of reflection. The strategies that control the modification process are presented, and the specification using reflection is compared with the specification without reflection, proposing some improvements to the reflective case.

2. Rewriting logic and the Maude language

We outline here some basic notions of rewriting logic and its implementation in the specification and programming language Maude, needed for the application case. For more information on the subject see [16,17].

A *rewrite theory* $\mathscr{R}$ is defined as a 4-tuple $\mathscr{R}=(\Sigma,E,L,R)$ where (Σ,E) is an equational signature, L is a set of labels, and R is a set of *rewrite rules* of the form $l:[t]_E \to [t']_E$, where $l \in L$, t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of term t modulo the equations E. In order to simplify the presentation, in the following we will not make explicit the equivalence class of terms.

Intuitively, the signature (Σ,E) of a rewrite theory describes a particular structure for the states of a system, and the rewrite rules describe which elementary local transitions are possible in the distributed state by concurrent local transformations.

Rewriting logic is reflective [2], that is, there is a universal rewrite theory $\mathscr{U}$ with a finite number of operations, equations and rules that can simulate any other finitely presentable rewrite theory $\mathscr{R}$ in the following sense: given any two terms t,t' in $\mathscr{R}$ there are corresponding terms $\langle\ \bar{\mathscr{R}},\bar{t}\ \rangle$ and $\langle\ \bar{\mathscr{R}},\bar{t'}\ \rangle$ in $\mathscr{U}$ such that

$$\mathscr{R} \vdash t \to t' \ \Leftrightarrow\ \mathscr{U} \vdash \langle\ \bar{\mathscr{R}},\bar{t}\ \rangle \to \langle\ \bar{\mathscr{R}},\bar{t'}\ \rangle.$$

We will denote the representation (reification) of an object level term ot in the meta-level by $\uparrow ot$ (see [7] for the details of the corresponding definition).

Conditional rewriting logic (that is, equations and rules can be conditional [16]) constitutes the foundation of the specification and programming language Maude. Systems in Maude are built out of basic elements called modules. *Functional modules* are used for the definition of algebraic data types, while *object-oriented modules* are used for the definition of object-oriented classes. An object-oriented module consists of an import list of modules, class declarations, message declarations, and rewrite rules for which the types (called sorts) of the variables appearing in terms are also declared. A class declaration includes the class identifier, attribute identifiers, and the type of each attribute, which can be an algebraic data type defined in a functional module, or an entire configuration defined in another object-oriented module.

An object is represented as a term $\langle\ O : C \mid a_1 : v_1, \ldots, a_n : v_n\ \rangle$, where O is the object's name belonging to a set *Oid* of object identifiers, C is the class identifier, a_i are the names of the object attributes, and v_i are their corresponding values, which are required to belong to the type associated to the attribute.

Rewrite rules represent the real code of the module, that is, the implementation of the method associated to a message received by an object. In general, a rewrite rule has the form

$$\begin{aligned}
&\textbf{crl}\ [l] : M_1 \ldots M_n \langle\ O_1 : C_1 \mid atts_1\ \rangle \cdots \langle\ O_m : C_m \mid atts_m\ \rangle \\
&\Rightarrow \qquad \langle\ O_{i_1} : C'_{i_1} \mid atts'_{i_1}\ \rangle \ldots \langle\ O_{i_k} : C'_{i_k} \mid atts'_{i_k}\ \rangle \\
&\qquad\quad \langle\ Q_1 : D_1 \mid atts''_1\ \rangle \ldots \langle\ Q_p : D_p \mid atts''_p\ \rangle \\
&\qquad\quad M'_1 \ldots M'_q \\
&\textbf{if}\ C
\end{aligned}$$

where $k, p, q, n, m \geqslant 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, C is a rule condition, and l is a label. The result of applying such a rewrite rule is that

- the messages $M_1, \ldots, M_n$ disappear;
- the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change;
- all the other objects O_j vanish;
- new objects $Q_1, \ldots, Q_p$ are created;
- new messages $M'_1, \ldots, M'_q$ are sent.

By convention, the only object attributes $atts_1, \ldots, atts_m$ made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only on the left-hand side of the rule are preserved unchanged, the original values of attributes mentioned only on the right-hand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

With respect to the inheritance relationship, Maude distinguishes two kinds of inheritance: *class inheritance* and *module inheritance*. Class inheritance is directly supported by Maude's order-sorted structure. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses contribute to the structure and behavior of the objects in the subclass, and cannot be modified in the subclass; in addition, the subclass can have new attributes and messages. On the other hand, module inheritance is used for code reuse, allowing the modification of the original code in several ways.

3. The network model

We use Maude to specify an object-oriented model of a broadband telecommunication network. Due to their complexity, these networks are modeled in layers that separate different communication aspects and simplify the global treatment. The selected model is based on the layered structure of broadband networks, which divides

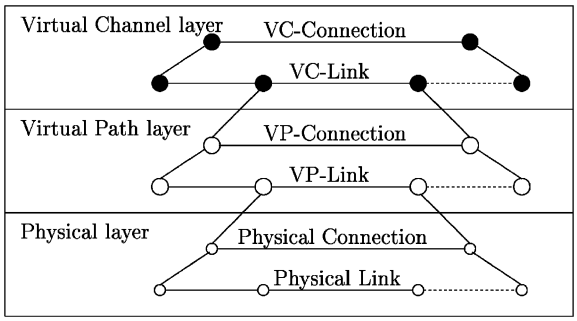

Fig. 1. Network structure.

the network into three logical layers: *physical* layer, *virtual path* (VP) layer, and *virtual channel* (VC) layer, each one related to different network functions (Fig. 1).

3.1. Network objects

The basic objects of the model are *nodes*, *links*, and *connections*, all of which appear in each of the three network layers. Nodes represent the network points where the communication signals are treated. Two kinds of nodes are distinguished:

- Transmission nodes, which have physical and transmission functions and are defined in the physical layer.
- Switching nodes, which have switching and crossconnection functions and are defined in the upper layers.

Links are defined between nodes of the same layer as manageable entities for which the carried information can be accessed at their two end points.

Connections are defined as configurable sequences of links in the same layer. They are used to support the communication services between each pair of users, where each user is related to a unique node. Upper layer links are supported by connections of the layer below, while physical links are supported by the transmission media.

In addition to the three basic objects used to define the topological network structure, there are other components of the network:

- Nodes are formed by pieces of equipment, mainly the node ports, which represent the physical characteristics (number of ports in a termination unit, port capacity, ...) and the cost of the elements defined in a node. The equipment of a transmission node is called *transmission equipment*, whereas the equipment of a switching node is called *switching equipment*.
- The *services* represent the characteristics, in terms of the bandwidth demand, of the communication services (fax, telephone, mail, etc.) provided by the network.

A *network* is formed by a set of links together with the nodes that they join and the corresponding connections between the nodes. A different network is defined for each layer of the model: *physical network*, *VP network*, and *VC network*. The links defined in each network are called, respectively, *physical links*, *VP links*, *VC links*, and analogously for nodes and connections.

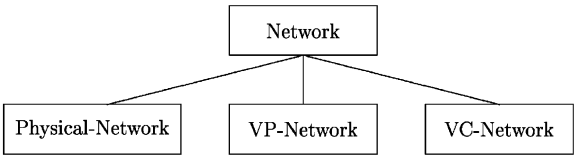

Fig. 2. Network inheritance relationship.

3.2. Network objects relationships

Three types of relationships between network objects are defined in the ITU recommendation X.720: inheritance relationship, containment relationship, and explicit group relationships [11,13].

The *inheritance* relationship captures the common properties of a set of classes by defining a taxonomic order between them. Inheritance appears in the model between generic classes and the respective specialized classes for each layer. For example, Fig. 2 shows a generic class Network that has three subclasses: Physical-Network, VP-Network, and VC-Network. Similar relationships are defined between the generic classes of nodes, links, and connections, and the respective specialized classes for each layer.

The *containment* relationship, also called *object composition*, captures the semantics associated with the "is-part-of" relationship between objects. In its strongest sense, object composition implies that the part cannot exist without its owner, nor can it be shared with another owner. That is, composite objects must be destroyed when the owner object is destroyed, and may only be created as part of the creation process of the owner.

In telecommunication networks it is a common practice to apply the containment relationship to the naming process, that is, to the construction of the object identifier. Object identifiers are the concatenation of the owner object identifier with a unique contained object identifier. The uniqueness of object identifiers for contained objects applies only inside the domain of the owner object. We will apply it to node equipment and its parts. It could also be used to model the relationship between the network and its elements (nodes, links, and connections), and between objects of different layers; however, since it imposes strong restrictions on the model, we have instead chosen explicit group relationships to model them.

Explicit group relationships are object relationships that do not involve the existence of any bonds or permanent associations, do not constrain creation and deletion operations, and allow multiple ownership. These relationships can be of different kinds: client–server, member-of, back-up, etc. The client–server relationship is used to represent the relation between different layers. The member-of relationship is defined naturally between links and connections of the same layer, and between the network and the set of nodes, links, and connections that form it. Back-up relationships are defined between links of the same layer and between connections of the same layer.

4. The network specification

The network objects and their relationships constitute the object-oriented model. The system evolves by requests (queries, modifications, deletions) that produce a chain of messages between the objects until a new stable configuration corresponding to the request is reached.

