Using reflection to specify transaction sequences in rewriting logic

Isabel Pita and Narciso Martí-Oliet
Escuela Superior de Informática
Universidad Complutense de Madrid, Spain
{ipandreu,narciso}@eucmos.sim.ucm.es

Abstract

We develop an application of the reflective properties of rewriting logic to the specification of the management process of broadband telecommunications networks. The application is illustrated by a process that modifies the demand of a service between two nodes in a network. The strategy language selected for controlling the process is based on the one presented in [2] which has been enhanced with a new operation that applies a strategy over a list of objects. The specification of the system is developed in the rewriting logic language Maude, which, thanks to its reflective capabilities, can also be used for specifying internally the strategies that control the system. Several modeling approaches are compared, emphasizing the benefits obtained from using reflection to control the rewriting process as opposed to the extra effort required to control the process at the object level itself.

1 Introduction

Rewriting logic was first proposed by Meseguer as a unifying framework for concurrency in 1990 [7, 8]. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [10, 6], and on the development of the Maude language [9, 2], a specification and programming language directly based on rewriting logic. Recently new research has focused on the reflective properties of rewriting logic [3, 4, 5]. Reflection allows a system to access its own metalevel, providing a powerful mechanism for controlling the rewriting process. Some *strategy languages* have already been proposed [2, 1, 5] 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.

In this paper we develop an application of the reflective properties of rewriting logic to the specification of the management process of broadband telecommunications networks. The application 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 [2] which has been enhanced with a new operation that applies a strategy over a list of objects. The management of a network is done through a *mediator*, a metaobject living in the metalevel and having access to the configuration of the network. In this way, we combine ideas coming from the field of logical reflection with ideas coming from the field of object-oriented reflection [9].

The 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 strategy language, and propose a new operation needed for the application. Then we present the selected scenario and define the strategies that control the modification process. Finally, the specification using reflection is compared with the specification presented in [12] without reflection, and some improvements are proposed. This paper is heavily based on our previous paper [12], which should be read for a complete understanding of the network application.

2 Rewriting logic and reflection

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 [8, 9].

A *rewrite theory* $\mathcal{R}$ is defined as a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where (Σ, E) is an equational signature, L is a set of labels, and R is a set of *rewrite rules* of the form $[t]_E \longrightarrow [t']_E$, where 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 [3, 4], that is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations, equations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in the following sense: given any two terms t, t' in $\mathcal{R}$ there are corresponding terms $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ and $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

We will denote the representation (reification) of an object level term *ot* in the metalevel by $\uparrow ot$ (see [3] for the details of the corresponding definition).

Conditional rewriting logic (that is, equations and rules can be conditional [8]) 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; system modules specify the initial model of a rewrite theory $\mathcal{R}$; and object-oriented modules are a special case of system modules, used for the definition of object-oriented classes.

An object is represented as a term `< O : C | a1: v1, ..., an: vn >`, where `O` is the object's name belonging to a set `OId` of object identifiers, `C` is the class identifier, the `ai`'s are the names of the object attributes, and the `vi`'s are their corresponding values.

Rewrite rules are used to implement the method associated to a message received by an object. In general, a rewrite rule in an object-oriented module has the form

$$\begin{array}{ll} \textbf{rl} & M_1 \ldots M_n < O_1 : C_1 \mid atts_1 > \ldots < O_m : C_m \mid atts_m > \\ \longrightarrow & < O_{i1} : C'_{i1} \mid atts'_{i1} > \ldots < O_{ik} : C'_{ik} \mid atts'_{ik} > \\ & < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\ & M'_1 \ldots M'_q \\ \textbf{if} & C \end{array}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is a rule condition. The result of applying a rule of this form is that the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i1}, \ldots, O_{ik}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and 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 lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only on the righthand side of the rule don't matter, and all attributes not explicitly mentioned are left unchanged.

