Available online at www.sciencedirect.com

ScienceDirect

Computer Standards & Interfaces 29 (2007) 481 – 498

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Writing and executing ODP computational viewpoint specifications using Maude

José Raúl Romero, Antonio Vallecillo *, Francisco Durán

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

Received 2 February 2006; received in revised form 27 October 2006; accepted 11 November 2006
Available online 19 January 2007

Abstract

The Reference Model of Open Distributed Processing (RM-ODP) is a joint standardization effort by ITU-T and ISO/IEC for the specification of large open distributed systems. RM-ODP is becoming increasingly relevant now because the size and complexity of large distributed systems is challenging current software engineering methods and tools, and because international standards have become key to achieve the required interoperability between the different parties and organizations involved in the design and development of complex systems. RM-ODP defines five viewpoints for decomposing the design activity into separate areas of concern. One of the RM-ODP viewpoints, the computational viewpoint, focuses on the basic functionality of the system and its environment, independently of its distribution. Although several notations have been proposed to model the ODP computational viewpoint, either they are not expressive enough to faithfully represent all its concepts, or they tend to suffer from a lack of formal support. In this paper we introduce the use of Maude as a formal notation for writing and executing ODP computational viewpoint specifications. Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. We show how Maude offers a simple, natural, and accurate way of modeling the ODP computational viewpoint concepts, allows the execution of the specifications produced, and offers good tool support for reasoning about them.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Rewriting logic; Maude; RM-ODP; ODP computational viewpoint; Environment contracts

1. Introduction

Viewpoint modeling is becoming an effective approach for dealing with the inherent complexity of large distributed systems. Current software architectural practices, as described in IEEE Std. 1471 [18], divide the design activity into several areas of concerns, each one focusing on a specific aspect of the system. Examples include the "4+1" view model [22], the Zachman's framework [40], or the Reference Model of Open Distributed Processing (RM-ODP) [19].

In particular, the rapid growth of distributed processing has led to the adaptation of the RM-ODP framework, which is a joint standardization effort by ISO/IEC and ITU-T that creates an architecture within which support of distribution, interworking and portability can be integrated. Several years after its final adoption as ITU-T Recommendation and ISO/IEC International Standard, the Reference Model of Open Distributed Processing is increasingly relevant, mainly because the size and complexity of current IT systems is challenging most of the current software engineering methods and tools. These methods and tools were not conceived for use with large, open and distributed systems, which are precisely the systems that the RM-ODP addresses. In addition, the use of international standards has become the most effective way to achieve the required interoperability between the different parties and organizations involved in the design and development of complex systems. As a result, we are now witnessing many major companies and organizations investigating RM-ODP as a promising alternative for specifying their IT systems, and for structuring their large-scale distributed software designs.

The RM-ODP framework provides five generic and complementary viewpoints on the system and its environment: *enterprise*, *information*, *computational*, *engineering* and *technology*

* Corresponding author.
E-mail addresses: jrromero@lcc.uma.es (J.R. Romero), av@lcc.uma.es (A. Vallecillo), duran@lcc.uma.es (F. Durán).

0920-5489/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2006.11.004

viewpoints. They allow different stakeholders to observe a system from different perspectives [23].

The RM-ODP is a general framework, and therefore it was conceived as both notation and methodology independent. However, the fact that the Reference Model does not prescribe any specific notation for representing its concepts and viewpoint languages really hinders the development of commercial tools for writing and analyzing ODP system specifications. Several notations have been proposed for the different viewpoints by different authors, which seem to agree on the need to represent the semantics of the ODP viewpoints concepts in a precise manner [3,7,19,23,37]. For example, formal description techniques such as Z and Object-Z have been proposed for the information and enterprise viewpoints [39], and LOTOS, SDL or Z for the computational viewpoint [19,38].

This paper is the third of a trilogy that explores the use of rewriting logic and Maude [9] for specifying ODP viewpoints. Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. We have already used Maude for successfully modeling the ODP enterprise viewpoint [12] and the information viewpoint [11]. Here we shall show how rewriting logic, and in particular Maude, provide the expressiveness required for modeling ODP computational viewpoint specifications.

The use of Maude provides additional advantages. The fact that rewriting logic specifications are executable allows us to apply a flexible range of increasingly stronger formal analysis methods and tools, such as run-time verification [17], model checking [13], or theorem proving [10]. Maude offers a comprehensive toolkit for automating such kinds of formal analysis of specifications. In addition, Maude offers support for the specification of real-time properties of systems, which are required to capture some quality of service (QoS) and other temporal constraints, that form part of the environment contracts of a computational specification. These kinds of contracts are essential in many service-based architectures (such as SOA) for modeling service level agreements and other QoS characteristics of distributed systems.