4.1. The object-oriented classes

In general, each kind of network object described in Section 3.1 gives rise to a class in the object-oriented specification, and in turn each class is specified in a corresponding object-oriented module in Maude. In addition, we need several parametric functional modules to specify algebraic data types such as lists, sets, tuples, etc, which are later on instantiated as sets of node identifiers for example.

The main object class of the model is the class *C-Network* which takes the role of the metaobject described in [17] for broadcasting messages, since it has the list of all the current objects in the system. This class acts as the root class of the model. That is, all accesses to the system are realized through this class or through its subclasses. The class is specialized for each of the network layers in subclasses *C-Physical-Network*, *C-VP-Network*, and *C-VC-Network*, as explained above.

Each network object in the class *C-Network* is characterized by all nodes, links, and connections that belong to the network.

```
class C-Network | NodeSet : NodeOidSet,
                  LinkSet : LinkOidSet,
                  ConnectionSet : ConnectionOidSet .
```

Because order does not matter on these groups of elements, the types of the attributes that define them are *sets* of object identifiers rather than lists of such identifiers. Declaring sets of object identifiers is sufficient in the proposed model, because the relationship between these elements and the network is a member-of relationship. If a containment relationship had been selected, it would have been better to represent the elements by sets of objects and represent them as subconfigurations [17,12], as discussed in Section 4.2.2.

The messages associated to the class *C-Network* represent the queries and updates that can be done in the system.

4.2. Implementation of the object relationships

Three relationships are considered in the model: the inheritance relationship, the containment relationship, and the explicit group relationships. The inheritance relationship is directly supported by class inheritance in Maude, which, as mentioned in Section 2, distinguishes two kinds of inheritance: class inheritance and module inheritance. On the contrary, containment and explicit group relationships are not directly supported by the language, although they can be specified in a natural way within it.

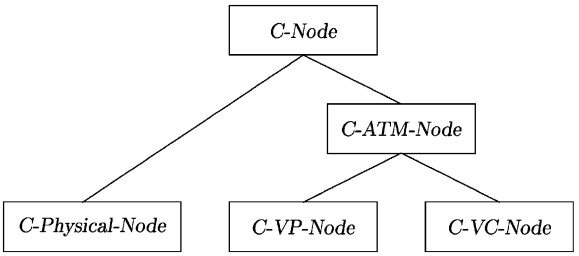

Fig. 3. Complete node inheritance relationship.

4.2.1. Inheritance relationship

Class inheritance is defined between generic classes *C-Network*, *C-Node*, *C-Link*, and *C-Connection*, and the classes specialized for each layer.

In the case of nodes, a new class *C-ATM-Node* is defined to capture the common behavior of the two classes *C-VP-Node* and *C-VC-Node*. Basically, it implements the cost messages, that return the cost of the node equipment, which are computed in a different way for the physical layer and for the other two layers because the equipment is different in each case. The class inheritance relationship is then as represented in Fig. 3.

As the node cost messages must be implemented also in the module for the class *C-Physical-Node*, we could use module inheritance to redefine them from the module defining the class *C-ATM-Node*. However, we have chosen not to do so because almost all of the original messages would need to be redefined, thus defying the purpose of code reuse. Instead, we simply provide the new rules for the corresponding messages in the module defining the class *C-Physical-Node*.

From our experience in this application, we conclude with respect to the design of the Maude language that it would be helpful to have the possibility of defining *abstract* operations in the superclasses in such a way that the implementation of these operations is forced in the subclasses.

4.2.2. The containment relationship

We compare two approaches to the implementation of the containment relationship. The first one considers the object identifiers of the contained objects as attributes of the owner object. The second one defines an attribute of the owner object as a subconfiguration consisting of the contained objects [17,12]. In both cases, the containment relationship impacts on the object creation and deletion rules of Maude. Object creation should not only ensure the uniqueness of object identity, but should also ensure the two properties of the containment relationship:

- The contained object must have a unique owner object.
- The contained object cannot exist if its owner object does not exist.

Object creation was defined in [17] by means of $\mathsf{new}(C \mid Atts)$ messages where C is the name of the class and *Atts* are the attributes. In order to impose uniqueness of object identity, it is assumed that in the initial configuration we have a collection of

different objects in class *ProtoObject* which has an attribute counter, whose value is a natural number used as part of the new object name. The rules for creating objects have the form

$$\textbf{rl}\ [\text{New}] : \text{new}(C \mid Atts)\langle\ O : ProtoObject \mid \text{counter} : N\ \rangle$$
$$\Rightarrow \langle\ O : ProtoObject \mid \text{counter} : N + 1\ \rangle \langle\ O.N : C \mid Atts\ \rangle\ .$$

This scheme has been slightly modified in [18] to force some attributes of the object to always have some fixed initial values. This is done by declaring an initially clause in the class definition which states the attributes that have an initial value as well as that value. For example, we can have the following class declaration:

$$\textbf{class}\ C \mid \mathsf{a1} : t1, \mathsf{a2} : t2, \mathsf{a3} : t3, \mathsf{a4} : t4\ .$$
$$\textbf{initially}\ \mathsf{a1} : v1,\ \mathsf{a2} : v2\ .$$

where ai are the attribute identifiers, *ti* are the attribute types, and *vi* are the attribute values. The rules for creating objects have now the form

$$\textbf{rl}\ [\text{New-ival1}] : \text{new}(C \mid \mathsf{a3} : v3, \mathsf{a4} : v4)\langle\ O : \text{ProtoObject} \mid \text{counter} : N\ \rangle$$
$$\Rightarrow \langle\ O : \text{ProtoObject} \mid \text{counter}: N + 1\ \rangle$$
$$\langle\ O.N : C \mid \mathsf{a1} : v1, \mathsf{a2} : v2, \mathsf{a3} : v3, \mathsf{a4} : v4\ \rangle\ .$$

The first approach to the implementation of the containment relationship declares the identifiers of contained objects as attributes of the owner object, and following the previous scheme we can force them to have an initial value in the creation process. For example, the creation rules for a class X with contained classes $Y1$ and $Y2$ could be defined as follows:

$$\textbf{class}\ Y1 \mid \mathsf{b1} : t1, \mathsf{b2} : t2\ .$$
$$\textbf{initially}\ \mathsf{b1} : v1, \mathsf{b2} : v2\ .$$
$$\textbf{class}\ Y2 \mid \mathsf{c1} : t1', \mathsf{c2} : t2'\ .$$
$$\textbf{initially}\ \mathsf{c1} : w1, \mathsf{c2} : w2\ .$$
$$\textbf{class}\ X \mid \mathsf{a1} : Oid, \mathsf{a2} : Oid, \mathsf{a3} : t3, \mathsf{a4} : t4\ .$$
$$\textbf{initially}\ \mathsf{a1} : Y1, \mathsf{a2} : Y2\ .$$
$$\textbf{rl}\ [\text{New-ival2}] : \text{new}(X \mid \mathsf{a3} : v3, \mathsf{a4} : v4)$$
$$\langle\ O : ProtoObject \mid \text{counter} : N\ \rangle$$
$$\Rightarrow\ \langle\ O : ProtoObject \mid \text{counter} : N + 1\ \rangle$$
$$\langle\ O.N : X \mid \mathsf{a1} : O.N.1, \mathsf{a2} : O.N.2, \mathsf{a3} : v3, \mathsf{a4} : v4\ \rangle$$
$$\langle\ O.N.1 : Y1 \mid \mathsf{b1} : v1, \mathsf{b2} : v2\ \rangle$$
$$\langle\ O.N.2 : Y2 \mid \mathsf{c1} : w1, \mathsf{c2} : w2\ \rangle\ .$$

The result of applying this rule is that the new message disappears, the attribute counter of the *ProtoObject* object is incremented, and three new objects are created in the system: an object of the class declared on the new message, whose identifier is unique in the system because it is formed automatically with the object identifier of the *ProtoObject* object and the value of the counter attribute, and with two attributes that are object identifiers. These two object identifiers are also unique in the system as they are formed with the owner object identifier. The other two objects correspond to the contained objects. This kind of rule guarantees that the contained objects are created with the owner object, and the naming process is the one commonly used in telecommunication networks.

The main differences introduced in this creation process with respect to the previous one are:

- The initially clause defines the class of the contained objects instead of the initial value of the attribute. The object identifier is given automatically by the creation rule. It consists of the combination of the owner object identifier and an identifier given to each contained object by the owner object (in the example above, a number associated to the attribute name).
- The rule for the new message creates the new object as well as the contained objects. Creating the contained objects directly instead of sending more new messages allows us to name the contained objects using the name of the owner object as it is usual in telecommunication networks. By sending new messages, the object would instead be created using the *ProtoObject* class and would therefore have a general identifier.
- No new message is provided for the contained objects, because they cannot be created outside the creation process of their owner objects.

The approach presented in [17] to object deletion uses a message delete(A), with a rule

$$\textbf{rl}\ [\text{Delete}] : \text{delete}(A)\langle\ A : X \mid Atts\ \rangle \Rightarrow \text{null}\ .$$

To maintain the containment relationship, contained objects should be deleted at the time the owner object is deleted. This is simply achieved by sending deletion messages to all contained objects in the owner object deletion rule.