3 The internal strategy language

Strategies are used to control the rewriting process, which in principle could go in many undesired directions. In Maude, thanks to the reflective capabilities of rewriting logic, strategies can be made *internal* to rewriting logic, that is, they can also be defined by means of rewrite rules. In fact there is a great freedom for defining different strategy languages inside Maude (see [2, 5] for two possibilities). 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]. 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 meta-apply : Term Label Nat -> Term .
```

A term `meta-apply(t,l,n)` is evaluated by converting the metaterm `t` to the term it represents, and matching the resulting term against all rules with the given label `l`. 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; otherwise, `error*` is returned.

The strategy language `STRAT` defined in [2] extends the kernel with operations to compose strategies, and also with operations to create and manipulate a solution tree obtained by the application of a strategy. It defines sorts `Strategy` and `StrategyExp` for strategies, and sorts `SolTree` and `SolTreeExp` for the solution tree. In particular, the operations that will be used in our application are:

- operations defining basic strategies:

```
op idle : -> Strategy .
op apply : Label -> Strategy .
op rew_=>_with_ : Term SolTreeExp Strategy -> StrategyExp .
op failure : -> StrategyExp .
```

- operations defining solution trees:

```
op ? : -> SolTreeExp .
op ^ : -> SolTree .
op _{<-_} : SolTree SolTree -> SolTree .
```

- operations that compose strategies:

```
op _;_ : Strategy Strategy -> Strategy .
op _;;_orelse_ : Strategy Strategy Strategy -> Strategy .
op _andthen_ : StrategyExp Strategy -> StrategyExp .
```

These operations satisfy the equations

```
eq rew T => ? with S = rew T => ^{<-T} with S .
eq rew T => S1T with (S ; S') = (rew T => S1T with S) andthen S' .
eq rew T => S1T with idle andthen S = rew T => S1T with S .
eq failure andthen S = failure .
eq rew T => S1T{<-T'} with apply(L) =
   if meta-apply(T',L,z) == error* then failure
   else rew T => S1T{<-meta-apply(T',L,z)} with idle fi .
eq rew T => S1T with (S ;; S'' orelse S') =
   if rew T => S1T with S == failure then rew T => S1T with S'
      else rew T => S1T with S andthen S'' fi .
```

A strategy expression initially has the form `rew T => ? with S`, meaning that we have to apply strategy `S` to the metaterm `T`. This strategy expression is then reduced by means of the equations to an expression of the form `rew T => S1T with S'`, where `S1T` is the solution tree already calculated and `S'` is the remaining strategy. If the reduction process succeeds, a strategy expression of the form `rew T => S1T with idle` is reached,where `S1T` represents the solutions obtained; otherwise, the strategy expression `failure` is generated.

Controlling the order in which 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 that 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 `meta-apply`.

We need a new operation in order to apply a strategy over a list of objects:

```
op Iterate : Strategy -> Strategy .
eq Iterate(S) = S ;; Iterate(S) orelse idle .
```

Intuitively, this composed strategy `Iterate(S)` is very general and simply applies several times a strategy `S`, finishing when this strategy can no longer be applied.

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 5). When all the objects have matched the rule, the `_;;_orelse_` operation will fail to apply and the recursion will end.

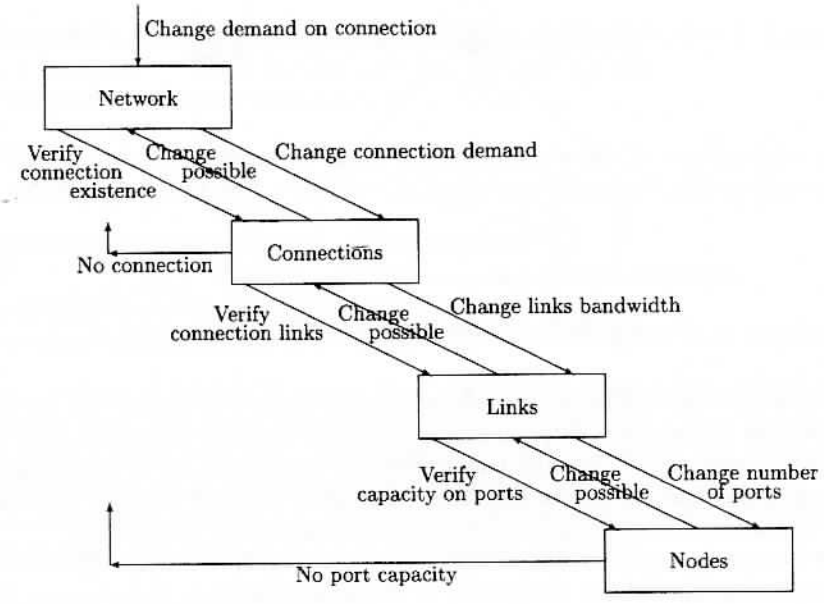

Figure 1: Modification process

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 that Clavel and Meseguer describe in [5], 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 reuse as much as possible the rules as defined in [12].

4 A network application

This strategy language is applied to the specification of a sequence of transactions that change the demand of a given service between two nodes in a telecommunications network. The selected scenario is taken from the one presented in [12], a database model of a broadband telecommunications network. The basic objects of the network are *nodes*, *links* and *connections*. Nodes represent the network points where the communication signals are treated. Links are defined between nodes as entities for which the carried information can be accessed at its two endpoints. Connections are defined as configurable sequences of links. They are used to support the communication services between each pair of nodes.

The protocol followed by the modification process is sketched in Figure 1. The network object `N` receives a message `ChDemand(O,N,NO1,NO2,<<S;ND>>)` from an external object `O` asking to change the demand of the connection between nodes `NO1` and `NO2` by adding a new demand `ND` of the service `S` (expressed as a tuple `<<S;ND>>`) to the existing demand. The network `N` sends messages 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, the modification process starts by changing the service demand on the connection, the bandwidth required on the links that support the

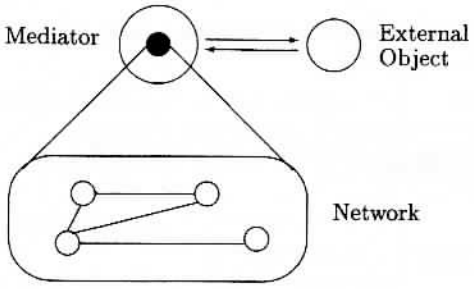

Figure 2: Metalevel mediator for a network

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. Only two error messages are considered. The first one is produced by the absence of connection between the given nodes, and the second one takes place when the port capacity required by the service is not supported by one of the nodes traversed by the connection.

The system is specified using the object-oriented facilities of the Maude language. The process can be defined completely at the object level, as it is done in [12], or it can be controlled from the metalevel using the reflective properties of the logic. In the following we will illustrate the benefits obtained from using this second approach.

5 Strategies and rules for the modification process

All the different objects that constitute a network (nodes, links, connections, etc.) as modelled in [12], form a configuration in the sense described in [9], that is, 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*

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

A *mediator* has as the value of its attribute `Config` a metaterm of the form `↑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, that is, the messages addressed to the network from an external object are processed by the mediator, that also will send back the corresponding answer. This situation is depicted in Figure 2, and constitutes a novel application of ideas from logical reflection to the description of object-oriented group reflection already sketched by Meseguer in [9].

The first approach to control the modification process through the mediator follows the protocol described in the previous section. We define two conditional rules. The first one is applied when the modification is possible, 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. Depending on the resulting strategy expression, the configuration is then changed or an error message is sent to the external object.