The structure of this document is as follows. First, Sections 2–4 serve as a brief introduction to the RM-ODP, the ODP computational viewpoint and Maude, respectively. Then, Section 5 presents our proposal, describing how to write computational specifications in Maude. Section 6 is dedicated to a case study that illustrates our approach. Section 7 compares our work with other related proposals and, finally, Section 8 draws some conclusions and outlines some future research activities.

2. Introduction to the RM-ODP

Distributed systems can be very large and complex, and the many different considerations which influence their design can result in a substantial body of specification, which needs a structuring framework if it is to be managed successfully. The purpose of the RM-ODP is to define such a framework.

RM-ODP is based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture. The framework for system specification provided by the RM-ODP has four fundamental elements:

(1) an object modelling approach to system specification;
(2) the specification of a system in terms of separate but interrelated viewpoint specifications;
(3) the definition of a system infrastructure providing distribution transparencies for system applications;
(4) a framework for assessing system conformance.

Most complex system specifications are so extensive that no single individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the systems specifications. A business executive will ask different questions of a system make-up than would a system implementor. The concept of RM-ODP viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system.

The RM-ODP framework provides five generic and complementary viewpoints on the system and its environment:

- The enterprise viewpoint, which focuses on the purpose, scope and policies for the system and its environment. It describes the business requirements and how to meet them.
- The information viewpoint, which describes the information managed by the system and the structure and content type of the supporting data.
- The computational viewpoint, which describes the functionality provided by the system and its functional decomposition in terms of objects which interact at interfaces.
- The engineering viewpoint, which focuses on the mechanisms and functions required to support distributed interactions between objects in the system.
- The technology viewpoint, which focuses on the choice of technology of the system. It describes the technologies chosen to provide the processing, functionality and presentation of information.

Each of these viewpoints satisfies an audience with interest in a particular set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Although separately specified, the viewpoints are not completely independent; key items in each are identified as related to items in the other viewpoints. However, the viewpoints are sufficiently independent to simplify reasoning about the complete specification. The mutual consistency among the viewpoints is ensured by the architecture defined by RM-ODP, and the use of a common object model provides the glue that binds them all together.

3. The computational viewpoint

The computational viewpoint describes the functionality of the ODP system and its environment, in terms of objects performing individual functions and interacting at well-defined interfaces. In the computational viewpoint, applications and ODP functions consist of configurations of interacting computational objects. This specification is independent from how these objects will later be grouped in clusters, distributed or replicated, and also independent from the final technology used to implement them.

The heart of the computational language is the object model, which defines: the form that object interfaces can have; the way in which interactions can take place at them; the actions an object can perform, in particular the creation of new objects and interfaces; and the establishment of bindings.

The computational object model provides the basis for ensuring consistency between engineering and technology specifications (including programming languages and communication mechanisms) thus allowing open interworking and portability of components in the resulting implementation. The computational language also enables the specifier to express constraints on the distribution of an application (in terms of environment contracts associated with individual interfaces and interface bindings of computational objects).

3.1. Computational language concepts

In the ODP Reference Model, the computational language uses a basic set of concepts and structuring rules, including those from ITU-T Recommendation X.902, ISO/IEC 10746-2, and several concepts specific to the computational viewpoint. An initial proposal for modeling some of these concepts was presented in [35]. In this paper we have extended that modeling approach, complementing it with the use of time, several kinds of Quality of Service (QoS) constraints for modeling environment contracts, and enhanced tool support (e.g., timed model checking). These issues are essential to any multimedia or real-time industrial application, and are covered here.

3.1.1. Objects and interfaces

ODP systems are modeled in terms of *objects*. An object contains information and offers services. A system is modeled as a configuration of interacting objects. In the computational viewpoint we refer to objects as *computational objects*, which model the entities defined in a computational specification. Computational objects are abstractions of entities that occur in the real world, in the ODP system, or in other viewpoints [19].

Computational objects have *state* and can interact with their environment at *interfaces*. An interface is an abstraction of the behavior of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur.

Objects are units of abstraction: object interfaces and interactions provide abstract views of the state of an object, hiding details of its implementation. Objects are units of encapsulation: the state of an object can only be accessed and modified by the environment through interactions. In ODP, an object may have multiple interfaces.

Binding objects are computational objects which provide a binding between a set of other computational objects. They help composing (synchronizing) two or more interfaces, e.g., a binding object may be responsible for ensuring that a certain level of quality of service is maintained between interacting objects.

3.1.2. Computational templates

Computational objects and interfaces can be specified by templates. In ODP, an <X> *template* is the specification of the common features of a collection of <X>s in sufficient detail that an <X> can be instantiated using it. Thus, an interface of a computational object is usually specified by a *computational interface template*, which is an interface template for either a signal interface, a stream interface, or an operation interface. A computational interface template comprises a signal, stream or operation interface signature, as appropriate; a behavior specification; and an environment contract specification.

An *interface signature* consists of a name, a causality role (producer, consumer, etc.) [34], and a set of signal signatures, operation signatures, or flow signatures as appropriate. Each of these signatures specify the name of the interaction and its parameters (names and types).