The second approach, which treats subobjects as part of the owner's object state, is simpler. The previous creation rule is defined in this case by:

$$\textbf{class}\ Y1 \mid \text{b1} : t1,\ \text{b2} : t2\ .$$
$$\textbf{initially}\ \text{b1} : v1,\ \text{b2} : v2\ .$$

$$\textbf{class}\ Y2 \mid \text{c1} : t1', \text{c2} : t2'\ .$$
$$\textbf{initially}\ \text{c1} : w1, \text{c2} : w2\ .$$

$$\textbf{class}\ X \mid \text{conf} : \textit{Subconfiguration of}\ \ Y1\ \ Y2, \text{a3} : t3, \text{a4} : t4\ .$$
$$\textbf{rl}\ [\text{New-subconfiguration}] : \text{new}(X \mid \text{a3} : v3, \text{a4} : v4)$$
$$\langle\ O : \textit{ProtoObject} \mid \text{counter} : N\ \rangle$$
$$\Rightarrow \langle\ O : \textit{ProtoObject} \mid \text{counter} : N + 1\ \rangle$$

$$\langle\ O.N : X \mid \mathsf{conf} : \langle\ O.N.1 : Y1 \mid \mathsf{b1} : v1, \mathsf{b2} : v2\ \rangle \\ \langle\ O.N.2\ :\ Y2\ \mid \mathsf{c1}\ :\ w1,\ \mathsf{c2}\ :\ w2\ \rangle\ , \\ \mathsf{a3}\ :\ v3,\ \mathsf{a4}\ :\ v4\ \rangle\ .$$

The second approach emphasizes the fact that one object is part of the other. Nevertheless, the first approach could be more elegant when chains of containment relationships are considered among the network objects since the second one may give rise to a clumsy implementation. The choice between both approaches would be based mainly on the language characteristics. In our proposed model, we have selected the second approach, as it was proposed in [17]. As previously mentioned, the class *C-ATM-Node* defines a containment relationship between the node and the equipment:

$$\textbf{class}\ \textit{C-ATM-Node}\ \mid\ \mathsf{EqConm}\ :\ \textit{Eq-Conm-Object}\ , \\ \mathsf{UtilL}\ :\ \textit{EQCUtilList}\ .$$

where *Eq-Conm-Object* is an object subsort that defines the switching equipment of the node.

4.2.3. Explicit group relationships

Explicit group relationships are modeled directly by defining a set of object identifiers as the value of an attribute. The only restriction imposed on the object creation process is that the objects appearing in the attribute value should also exist in the system. Following this idea, a network object, related by a member-of relationship to the set of nodes, links, and connections that form it, is initially created with empty sets of such elements. Then, the nodes, links, and connections are added to the network using messages AddNodeTo_, AddLinkTo_Between_and_ and AddConnectionTo_Between_and_. The rules implementing these messages create the objects at the same time that they are introduced as elements of the network object.

$$\textbf{rl}\ [\text{Add-node}] : \mathsf{AddNodeTo}\ N \\ \Rightarrow (\mathsf{new}\ \textit{C-Node}\ \mathsf{ack}\ N\ \mathsf{req}\ X)\ \mathsf{AddNodeToNetwork}(N,X)\ .$$

$$\textbf{rl}\ [\text{Add-node}] : (\mathsf{to}\ N : X\ \mathsf{is}\ No)\ \mathsf{AddNodeToNetwork}(N,X) \\ \langle\ N : \textit{C-Network} \mid \mathsf{NodeSet} : Ns\ \rangle \\ \Rightarrow \langle\ N : \textit{C-Network} \mid \mathsf{NodeSet} : No\ Ns\ \rangle\ .$$

The node object is created using the new message, because the node identifier is given in the system name space controlled by the object of class *ProtoObject*, as the explicit group relationships do not create a naming space like the containment relationship. We use the new message with acknowledgement [17] that returns the message (to $N : X$ is *No*) when it is attended, where X is a message identifier that is generated when the message is sent and *No* is the object identifier assigned to the new object. Variable instantiation on the right-hand side of a rule is supported by Maude 2.0. Once the node object has been created in the system, the second rule is used to introduce it in the corresponding network.

In the case of links and connections, which have additional group relationships declared with other objects, the rules also check that all necessary requirements are satisfied. For example, the rules for the message (AddLinkTo N Between $No1$ and $No2$) that adds a link to a physical network must check that the two associated nodes have previously been added to the network.

```
rl [Add-link] : (AddLinkTo N Between No1 and No2)
   < N : C-Physical-Network | NodeSet : No1 No2 Ns >
=> < N : C-Physical-Network | >(new C-Physical-Link ack N req X)
   AddLinkToNetwork(N,X,No1,No2) .

rl [Add-link] : (to N : X is L) AddLinkToNetwork(N,X,No1,No2)
   < N : C-Physical-Network | LinkSet : Ls >
   < L : C-Physical-Link | >
=> < N : C-Physical-Network | LinkSet : L Ls >
   < L : C-Physical-Link | Nodes : << No1; No2 >> > .
```

Using this object creation process it is guaranteed that the resulting network topology is consistent, and there will not be attribute values related to objects that do not exist. Consistency should be maintained in the deletion process by imposing that objects related to other objects cannot be eliminated without deleting previously the value of the attribute on the related object. Notice that in explicit group relationships it is enough to delete the value of an attribute and it is not necessary to delete the complete object like in the containment relationship.

4.3. Query messages

Two query messages are presented: the first one obtains the value of an attribute that is directly defined in the model, and the second one calculates the required value from existing attribute values. In both cases, the query is sent to a network object, as the root of the system, which in turn sends messages to the appropriate network components to obtain the required values. Then, it summarizes the information contained in the returning messages in order to forward a unique message to the external object that made the original query.

The message LinkLoad?(O,N,L) obtains the load of a selected link L in a network N and replies to an external object O. Two rules are used to implement this message:

```
rl [Link-load] : LinkLoad?(O,N,L)< N : C-Network | LinkSet : L Ls >
=> < N : C-Network | >(L.Load reply to N and O) .
rl [Link-load] : (to N and O,L.Load is Ld)
   < N : C-Network | LinkSet : L Ls >
=> < N : C-Network | >(To O LinkLoad L is Ld in N) .
```

where the used variables are declared by

```
var O : Oid .
var L : LinkOid .
var Ld : Nat .
var Ls : LinkOidSet .
var N : NetworkOid .
```

The first rule forwards the query to the appropriate link object. This rule only applies when the link L appears in the network set of links, given by the LinkSet attribute, as it is required on the left-hand side of the rule by declaring the value of the LinkSet attribute to be $L\ Ls$, where L is the link of the message and Ls is a set of link identifiers. The second rule receives the acknowledgement of the link object indicating the value Ld of the Load attribute, and forwards the corresponding acknowledgement to the object that made the original query. In this way, the only system objects that interact with external objects are the *C-Network* objects.

The messages (*L*.Load reply to *N* and *O*) and (to *N* and *O*, *L*.Load is *Ld*) obtain the value of an attribute directly defined in a class that is not hidden — in this case the value of the Load attribute of a link L — and send it to the network object N in the way described in [17], although not implemented yet in Maude. The external object O is necessary in the message to identify the appropriate return address.

The message NodeLinks?(O,N,No) obtains all the links in the network N that have a selected node No as their origin or as their destination. The entire system should be investigated. A message is broadcast to all links in the network N, and the links ending in node No are collected and forwarded to the external object O. The corresponding rules are

```
vars No M1 M2 : NodeOid .
var Ns : NodeOidSet .
var Ls1 : LinkOidSet .

rl [Node-links] : NodeLinks?(O,N,No)
   < N : C-Network | NodeSet : No Ns, LinkSet : Ls >
=> FindLink(O,N,No,Ls,null)< N : C-Network | > .

crl [Find-link] : FindLink(O,N,No,L Ls,Ls1)
   < L : C-Link | Nodes :<< M1;M2 >> >
=> < L : C-Link | > FindLink(O,N,No,Ls,L Ls1)
if (M1 == No) or (M2 == No) .

crl [Find-link] : FindLink(O,N,No,L Ls,Ls1)
   < L : C-Link | Nodes: << M1;M2 >> >
```

$\Rightarrow \langle\ L : C\text{-}Link \mid \rangle$ FindLink($O, N, No, Ls, Ls1$)
if $(M1 = / = No)$ **and** $(M2 = / = No)$.

rl [Find-link] : FindLink($O, N, No,$ null, Ls)
$\Rightarrow$ (To N and O NodeLinks No are Ls) .

rl [Node-links] : (To N and O NodeLinks No are Ls) $\langle\ N : C\text{-}Network \mid \rangle$
$\Rightarrow \langle\ N : C\text{-}Network \mid \rangle$ (To O in N Node No Links Ls) .

The first and last rules are implemented in the object-oriented module defining the class *C-Network*. Two attributes of the *C-Network* class are used in the first rule: NodeSet, which declares the set of nodes that belong to the network and whose value for the network N includes the node identifier No given in the message and a set of node identifiers Ns; and LinkSet, which declares the set of links that belong to the network and is represented by a set of link identifiers Ls.

The three rules implementing the message FindLink belong to the object-oriented module defining the class *C-Link*. Once the network object N broadcasts the query to all the associated links, the rules for FindLink collect the links that have the given node as one of its endpoints by adding them one at a time to the set of link identifiers defined in the last parameter of the FindLink($O, N, No, Ls, Ls1$) message, and send a new message to the remaining links in the set Ls until there are no more links in this set. When all the network links have been treated, a message (To N and O NodeLinks No are Ls) with the set Ls of all links that fulfil the condition is sent to the network object N, which in turn replies to the external object O.