```
rl [ChDemand1]: ChDemand(O,N,NO1,NO2,<<S;ND>>)< N : Mediator | Config: T >
=> < N : Mediator | Config: T' > (To O AckChDemand NO1 and NO2 in N)
if rew T* => ? with stratChDemand1 == rew T* => ^{<-T'} with idle .
```

```
rl [ChDemand1]: ChDemand(O,N,NO1,NO2,<<S;ND>>)< N : Mediator | Config: T >
=> < N : Mediator > (To O NoChDemand NO1 and NO2 in N)
if rew T* => ? with stratChDemand1 == failure .
```

Where the metaterm T represents a correct state of the network; T* denotes the configuration [T,↑ComMod(O,N,NO1,NO2,<<S;ND>>)]; and stratChDemand1 denotes the strategy

```
apply(ComMod);Iterate(apply(LinkListLoadReq);
                      apply(PortNodeReq);apply(PorNodeReq));
apply(LinksVerified);apply(ComMod2);
Iterate(apply(ChLinkListLoad);apply(ChPortNodes);apply(ChPortNodes)) .
```

The terms ComMod, LinkListLoadReq, and PortNodeReq are labels of rules defined at the object level, which are described below for the reflective case. The definition of these rules and the rules with labels LinksVerified, ComMod2, ChLinkListLoad, ChPortNodes for the nonreflective case can be found in [12].

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

```
rew T* => ? with stratChDemand1 .                                    (1)
```

using the equations of the strategy language. First it tries to apply the ComMod rule at the object level, which verifies the existence of connection between nodes NO1 and NO2 in network N.

```
rl [ComMod]: ComMod(O,N,NO1,NO2,<<S;ND>>)
   < C : C-Connection | Nodes: <<M1;M2>>, LinkList: LL >              (2)
=> < C : C-Connection > LinkListLoadReq(O,N,C,<<S;ND>>,LL)
if ((M1 == NO1) and (M2 == NO2)) or ((M2 == NO1) and (M1 == NO2)) .
```

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

```
rew T* => ^{<-meta-apply(T*,ComMod,0)} with
 Iterate(apply(LinkListLoadReq);apply(PortNodeReq);apply(PortNodeReq));
 apply(LinksVerified);apply(ComMod2);                                (3)
 Iterate(apply(ChLinkListLoad);apply(ChPortNodes);apply(ChPortNodes)) .
```

Otherwise, rew T* => ^{<-T*} with apply(ComMod) reduces to failure. 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 reduction of the strategy expression (3) continues by applying the recursive equation Iterate defined in Section 3.

```
rew T* => ^{<-[T,↑LinkListLoadReq(O,N,C,<<S;ND>>,LL)]} with
 Iterate(apply(LinkListLoadReq);apply(PortNodeReq);apply(PorNodeReq));
 apply(LinksVerified);apply(ComMod2);
 Iterate(apply(ChLinkListLoad);apply(ChPortNodes);apply(ChPortNodes)) .
```

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, a successful message is sent to the connection object.

```
rl [LinkListLoadReq]: LinkListLoadReq(O,N,C,<<S;ND>>,L LL)
   < L : C-Link | Nodes: <<NO1;NO2>> >
=> < L : C-Link> PortNodeReq(S,NO1)                                    (4)
   PortNodeReq(S,NO2) LinkListLoadReq(O,N,C,<<S;ND>>,LL) .

rl [LinkListLoadReq]: LinkListLoadReq(O,N,C,<<S;ND>>,nil)
=> (To C and O and N and <<S;ND>> LinksVerified) .

rl [PortNodeReq] PortNodeReq(S,NO)
   < NO : C-Node | Eq: < ET : C-Eq | Capacity: CAP > >
   < S : C-Service | Capacity: CP >
=> < NO : C-Node > < S : C-Service > if (CP <= CAP) .
```

The application of the recursive equation `Iterate` continues until either all the links on the connection have been treated or some node produces an error. In the first case, the modification part of the process is accomplished using the remaining strategy

```
apply(LinksVerified);apply(ComMod2);
Iterate(apply(ChLinkListLoad);apply(ChPortNodes);apply(ChPortNodes))
```

which cannot fail because the verification part guarantees that the modification is possible. The strategy expression is completely reduced in this way producing a metaterm `T'` that represents the network configuration with the changed demand on the solution tree. The condition of the first rule is then fulfilled and the network configuration is changed to `T'`. A node error when verifying the links reduces the strategy expression to `failure` as in the no connection case.

6 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 `ComMod` rule used to verify the existence of connection between two nodes (2), replaces the following three rules defined in [12]:

```
rl [ComMod]: ComMod(O,NO1,NO2,C CS,<<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)) .

rl [ComMod]: ComMod(O,NO1,NO2,C CS,<<S;D>>)
   < C : C-Connection | Nodes: <<M1;M2>>, LinkList: LL >
=> < C : C-Connection > ComMod(O,N,NO1,NO2,CS,<<S;D>>)
if ((M1 =/= NO1) or (M2 =/= NO2)) and ((M2 =/= NO1) or (M1 =/= NO2)) .

rl [ComMod]: ComMod(O,N,NO1,NO2,null,<<S;D>>)
=> (To N and O NoConnectionBetweenNodes NO1 and NO2) .
```

These rules need an extra parameter, the set of connections in the network, which is used to go over the network connections until either the one defined between the given nodes is found, or the set is empty. The first rule treats the existence of 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 is defined between the given nodes.

On the other hand, using reflection only a rule to treat the existence of connection is needed. 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 given in (4) replace the following three rules defined in [12]:

```
rl [LinkListLoadReq]: LinkListLoadReq(O,N,C,<<S;ND>>,L LL)
   < L : C-Link | Nodes: <<NO1;NO2>> >
=> < L : C-Link > PortNodeReq(O,N,C,L,<<S,ND>>,NO1)
   PortNodeReq(O,N,C,L,<<S;ND>>,NO2)
   LinkListLoadReq2(O,N,C,<<S;ND>>,LL,L,NO1,NO2) .

rl LinkListLoadReq2(O,N,C,<<S;ND>>,L LL,L1,NO1,NO2)
   (To L1 and O and N and C and <<S;ND>> PortInNode NO1)
   (To L1 and O and N and C and <<S;ND>> PortInNode NO2)
   < L : C-Link | Nodes: <<M1;M2>> >
=> < L : C-Link > PortNodeReq(O,N,C,L,<<S,ND>>,M1)
   PortNodeReq(O,N,C,L,<<S;ND>>,M2)
   LinkListLoadReq2(O,N,C,<<S;ND>>,LL,L,M1,M2) .

rl LinkListLoadReq2(O,N,C,<<S;ND>>,nil,L,NO1,NO2)
   (To L and O and N and C and <<S;ND>> PortInNode NO1)
   (To L and O and N and C and <<S;ND>> PortInNode NO2)
=> (To C and O and N and <<S;ND>> LinksVerified) .
```

Keeping the control at the object level requires an additional message `LinkListLoadReq2` used only internally for implementation purposes. The second and third rules define the behaviour 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.

7 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 send different messages to the external object. Consider the following three rules:

```
rl [ChDemand2]: ChDemand(O,N,NO1,NO2,<<S;ND>>)< N : Mediator | Config: T >
=> < N : Mediator | Config: T' >(To O AckChDemand NO1 and NO2 in N)
if rew T@ => ? with stratChDemand2 == rew T@ => ^{<-T'} with idle .

rl [ChDemand2]: ChDemand(O,N,NO1,NO2,<<S;ND>>)< N : Mediator | Config: T >
=> < N : Mediator > (To O NoConnectionBetween NO1 and NO2 in N)
if rew T@ => ? with stratChDemand2 == rew T@ => SlT with NoConnection .

rl [ChDemand2]: ChDemand(O,N,NO1,NO2,<<S;ND>>)< N : Mediator | Config: T >
=> < N : Mediator > (To O ServiceCapacityNoSupported)
if rew T@ => ? with stratChDemand2 == rew T@ => SlT with NoPortCapacity .
```

Where the metaterm `T@` denotes the configuration `[T,↑MCom(O,N,NO1,NO2,<<S;ND>>)]`, and `stratChDemand2` 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 applying one of the following equations:

```
eq NoConnection;S = NoConnection .
eq NoPortCapacity;S = NoPortCapacity .
```

At the object level the rules `MCom` and `MComNS` integrate the `ComMod` and `ComMod2` rules of the first approach. They verify the existence of 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.