4.4. Modification messages

The message ChDemand($O, N, No1, No2, \ll S; D \gg$) changes the communication demand of the connection between nodes $No1$ and $No2$ in the network N, by adding the bandwidth demand D of service S (expressed as a pair $\ll S; D \gg$) to the existing connection demand. If the demand is indeed changed, the message (To O AckChDemand $No1$ and $No2$ in N) is sent to the external object; otherwise, a message indicating the reason why the change has not been done is sent.

Assuming that

- the network topology is correct, that is, the nodes defined as endpoints of the links and the links defined on a connection are part of the network configuration, and the application of rewriting rules at the object level guarantees the correction of network topologies, and that
- there is no restriction in adding ports to a node if the port capacity is supported by the node,

only two error messages are used: (To O NoConnectionBetween $No1$ and $No2$ in N) for the case in which there is no connection defined between the given nodes, and (To O ServiceCapacityNoSupported) for the case in which there is no port of the needed

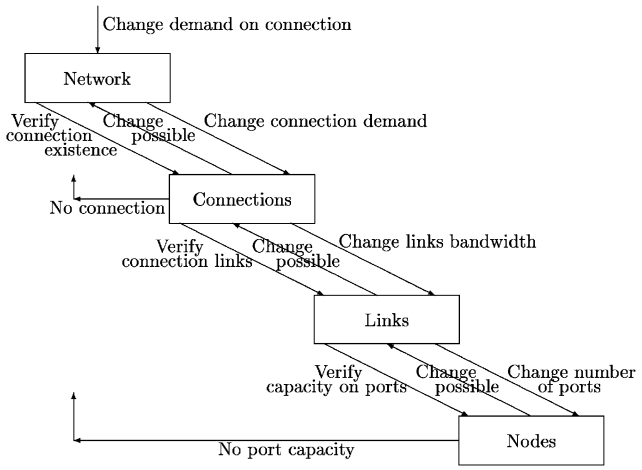

Fig. 4. Modification process.

capacity in one of the nodes traversed by the connection. For simplicity, we assume that if there are ports of a given capacity in a node of an upper layer, then there are ports of the same or greater capacity on the nodes of the layers below.

The protocol followed by the modification process in one layer is sketched in Fig. 4. First, messages are sent to the connection, links, and nodes involved in the requested change, to verify whether the modification is possible. If the returned messages indicate that the modification can indeed be done, then the modification process starts changing the service demand on the connection, the bandwidth required on the links that support the connection, and the number of ports of the traversed nodes; otherwise, the corresponding error message is sent to the external object O through the network object. In this version of the modification process, the ChDemand operations are only executed serially, that is, the network object does not accept a new request until the last one has been answered.

The network object N forwards the query to its set of connections, using the message $\mathsf{ComMod}(O, N, No1, No2, Cs, \ll S; D \gg)$ by means of the rule

var D : *Nat* .
var S : *ServiceOid* .
vars $No1\ No2$: *NodeOid* .
var Cs : *ConnectionOidSet* .

rl [Ch-demand] : $\mathsf{ChDemand}(O, N, No1, No2, \ll S; D \gg)$
$\langle\ N$: *C-Network* | NodeSet : $No1\ No2\ Ns$, ConnectionSet : $Cs\ \rangle$
$\Rightarrow \langle\ N$: *C-Network* | $\rangle \mathsf{ComMod}(O, N, No1, No2, Cs, \ll S; D \gg)$.

The set of connections is explored one connection at a time. If a connection between the required nodes *No1*, *No2* is found, then a new message is sent to the list of links

that form the connection to verify whether the modification is possible; otherwise, an error message is sent to the network object N, which forwards it to the external object O. A different and more complex approach could be to create the connection when it does not exist. The corresponding rules are

vars $M1\ M2$: *NodeOid* .
var Ll : *LinkOidList* .

crl [Com-mod1] : ComMod($O, N, No1, No2, C\ Cs, \ll S; D \gg$)
$\langle\ C$: *C-Connection* | Nodes : $\ll M1;\ M2 \gg$, LinkList : $Ll\ \rangle$
$\Rightarrow \langle\ C$: *C-Connection* | $\rangle$ LinkListLoadReq($O, N, C, \ll S; D \gg, Ll$)
if $((M1 == No1)$ **and** $(M2 == No2))$ **or**
$((M2 == No1)$ **and** $(M1 == No2))$.

crl [Com-mod1] : ComMod($O, N, No1, No2, C\ Cs, \ll S; D \gg$)
$\langle\ C$: *C-Connection* | Nodes: $\ll M1; M2 \gg\ \rangle$
$\Rightarrow \langle\ C$: *C-Connection* | $\rangle$ ComMod($O, N, No1, No2, Cs, \ll S; D \gg$)
if $((M1 = / = No1)$ **or** $(M2 = / = No2))$ **and**
$((M2 = / = No1)$ **or** $(M1 = / = No2))$.

rl [Com-mod1] : ComMod($O, N, No1, No2$, null, $\ll S; D \gg$)
$\Rightarrow$ (To N and O NoConnectionBetween *No1* to *No2*) .

rl [error] : (To N and O NoConnectionBetween *No1* to *No2*)
$\langle\ N$: *C-Network* | $\rangle$
$\Rightarrow \langle\ N$: *C-Network* | $\rangle$ (To O NoConnectionBetween *No1* to *No2* in N) .

If a connection between the required nodes is found in the network, the verification process continues with the LinkListLoadReq($O, N, C, \ll S; D \gg, Ll$) message, which goes over the list of links that form the connection, sending messages to its ending nodes to verify their port capacity. The rules are

vars $L\ L1$: *LinkOid* .

rl [Link-list-req] : LinkListLoadReq($O, N, C, \ll S; D \gg, L\ Ll$)
$\langle\ L$: *C-Link* | Nodes : $\ll No1;\ No2 \gg\ \rangle$
$\Rightarrow \langle\ L$: *C-Link* | $\rangle$PortNodeReq($O, N, C, L, \ll S; D \gg, No1$)
PortNodeReq($O, N, C, L, \ll S; D \gg, No2$)
LinkListLoadReq2($O, N, C, \ll S; D \gg, Ll, L, No1, No2$) .

```
rl [Link-list-req] : LinkListLoadReq2(O,N,C,≪ S;D ≫,L Ll,L1,No1,No2)
   (To L1 and O and N and C and ≪ S;D ≫ PortInNode No1)
   (To L1 and O and N and C and ≪ S;D ≫ PortInNode No2)
   ⟨ L : C-Link | Nodes : ≪ M1; M2 ≫ ⟩
⇒ ⟨ L : C-Link | ⟩ PortNodeReq(O,N,C,L,≪ S;D ≫,M1)
   PortNodeReq(O,N,C,L,≪ S;D ≫,M2)
   LinkListLoadReq2(O,N,C,≪ S;D ≫,Ll,L,M1,M2) .

rl [Link-list-req] : LinkListLoadReq2(O,N,C,≪ S;D ≫,nil,L,No1,No2)
   (To L and O and N and C and ≪ S;D ≫ PortInNode No1)
   (To L and O and N and C and ≪ S;D ≫ PortInNode No2)
⇒ (To C and O and N and ≪ S;D ≫ LinksVerified) .
```

A new message LinkListLoadReq2 is used to go over the list and summarize the acknowledgement message of the ending nodes. This auxiliary message is used only for the implementation of this process and it will not be invoked by any other object. In this case, an encapsulation mechanism would be very useful, as it would avoid the possible misuse of this message by any other object. It is possible to build an iterator *at the metalevel*, as we do in Section 5, that goes over generic lists and simplifies the rules at the object level, but the language does not provide this kind of operator at the object level.

Finally, the message PortNodeReq($O,N,C,L,\ll S;D\gg,No$) is sent to the node No to verify whether it has ports to carry at least the bandwidth demanded by service S, thus finishing the verification process. The implementation of this rule in the physical layer is different from those in the other two layers, because of the different equipment defined in each kind of node. The rules for physical nodes are:

```
vars Cap Cp : Nat .
var E : EquipmentOid .

crl [Port-node-req] : PortNodeReq(O,N,C,L,≪ S;D ≫,No)
⟨ No : C-Physical-Node | EqTrans : ⟨ E : C-Eq-Trans | Capacity : Cp ⟩⟩
   ⟨ S : C-Service | Capacity : Cap ⟩
⇒ ⟨ No : C-Physical-Node | ⟩ ⟨ S : C-Service | ⟩
    (To L and O and N and C and ≪ S;D ≫ PortInNode No)
if (Cap <= Cp) .

crl [Port-node-req] : PortNodeReq(O,N,C,L,≪ S;D ≫,No)
   ⟨ No : C-Physical-Node | EqTrans : ⟨ E : C-Eq-Trans | Capacity : Cp ⟩⟩
```

$\langle\ S\ :\ C\text{-}Service\ |\ \mathsf{Capacity}\ :\ Cap\ \rangle$
$\Rightarrow \langle\ No\ :\ C\text{-}Physical\text{-}Node\ |\ \rangle\ \langle\ S\ :\ C-Service\ |\ \rangle$
(To N and O NoPortCapacity Cp Node No)
if $(Cap > Cp)$.

The rules check that the port capacity Cp is enough to carry one communication. Notice that the value of the EqTrans attribute is a configuration consisting of a transmission equipment object instead of an object identifier. The reason is that a containment relationship has been declared between the nodes and the associated equipment, and this relationship is implemented using configurations as the value of the corresponding attributes as explained in Section 4.2.2.

Once it is verified that all the links and nodes in the connection can support the change of demand, the connection object starts the modification process by means of the message ComMod2($O, N, C, \ll S; D \gg, Dl$)

rl [Links-verified] : (To C and O and N and $\ll S; D \gg$ LinksVerified)
$\langle\ C\ :\ C\text{-}Connection\ |\ \mathsf{DemandList}\ :\ Dl\ \rangle$
$\Rightarrow \langle\ C\ :\ C\text{-}Connection\ |\ \rangle$ ComMod2($C, \ll S; D \gg, Dl$)
(To N and O ConnectionModified C) .

where Dl denotes a list of tuples, each one consisting of a service identifier and the number of communications of this service carried by the connection.

The ComMod2 message goes over the demand list Dl looking for a tuple whose service identifier is the one used in the message. If it exists, the demand is changed; otherwise, the service is added to the demand list. In both cases a message to change the links is sent.

vars $S\ S1\ S2$: $ServiceOid$.
vars $D1\ D2$: Nat .
vars $Dl1\ Dl2$: $DemandList$.

rl [Com-mod2] : ComMod2($C, \ll S; D1 \gg, \ll S; D2 \gg Dl$)
$\langle\ C\ :\ C\text{-}Connection\ |\ \mathsf{DemandList}: Dl1 \ll S; D2 \gg Dl2,\ \mathsf{LinkList}: Ll\ \rangle$
$\langle\ S\ :\ C\text{-}Service\ |\ \mathsf{Capacity}: Cp\ \rangle$
$\Rightarrow \langle\ C\ :\ C\text{-}Connection\ |\ \mathsf{DemandList}\ :\ Dl1 \ll S; D1 + D2 \gg Dl2\ \rangle$
$\langle\ S\ :\ C\text{-}Service\ |\ \rangle$ChLinkListLoad($Ll, D1 * Cp$) .

crl [Com-mod2] : ComMod2($C, \ll S1; D1 \gg, \ll S2; D2 \gg Dl$)
$\langle\ C\ :\ C\text{-}Connection\ |\ \rangle$
$\Rightarrow \langle\ C\ :\ C\text{-}Connection\ |\ \rangle$ComMod2($C, \ll S1; D1 \gg, Dl$)
if $(S1 = /= S2)$.

```
rl [Com-mod2] : ComMod2(C, ≪ S; D1 ≫, nil)
  ⟨ C : C-Connection | DemandList : Dl, LinkList : Ll ⟩
  ⟨ S : C-Service | Capacity : Cp ⟩
⇒ ⟨ C : C-Connection | DemandList : Dl ≪ S; D1 ≫ ⟩
  ⟨ S : C-Service | ⟩ChLinkListLoad(Ll, D1 * Cp) .
```

The message ChLinkListLoad(Ll, R) changes the load of the links in the list Ll adding the bandwidth R. Like in the PortNodeReq message, the rules that implement this message in the physical layer are different from the rules in the other two layers. In the former they are

```
vars Ld R : Nat .

rl [ChLinkListLoad] : ChLinkListLoad(L Ll, R)
  ⟨ L : C-Physical-Link | Nodes : ≪ No1; No2 ≫, Load : Ld ⟩
⇒ ⟨ L : C-Physical-Link | Load : Ld + R ⟩ChPortNode(No1, R)
  ChPortNode(No2, R) ChLinkListLoad(Ll, R) .

rl [ChLinkListLoad] : ChLinkListLoad(nil,R) ⇒ nil .
```

Finally, the message ChPortNode(No, R) changes the number of ports used in node No in accordance with the new bandwidth R. Once again, the rule for the physical layer is different from the rule for the other two layers.

```
rl [ChPortNode] : ChPortNode(No, R) < No : C-Physical-Node |
  EqTrans : ⟨ E : C-Eq-Trans | Capacity : Cp ⟩, Used : U ⟩
⇒ ⟨ No : C-Physical-Node | Used : U + R div Cp ⟩ .
```

5. Reflection and the internal strategy language

The reflective properties of rewriting logic are used in Maude to control the rewriting process by means of strategies, since keeping the control at the object level would complicate unnecessarily the specification and confuse control issues with specification issues. In Maude, strategies can be made *internal* to rewriting logic. That is, they can also be defined by means of rewrite rules. In fact, there is great freedom for defining different strategy languages inside Maude [2,6]. This can be done in a completely user-definable way, so the users are not limited by a fixed and closed strategy language.

A methodology for defining a strategy language is outlined in [2,6]. First, a kernel is defined stating how rewriting in the object level is accomplished at the metalevel. In particular, Maude supports a strategy language kernel which defines the operation

```
op metaApply : Module Term Qid Substitution MachineInt → ResultTriple? .
```

A term metaApply(R,t,l,σ,n) is evaluated by converting the metaterm t to the term it represents, reducing it using the equations of the module R, and matching the resulting term against all rules with the given label l, partially instantiated with σ. The first n successful matches are discarded, and if there is an $(n+1)$th successful match its rule is applied, and the resulting term is converted to a metaterm and returned with the corresponding sort or kind, and the matching substitution; otherwise, failure is returned.

The strategy language STRAT defined in [6] defines sorts *Strategy* and *StrategyExp* for rewriting strategies and strategy expressions, respectively, and extends the kernel with operations to compose strategies. In this paper, however, we will not use their solution tree structure and we have adapted the syntax to the latest version of the language available at the time of writing [5]. In particular, the operations that will be used in our application are:

- operations defining basic strategies:

op idle: $\rightarrow$ *Strategy* .

op apply: *Label* $\rightarrow$ *Strategy* .

op rew_in_with_: *Term Module Strategy* $\rightarrow$ *StrategyExp* .

op failure: $\rightarrow$ *StrategyExp* .

- operations that compose strategies:

op _;_: *Strategy Strategy* $\rightarrow$ *Strategy* .

op _;;_orelse_: *Strategy Strategy Strategy* $\rightarrow$ *Strategy* .

op _andthen_: *StrategyExp Strategy* $\rightarrow$ *StrategyExp* .

These operations satisfy the following equations:

eq rew T in M with $(S;S')$=(rew T in M with S) andthen S' .

eq (rew T in M with idle) andthen S=rew T in M with S .

eq failure andthen S=failure .

ceq rew T in M with apply(L) =
if (metaApply(M,T,L, none,0)) :: *ResultTriple* then
rew getTerm(metaApply(M,T,L, none,0)) in M with idle
else failure .

ceq rew T in M with $(S;;S''$ orelse $S')$ =
if rew T in M with S== failure then rew T in M with S'
else (rew T in M with S) andthen S'' .

A strategy expression initially has the form rew T in M with S, meaning that we have to apply the strategy S to the metaterm T in module M. This strategy expression is then reduced by means of the equations in module M to an expression of the form

rew T' in M with S', where T' is the term already calculated and S' is the remaining strategy. If the reduction process succeeds, a strategy expression of the form rew T'' in M with idle is reached, where T'' represents the solution term obtained; otherwise, the strategy expression failure is generated.

Controlling the order in which a transaction consisting of a sequence of rewriting steps will take place at the object level requires the use of the concatenation operation on strategies, and of the operation that applies a rule at the object level. The idea is that the strategy concatenates the rewriting rules in the order in which they should be used in the process by means of the operation _;_. Notice that rules are applied at the object level as they are required by the strategy, and that the strategy controls the failure situations when there is no rule to apply by means of the equation that defines the apply operation using metaApply.

We add a new operation on strategies to the STRAT language of [6] in order to apply a strategy over a list of objects:

op Iterate: *Strategy* → *Strategy* .

eq Iterate(S) = S ;; Iterate(S) orelse idle .

Intuitively, the strategy Iterate(S) is very general and simply applies several times a strategy S, finishing when this strategy can no longer be applied. Following the original operations definition in the strategy language STRAT [6], we have defined the Iterate operation by means of an equation; however, since the equations should be terminating the user must ensure that for his concrete application the reduction process finishes. For our purposes of applying a given rule to a list of objects, we wish to apply the same strategy but each time to a different object. That is, we want to iterate over the given list. If the strategy cannot be applied more than once to a given object (for example, when one of its effects is the removal of the object from the list), there is nothing else to consider. However, if this is not the case, as in our application, we need to make sure that each object in the list is treated exactly once. To accomplish this, we use a list parameter in the rewriting rules at the object level, keeping track of the list of objects that have not been treated yet (see the rules LinkListLoadReq in Section 6.1). When all the objects have matched the rule, the strategy S will fail to apply and the _;;_orelse_ operation will finish with the idle strategy.

A different possibility is to keep track of the list of objects at the level of the strategy language, instead of doing it at the object level. This can be done, for example, in the strategy language as Clavel and Meseguer describe in [2], where they use substitutions and bindings to pass information from one level to the other. We have decided not to do so in order to keep the strategy language simpler, and to reuse as much as possible the rules of the nonreflective application.

6. Network specification using reflection

In this section we use reflection to specify the modification process sketched in Fig. 4.

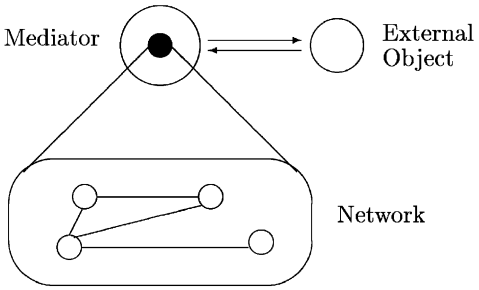

Fig. 5. Metalevel mediator for a network.