```
rl [MCom] MCom(O,N,NO1,NO2,<<S;ND>>)
   < C : C-Connection | Nodes: <<M1;M2>>, DemandList: DL1 <<S;NC>> DL2,
                        LinkList: LL >
   < S : C-Service | Capacity: CP >
=> < C : C-Connection | DemandList: DL1 <<S;NC+ND>> DL2 >
   < S : C-Service> LinkListLoad(S,LL,ND*CP)
if ((M1 == NO1) and (M2 == NO2)) or ((M1 == NO2) and (M2 == NO1)) .
```

```
rl [MComNS] MCom(O,N,NO1,NO2,<<S;ND>>)
   < C : C-Connection | Nodes: <<M1;M2>>, DemandList: DL1, LinkList: LL >
   < S : C-Service | Capacity: CP >
=> < C : C-Connection | DemandList: DL1 <<S;ND>> > <S : C-Service >
   LinkListLoad(S,LL,ND*CP)
if ((M1 == NO1) and (M2 == NO2)) or ((M1 == NO2) and (M2 == NO1)) .
```

The first `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. If the list of links is empty it does nothing. Finally, the rule `PortNodes` verifies if a node supports a given capacity, and, if possible, changes the number of ports on the node.

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

```
rew T@ => ? with stratChDemand2 .                                    (5)
```

If the connection between nodes `NO1` and `NO2` exists in the configuration represented by `T@`, then the rule `MCom` or `MComNS` can be applied at the object level, and the strategy expression is reduced in both cases to

```
rew T@ => ^{<-T@'} with Iterate(apply(LinkListLoad);
             (apply(PortNode);;idle orelse NoPortCapacity);
             (apply(PortNode);;idle orelse NoPortCapacity))
```

using the equations that define the operation _;;_`orelse`_.

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@ => {<-T'} with idle` using the operation `meta-apply`. 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@ => S1T 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.

8 Concluding remarks

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

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.

Acknowledgements. We are very grateful to José Meseguer and Manuel G. Clavel for their detailed comments on a previous version of this paper, as well as their careful explanations of their papers on reflection and internal strategies.

References

[1] P. Borovanský, C. Kirchner, and H. Kirchner, Controlling rewriting by rewriting, in [11].

[2] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, Principles of Maude, in [11].

[3] M.Clavel and J. Meseguer, Axiomatizing reflective logics and languages, in: G. Kiczales, ed., *Proc. Reflection'96, San Francisco, CA*, April 1996, 263–288.

[4] M. Clavel and J. Meseguer, Reflection and strategies in rewriting logic, in [11].

[5] M. Clavel and J. Meseguer, *Internal strategies in a reflective logic*, manuscript, Computer Science Laboratory, SRI International, April 1997.

[6] N. Martí-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, Technical report SRI-CSL-93-05, SRI International, August 1993. To appear in D. M. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[7] J. Meseguer, Rewriting as a unified model of concurrency, in: J.C.M.Baeten y J.W.Klop, eds., *Proc. CONCUR'90*, LNCS 458, Springer-Verlag, 1990, 384–400.

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

[9] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993, 314–390.

[10] J. Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: U. Montanari and V. Sassone, eds., *Proc. CONCUR'96*, LNCS 1119, Springer-Verlag, 1996, 331–372.

[11] J. Meseguer, editor, *Proc. First Int. Workshop on Rewriting Logic and its Applications, Asilomar, CA*, Electronic Notes in Theoretical Computer Science 4, Elsevier, Sept. 1996. URL `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`

[12] I. Pita and N. Martí-Oliet, A Maude specification of an object-oriented database model for telecommunication networks, in [11].