All the different objects that constitute a network form a configuration in the sense described in [17], that is, they form a distributed state of a concurrent object-oriented system at the object level. To control a network, we add *at the metalevel* a class of *metaobjects* called *mediators*:

class *Mediator* | Config : *Network-Configuration* .

where *Network-Configuration* is a subsort of the Maude predefined sort *Term*; that is, the value of the attribute Config is a metaterm representing *at the metalevel* the multiset of objects that constitute a network.

A *mediator* has as the value of its attribute Config a metaterm of the form $\uparrow C$, where C denotes the configuration of the network managed by the mediator. We assume that all the control of the network is done through its mediator, which takes the role of the network at the object level. That is, the messages addressed to the network from an external object are processed by the mediator, that will also send back the corresponding answer. This situation is depicted in Fig. 5, and constitutes a novel application of ideas from logical reflection to the description of object-oriented group reflection already sketched by Meseguer in [17].

6.1. Strategies and rules for the reflective modification process

The first approach to control the modification process through the mediator follows the protocol described in Fig. 4. We define two conditional rules in the metalevel. The first one is applied when the modification is possible, and the other treats the error cases. The idea is to reduce the strategy expression that modifies the demand on the network configuration as much as possible, matching the resulting strategy expression against the left-hand side of the matching condition in the rules below. Matching equations are mathematically interpreted as ordinary equations; however, they may have new variables, that do not appear in the left-hand side of the corresponding rule. In the execution of a matching equation, these new variables become instantiated by matching the left-hand side of the condition against the right-hand side. The configuration is then

changed according to the given matching or an error message is sent to the external object.

```
crl [ChDemand1] : ChDemand(O, N, No1, No2, ≪ S ; D ≫ )
    ⟨ N : Mediator | Config : T ⟩
⇒ ⟨ N : Mediator | Config : T' ⟩
  (To O AckChDemand No1 and No2 in N)
if rew T' in M-NET with idle := rew T_C in M-NET with stratChDemand1 .

crl [ChDemand1] : ChDemand(O, N, No1, No2, ≪ S ; D ≫ )
  ⟨ N : Mediator | Config : T ⟩
⇒ ⟨ N : Mediator | ⟩ (To O NoChDemand No1 and No2 in N)
if rew T_C in M-NET with stratChDemand1==failure .
```

where the metaterm T represents a correct state of the network, T_C denotes the configuration $--[T, \uparrow \mathsf{ConnectionReq}(O, N, No1, No2, \ll S; D \gg)]$, M-NET is the module that defines the network, and stratChDemand1 denotes the strategy

```
apply(ConnectionReq); apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq); apply(LinkListLoadReq));
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode)) .
```

The constants ConnectionReq, LinkListLoadReq, PortNodeReq, ChConnection, ChLinkListLoad, and ChPortNode are the labels of the rules defined at the object level, which are described below for the reflective case.

When a ChDemand message is received by the network mediator, in order to apply one of the ChDemand1 rules above, the condition of the rule is evaluated. This requires reducing the strategy expression

```
rew T_C in M-NET with stratChDemand1 .        (1)
```

using the equations of the strategy language. First, it tries to apply the ConnectionReq rule at the object level, which verifies the existence of connection between nodes *No1* and *No2* in network N.

```
crl [ConnectionReq] : ConnectionReq(O, N, No1, No2, ≪ S ; D ≫ )
  ⟨ C : C-Connection | Nodes : ≪ M1 ; M2 ≫, LinkList : Ll ⟩
⇒ ⟨ C : C-Connection | ⟩ LinkListLoadReq(O, N, C, ≪ S ; D ≫, Ll )
if ((M1 == No1) and (M2 == No2)) or
  ((M2 == No1) and (M1 == No2)) .             (2)
```

If the connection exists, then the initial strategy expression (1) is reduced to

```
rew getTerm(metaApply(M-NET,T_C,ConnectionReq,none,0)) in M-NET
with apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq);
        apply(LinkListLoadReq));
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode)) .
```
(3)

Otherwise, expression (1) reduces to failure, because the rule ConnectionReq cannot be applied, and the failure is propagated through the rest of the expression. In this case, the condition of the second rule is fulfilled and the message (To O NoChDemand $No1$ and $No2$ in N) is sent to the external object O.

If there is no error, the strategy expression (3) reduces to

```
rew _[T, ↑ LinkListLoadReq(O,N,C,≪ S;D ≫,Ll)] in M -NET
with apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq);
        apply(LinkListLoadReq));
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode)) .
```

The LinkListLoadReq rules send messages to the nodes of the link to verify whether they support the demanded capacity by means of the PortNodeReq rule. Once all the links have been verified, the connection object starts changing its service demand.

```
rl [LinkListLoadReq] : LinkListLoadReq(O,N,C,≪ S;D ≫,L Ll)
  ⟨ L : C-Link | Nodes : ≪ No1;No2 ≫ ⟩
⇒ ⟨ L : C-Link | ⟩PortNodeReq(S,No1)
  PortNodeReq(S,No2) LinkListLoadReq(O,N,C,≪ S;D ≫,Ll) .
```
(4)

```
rl [LinkListLoadReq] : LinkListLoadReq(O,N,C,≪ S;D ≫,nil)
  ⟨ C : C-Connection | DemandList : Dl ⟩
⇒ ⟨ C : C-Connection | ⟩ChConnection(C,≪ S;D ≫,Dl) .

crl [PortNodeReq] : PortNodeReq(S,No)
  ⟨ No : C-Node | Eq : ⟨ Et : C-Eq-Trans | Capacity : Cap ⟩⟩
  ⟨ S : C-Service | Capacity : Cp ⟩
```

$\Rightarrow \langle\ No : C\text{-}Node \mid \rangle \langle\ S : C\text{-}Service \mid \rangle$
if $(Cp <= Cap)$.

The application of the recursive equation Iterate continues until either some node produces an error or all the links on the connection have been treated.

In the first case, the strategy expression is reduced to failure as the Iterate operation finishes without sending a ChConnection message in the object level and then the ChConnection rule cannot be applied. Otherwise, the modification part of the process is accomplished using the remaining strategy

```
Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode))
```

which cannot fail because the verification part guarantees that the modification is possible.

The ChConnection message goes over the demand list looking for a tuple whose service identifier is the one used in the message. The Iterate operation is used until either the service is found in the demand list or the demand list is nil. The Iterate operation is then reduced to idle and the ChLinkListLoad rule can be applied. It is equivalent to the ComMod2 message of the nonreflective case.

rl [ChConnection] : ChConnection$(C, \ll S; D1 \gg, \ll S; D2 \gg Dl)$
$\langle\ C : C\text{-}Connection \mid \mathsf{DemandList} : Dl1 \ll S; D2 \gg Dl2,$
$\mathsf{LinkList} : Ll\ \rangle$
$\langle\ S : C\text{-}Service \mid \mathsf{Capacity} : Cp\ \rangle$
$\Rightarrow \langle\ C : C\text{-}Connection \mid \mathsf{DemandList} : Dl1 \ll S; D1 + D2 \gg Dl2\ \rangle$
$\langle\ S : C\text{-}Service \mid \rangle$ChLinkListLoad$(Ll, D1 * Cp)$.

crl [ChConnection] : ChConnection$(C, \ll S1; D1 \gg, \ll S2; D2 \gg Dl)$
$\langle\ C : C\text{-}Connection \mid \rangle$
$\Rightarrow \langle\ C : C\text{-}Connection \mid \rangle$ChConnection$(C, \ll S1; D1 \gg, Dl)$
if $(S1 = / = S2)$.

rl [ChConnection] : ChConnection$(C, \ll S; D1 \gg, \mathsf{nil})$
$\langle\ C : C\text{-}Connection \mid \mathsf{DemandList} : Dl, \mathsf{LinkList} : Ll\ \rangle$
$\langle\ S : C\text{-}Service \mid \mathsf{Capacity} : Cp\ \rangle$
$\Rightarrow \langle\ C : C\text{-}Connection \mid \mathsf{DemandList} : Dl \ll S; D1 \gg\ \rangle$
$\langle\ S : C\text{-}Service \mid \rangle$ChLinkListLoad$(Ll, D1 * Cp)$.

The ChLinkListLoad rule is slightly modified from the nonreflective case. Now the ChLinkListLoad message is not generated for the empty list.

crl [ChLinkListLoad] : ChLinkListLoad$(L\ Ll, R)$

$\langle\ L : C\text{-}Physical\text{-}Link \mid \mathsf{Nodes} : \ll No1; No2 \gg, \mathsf{Load} : Ld\ \rangle$
$\Rightarrow \langle\ L : C\text{-}Physical\text{-}Link \mid \mathsf{Load} : Ld + R\ \rangle \mathsf{ChPortNode}(No1, R)$
$\mathsf{ChPortNode}(No2, R)\ \mathsf{ChLinkListLoad}(Ll, R)$
if $(Ll = /= \mathsf{nil})$.

crl [ChLinkListLoad] : $\mathsf{ChLinkListLoad}(L\ Ll, R)$
$\langle\ L : C\text{-}Physical\text{-}Link \mid \mathsf{Nodes} : \ll No1; No2 \gg,\ \mathsf{Load} : Ld\ \rangle$
$\Rightarrow \langle\ L : C\text{-}Physical\text{-}Link \mid \mathsf{Load} : Ld + R\ \rangle$
$\mathsf{ChPortNode}(No1, R)\ \mathsf{ChPortNode}(No2, R)$
if $(Ll == \mathsf{nil})$.

Finally, the ChPortNode rule does not change from the nonreflective case as described at the end of Section 4.4.

They are applied by means of the Iterate operation until the list of links is empty and the operation finishes with idle. The strategy expression is completely reduced in this way producing a metaterm T' that represents the network configuration with the changed demand. The condition of the first rule is then fulfilled and the network configuration is changed to T'.

6.2. Comparison of the two approaches

The use of reflection simplifies the rules at the object level, mainly due to the metalevel control of failure situations and of the rule application order.

The ConnectionReq rule (2), used to verify the existence of connection between two nodes, replaces the three Com-mod1 rules defined in Section 4.4.

These rules need an extra parameter, namely the set of connections in the network, which is used to go over the network connections until either the connection defined between the given nodes is found, or the set is empty. The first rule treats the existence of a connection between the given nodes, the second rule is used to go over the set of connections passing over the connections not defined between the given nodes, and the third rule is used to send the error message once all connections have been tested and no one was defined between the given nodes.

On the other hand, using reflection one rule is sufficient to treat the existence of a connection. If this rule does not match, the strategy in the metalevel is in charge of sending the failure message to the external object. In this way, a lot of messages are avoided, because there is no need to go over the set of all connections in the network.

The LinkListLoadReq set of rules is another example of rule simplification using reflection. The two rules (4) replace the three rules defined in Section 4.4.

Keeping the control at the object level requires the additional message LinkListLoad Req2 used only internally for implementation purposes. The second and third rules define the behavior of this message. It is used to go over the list of links that support the connection, sending the appropriate messages to its nodes to verify whether they

can support the demanded capacity. At the same time, the rules collect the successful returning messages from the nodes of the previous link in the list. Only in the case that all the nodes of the links in the list can support the demanded capacity, a successful message is sent by the third rule to the connection object.

Using reflection, this extra message is avoided, keeping the specification free from implementation issues. The LinkListLoadReq rule is used to go over the list of links. If a node cannot support the demanded capacity, then the PortNodeReq rule will not match and a failure will be automatically generated by the strategy at the metalevel, removing the need for collecting the nodes returning messages. As in the previous case this avoids the use of a lot of messages.

6.3. Improving control by changing strategies

The strategy language allows us to simplify not only the rules but also the protocol by simultaneously carrying out the verification and the modification processes. It is also possible to differentiate the two failure situations and to send different messages to the external object like in the nonreflective case. Consider the following three rules:

crl [ChDemand2] : ChDemand($O, N, No1, No2, \ll S; D \gg$)
$\langle\ N : Mediator \mid \text{Config} : T\ \rangle$
$\Rightarrow \langle\ N : Mediator \mid \text{Config} : T'\ \rangle$
(To O AckChDemand $No1$ and $No2$ in N)
if rew T' in M-NET with idle := rew T'_C in M-NET with stratChDemand2 .

crl [ChDemand2] : ChDemand($O, N, No1, No2, \ll S; D \gg$)
$\langle\ N : Mediator \mid \text{Config} : T\ \rangle$
$\Rightarrow \langle\ N : Mediator \mid\ \rangle$ (To O NoConnectionBetween $No1$ and $No2$ in N)
if rew T' in M-NET with NoConnection :=
rew T'_C in M-NET with stratChDemand2 .

crl [ChDemand2] : ChDemand($O, N, No1, No2, \ll S; D \gg$)
$\langle\ N : Mediator \mid \text{Config} : T\ \rangle$
$\Rightarrow \langle\ N : Mediator \mid\ \rangle$(To O ServiceCapacityNoSupported)
if rew T' in M-NET with NoPortCapacity :=
rew T'_C in M-NET with stratChDemand2 .

where the metaterm T'_C denotes the configuration $\_[T, \uparrow \text{MCom}(O, N, No1, No2, \ll S;$ D $\gg$)], and stratChDemand2 denotes the strategy

(apply(MCom) ;; Iterate(apply(LinkListLoad);

```
        (apply(PortNode) ;; idle orelse NoPortCapacity);
        (apply(PortNode) ;; idle orelse NoPortCapacity))
    orelse (apply(MComNS) ;; Iterate(apply(LinkListLoad);
        (apply(PortNode) ;; idle orelse NoPortCapacity);
        (apply(PortNode) ;; idle orelse NoPortCapacity))
    orelse NoConnection)) .
```

The main idea is to use the _;;_orelse_ operation, which allows us to verify whether the appropriate rules can be applied at the object level. In case the process succeeds, the strategy ends with idle; otherwise, a special strategy (either NoConnection or NoPortCapacity) is generated, and the reduction process finishes either by the _;;_orelse_ operation or by applying the following equation:

```
eq NoConnection andthen S = NoConnection .
eq (rew T in M with NoPortCapacity) andthen S =
   rew T in M with NoPortCapacity .
```

At the object level the rules MCom and MComNS integrate the rules Com-mod1 and Com-mod2 defined in Section 4.4. They verify the existence of a connection between the given nodes and change the DemandList attribute of the connection object. If the service is already defined, the rule MCom is applied and the service demand is increased; otherwise, the rule MCom cannot match and MComNS is applied, adding the new service to the demand list.

```
crl [MCom] : MCom(O,N,No1,No2,≪ S;D ≫)
  ⟨ C : C-Connection | Nodes: ≪ M1;M2 ≫,
    DemandList: Dl1 ≪ S;D1 ≫ Dl2, LinkList: Ll ⟩
  ⟨ S : C-Service | Capacity: Cp ⟩
⇒ ⟨ C : C-Connection | DemandList: Dl1 ≪ S;D1 + D ≫ Dl2 ⟩
  ⟨ S : C-Service | ⟩LinkListLoad(S,Ll,D * Cp)
if ((M1 == No1) and (M2 == No2)) or
  ((M1 == No2) and (M2 == No1)) .

crl [MComNS] : MCom(O,N,No1,No2,≪ S;D ≫)
  ⟨ C : C-Connection | Nodes: ≪ M1;M2 ≫, DemandList: Dl1,
                       LinkList: Ll ⟩
  ⟨ S : C-Service | Capacity: Cp ⟩
⇒ ⟨ C : C-Connection | DemandList: Dl1 ≪ S;D ≫ ⟩
  ⟨ S : C-Service | ⟩LinkListLoad(S,Ll,D * Cp)
if ((M1 == No1) and (M2 == No2)) or
  ((M1 == No2) and (M2 == No1)) .
```

The LinkListLoad rule modifies the bandwidth of the first link of the list and sends messages to change the number of ports of its two endnodes.

```
crl [LinkListLoad] : LinkListLoad(S, L Ll, B)
   ⟨ L : C-Link | Nodes : ≪ No1; No2 ≫, Load : Ld ⟩
⇒ ⟨ L : C-Link | Load : Ld + B ⟩ LinkListLoad(S, Ll, B)
   PortNode(S, No1, B) PortNode(S, No2, B)
if Ll =/= nil .

crl [LinkListLoad] : LinkListLoad(S, L Ll, B)
   ⟨ L : C-Link | Nodes : ≪ No1; No2 ≫, Load : Ld ⟩
⇒ ⟨ L : C-Link | Load : Ld + B ⟩
   PortNode(S, No1, B)PortNode(S, No2, B)
if Ll == nil .
```

Finally, the rule that verifies if a node supports a given capacity, and, if possible, changes the number of ports on the node is

```
crl [PortNode] : PortNodes(S, No, B)⟨ S : C-Service | Capacity : Cp ⟩
⟨ No : C-Node | Eq : ⟨ Et : C-Eq-Trans | Capacity : Cap ⟩, Used : U ⟩
⇒ ⟨ No : C-Node | Used : U + B DIV Cap ⟩⟨ S : C-Service | ⟩
if (Cp <= Cap) .
```

When a ChDemand message is received by the network mediator, the condition of the rules is evaluated by reducing the strategy expression

```
rew T'C in M-NET with stratChDemand2 .        (5)
```

If the connection between nodes *No1* and *No2* exists in the configuration represented by T'_C, then either the rule MCom or the rule MComNS can be applied at the object level, and then the strategy expression is reduced in both cases, by means of the equations that define the operation _;;_orelse_, to the following expression:

```
rew T''C in M-NET with Iterate(apply(LinkListLoad);
     (apply(PortNode) ;; idle orelse NoPortCapacity);
     (apply(PortNode) ;; idle orelse NoPortCapacity)) .
```

Next, the recursive strategy Iterate is applied and if the nodes can treat the demanded capacity, the reduction process continues. When there are no more links in the connection, the LinkListLoad rule does not match and produces a failure in the Iterate strategy, which finishes the application of the recursive strategy. The reduction process ends with the reduction of the strategy expression to the form rew T' in M-NET with idle using the operation metaApply. The condition of the first rule is then fulfilled and the rule is applied changing the network configuration and sending the successful message to the external object.

If the connection does not exist, then the strategy expression (5) is reduced to the term rew T'' in M-NET with NoConnection which fulfills the condition of the

second rule. The case in which one node does not support a demanded capacity is treated similarly with error message NoPortCapacity.

Notice that this approach reduces even more the number of messages exchanged in the system, because of the integration of the verification and the modification processes.

7. Concluding remarks

Rewriting logic and the Maude language are very appropriate for the specification of object-oriented systems as they integrate the algebraic data types with the object classes, and allow the definition of static and dynamic aspects of a system within one language. They are very appropriate for the specification of object relationships, specially the containment relationship that can be gracefully specified using subobjects contained in other objects and the proposed creation and deletion rules. We show that the subconfiguration concept as introduced in [17] helps in making clear specifications and can be incorporated in future versions of Maude.

Reflection clearly separates the object level behavior from control and management aspects, increasing the application's *modularity*. In this way, the object level is greatly simplified, because the rewriting rules are controlled at the metalevel eliminating the necessity of extra rules at the object level. As we have explained, internal strategies allow us to control at the metalevel

- the order in which the rewriting rules are applied,
- the management of failure situations, either in general or by distinguishing different kinds of failure,
- the successive application of one strategy, by means of the Iterate operation, which gives us a generic pattern to go over lists, thus removing the control from the specific lists,
- the integration of the verification and modification processes,
- the atomicity requirement of some transaction sequences that should be executed in order and without interruption of other executions.

In particular, we have shown how rewriting logic reflection can be used for monitoring functionality that is intrinsically reflective in the object-oriented sense as is the case for the network mediator.

Another advantage is *adaptability*. The same object level can be managed in several ways by changing the strategy controlling it, perhaps through the use of *metastrategies*. This would support the more flexible and runtime adaptable next generation of *active networks* currently being designed.

Of course, the price we are paying for these advantages is the need to go up to the metalevel. However, we have to point out that Maude already provides this access in general (because rewriting logic is reflective) as well as through the internal strategy language. We only need to write down the specific strategies required for our application, which is considerably simpler than complicating the rewriting rules at the object level due to control considerations.

Another important benefit is that the reflective structure facilitates a distributed environment. Different networks can be defined, which will exchange data at the metalevel,

controlling each one its own object level. Then, it is easy to define a metanetwork that controls all the defined networks.

With respect to the design of the Maude language, in our opinion there are some aspects that need more attention. For example, as we have already remarked, sometimes it is necessary to define internal messages in a class that should not be seen from outside the class. This is not possible in the current design of the language, since all messages receive the same treatment. More generally, it is necessary a more detailed study of the subject of *encapsulation* in the context of Maude.

From a similar point of view, the distinction in Maude between class inheritance and module inheritance is not completely satisfactory, due to the impossibility of redefining attributes or messages in subclasses. On the one hand, this forces the creation of additional subclasses, like for example the class *C-ATM-Node* in our application, and on the other hand, it creates inheritance relationships between modules that in principle have no reason to exist. Moreover, from a modeling standpoint, it does not allow the existence in a subclass of method specializations that are consistent with real life inheritance classifications.

As we have pointed out for the cost messages specification, it would be useful to have an abstraction. Then, all subclasses would exhibit a common set of operations, although each one will have a different behavior.

An open subject is the verification of properties stated about the specification. Some work has been done recently on the use of modal and temporal logics on top of rewriting logic for defining the properties [8,10], but there is still a lot of work to do in order to put these logics into practice.

Acknowledgements

We are very grateful to José Meseguer, Manuel G. Clavel, and the anonymous referees for their detailed comments on previous versions of this paper.

This paper integrates updated and revised versions of the research results previously reported in [22,23].

References

[1] P. Borovanský, C. Kirchner, H. Kirchner, Controlling rewriting by rewriting, in: J. Meseguer (Ed.), Proc. Ist Internat. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Vol. 4, WRLA'96, Asilomar, California, September 3–6, Elsevier, Amsterdam, 1996, pp. 168–188. http://www.elsevier.nl/locate/entcs/volume4.html.

[2] M. Clavel, Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. CSLI Lecture Notes, CSLI Publications, Stanford, CA. 2000.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, Metalevel computation in Maude, in: C. Kirchner, H. Kirchner (Eds.), Proc. 2nd Internat. Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, Electronic Notes in Theoretical Computer Science, Vol. 15, Elsevier, Amsterdam, 1998, pp. 3–24. http://www.elsevier.nl/locate/entcs/volume15.html.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J.F. Quesada, Maude: specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. http://maude.csl.sri.com/manual, January 1999.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J.F. Quesada, Towards Maude 2.0. in: K. Futatsugi (Ed.), Proc. 3rd Internat. Workshop on Rewriting Logic and its Applications, WRLA 2000, Electronic Notes in Theoretical Computer Science, Vol. 36, Kanazawa, Japan, September 18–20, Elsevier, Amsterdam, 2000, pp. 297–318. http://www.elsevier.nl/locate/entcs/volume36.html.

[6] M. Clavel, S. Eker, P. Lincoln, J. Meseguer, Principles of Maude, in: J. Meseguer (Ed.), Proc. Ist Internat. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Vol. 4, WRLA'96, Asilomar, California, September 3–6, Elsevier, Amsterdam, 1996, pp. 65–89. http://www.elsevier.nl/locate/entcs/volume4.html.

[7] M. Clavel, J. Meseguer, Axiomatizing reflective logics and languages, in: G. Kiczales (Ed.), Proc. Reflection'96, San Francisco, California, April, 1996, pp. 263–288.

[8] G. Denker, From rewrite theories to temporal logic theories, in: C. Kirchner, H. Kirchner (Ed.), Proc. 2nd Internat. Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, Electronic Notes in Theoretical Computer Science, Vol. 15, Elsevier, Amsterdam, 1998, pp. 273–294. http://www.elsevier.nl/locate/entcs/volume15.html.

[9] G. Denker, J. Meseguer, C.L. Talcott, Formal specification and analysis of active networks and communication protocols: the Maude experience, in: D. Maughan, G. Koob, S. Saydjari (Eds.), Proc. DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25–27, IEEE Computer Society Press, Silver Spring, MD, 2000, pp. 251–265. http://schafercorp-ballston.com/discex/.

[10] J.L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, I. Pita, Towards a verification logic for rewriting logic, in: D. Bert, C. Choppy, P. Mosses (Eds.), Recent Trends in Algebraic Development Techniques, 14th Internat. Workshop, WADT'99, Selected Papers, Lecture Notes in Computer Science, Vol. 1827, Chateau de Bonas, France, September 15–18, 1999, Springer, Berlin, 2000, pp. 438–458.

[11] ISO/IEC DIS 10165-1/ITU-TS X.720, Management Information Model, 1990.

[12] U. Lechner, C. Lengauer, F. Nickl, M. Wirsing (Objects + concurrency) & reusability — a proposal to circumvent the inheritance anomaly, in: P. Cointe (Ed.), ECOOP'96 — Object-Oriented Programming, 10th European Conference, Proceedings, Lecture Notes in Computer Science, Vol. 1098, Linz, Austria, July 8–12, Springer, Berlin, 1996, pp. 232–247.

[13] L. López Zueros, I. Pita, Diseño orientado a objetos aplicado a la gestión integrada de redes, Proc. Telecom I+D Conference, Madrid, 1992, pp. 359–368.

[14] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in: D. Gabbay (Ed.), Handbook of Philosophical Logic, Second Edition, Vol. 9, Kluwer Academic Publishers, Dordrecht, 2002, http://maude.csl.sri.com/papers.

[15] J. Meseguer, Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02R, SRI International, Computer Science Laboratory, February 1990, Revised June 1990. Appendices on functorial semantics have not been published elsewhere.

[16] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoret. Comput. Sci. 96 (1) (1992) 73–155.

[17] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, The MIT Press, Cambridge, MA, 1993, pp. 314–390.

[18] J. Meseguer, Solving the inheritance anomaly in concurrent object-oriented programming, in: O.M. Nierstrasz (Ed.), ECOOP'93 — Object-Oriented Programming, 7th European Conference, Proceedings, Lecture Notes in Computer Science, Vol. 707, Kaiserslautern, Germany, July 26–30, Springer, Berlin, 1993, pp. 220–246.

[19] J. Meseguer, Rewriting logic as a semantic framework for concurrency: a progress report, in: U. Montanari, V. Sassone (Eds.), CONCUR'96: Concurrency Theory, 7th International Conference, Proceedings, Lecture Notes in Computer Science, Vol. 1119, Pisa, Italy, August 26–29, Springer, Berlin, 1996, pp. 331–372.

[20] J. Meseguer, Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems, in: S.F. Smith, C.L. Talcott (Eds.), Proc. IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, Stanford, California, USA, Kluwer Academic Publishers, Dordrecht, 2000, pp. 89–117.

[21] P.C. Ölveczky, Specification and analysis of real-time and hybrid systems in rewriting logic. Ph.D. Thesis, University of Bergen, Norway, 2000. http://maude.csl.sri.com/papers.

[22] I. Pita, N. Martí-Oliet, A Maude specification of an object oriented database model for telecommunication networks, in: J. Meseguer (Ed.), Proc. Ist Internat. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Vol. 4, WRLA'96, Asilomar, California, September 3–6, Elsevier, Amsterdam, 1996, pp. 404–422. http://www.elsevier.nl/locate/entcs/volume4.html.

[23] I. Pita, N. Martí-Oliet, Using reflection to specify transaction sequences in rewriting logic, in: J.L. Fiadeiro (Ed.), Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Selected Papers, Lecture Notes in Computer Science, Vol. 1589, Lisbon, Portugal, April 2–4, 1998, Springer, Berlin, 1999, pp. 261–276.