3.1.3. Interactions

RM-ODP prescribes three particular types of interactions: *signals*, *operations*, and *flows*. A signal may be regarded as a single, atomic action between computational objects. Signals constitute the most basic unit of interaction in the computational viewpoint. Operations are used to model object interactions as represented by most message passing object models, and come in two flavors: *interrogations* and *announcements*. An interrogation is a two-way interaction between two objects: the client object invokes the operation (invocation) on one of the server object interfaces; after processing the request, the server object returns some result to the client object, in the form of a termination. An announcement is a one-way interaction between a client object and a server object. In contrast to an interrogation, after invocation of an announcement operation on one of its interfaces, the server object does not return a termination. Terminations model every possible outcome of an operation.

Operations can be defined in terms of signals. Every invocation can then be defined by two signals, one outgoing from the client (the invocation submit), and the corresponding signal that reaches the server (the invocation deliver). Likewise, terminations can be modeled by two other signals, the one that is sent by the server (the termination submit), and the one that finally reaches the client (the termination deliver).

Flows model streams of information, i.e., a flow represents an abstraction of a sequence of interactions from a producer to a consumer, whose exact semantics depends on the specific application domain. In the ODP computational viewpoint, flows can also be expressed in terms of signals [19].

3.1.4. Environment contracts

Computational object templates may have environment contracts associated with them. These environment contracts may be regarded as agreements on behaviors between the object and its environment, including QoS constraints, usage and management constraints, etc. QoS constraints involve temporal, volume and dependability constraints, amongst others, and they can imply other usage and management constraints, such as location and distribution transparency constraints.

3.2. Structure of ODP computational specifications

A computational specification describes the functional decomposition of an ODP system, in distribution transparent terms, as: (a) a configuration of computational objects (including binding objects); (b) the internal actions of those objects; (c) the interactions that occur among those objects; and (d) environment contracts for those objects and their interfaces.

A computational specification is constrained by the rules of the computational language. These comprise: (a) interaction rules, binding rules, and type rules, that provide distribution transparent interworking; (b) template rules, that apply to all computational objects; and (c) failure rules, that apply to all computational objects and identify the potential points of failure in computational activities.

A computational specification also defines an initial set of computational objects and their behavior. The configuration will change as the computational objects instantiate further computational objects or computational interfaces; perform binding actions; effect control functions upon binding objects; delete computational interfaces; or delete computational objects.

4. Rewriting logic and Maude

Maude [8,9] is a high-level language and a high-performance interpreter and compiler of the OBJ [15] algebraic specification family that supports membership equational logic and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute more than 3 million rewriting steps per second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, or even programming languages. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to its manual [9] for more details.

Rewriting logic [25] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where ($\Sigma$, E) is an equational theory describing its set of states as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E), and R is a collection of rewrite rules. Maude's underlying equational logic is membership equational logic [26], a Horn logic whose atomic sentences are equalities $t=t'$ and membership assertions of the form $t{:}S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort overloading of operators, and definition of partial functions with equationally defined domains.

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

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

If a functional specification is terminating, confluent, and sort-decreasing, then it can be executed [6]. Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is found. Some equations, like those expressing the commutativity of binary operators, are not terminating but nonetheless they are supported by means of operator attributes, so that Maude performs simplification modulo the equational theories provided by such attributes, which can be associative (`assoc`), commutativity (`comm`), identity (`id`), and idempotence (`idem`). The above properties must therefore be understood in the more general context of simplification modulo such equational theories.

While functional modules specify membership equational theories, rewrite theories are specified by system modules. A system module may have the same declarations of a functional module plus rules of the form $t \rightarrow t'$, where t and t' are Σ-terms, which specify the dynamics of a system in rewriting logic. These rules describe the local, concurrent transitions possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

4.1. Object-oriented specifications: Full Maude

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

In a concurrent object-oriented system, the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The predefined sort `Configuration` represents configurations of Maude objects and messages, with `none` as empty configuration and the empty syntax operator _ _ as union of configurations.

```
sort Configuration .
subsorts Object < Configuration .
subsorts Message < Configuration .
op none : -> Configuration [ ctor] .
op _ _ : Configuration Configuration
    -> Configuration
    [ctor assoc comm id: none] .
```

Thus, rewrite rules define transitions between configurations, and their general form is:

```
crl [r]:
  < O1 : C1 | atts1 > ... < On : Cn | attsn >
  M1 ... Mm
  => < Oi1 : C'i1 | atts'i1 > ... < Oik : C'ik | atts'ik >
     < Q1 : C''1 | atts''1 >...< Qp : C''p | atts''p >
     M'1 ... M'q
  if Cond .
```

where r is the rule label, $M_1 \ldots M_m$ and $M'_1 \ldots M'_q$ are messages, $O_1 \ldots O_n$ and $Q_1 \ldots Q_p$ are object identifiers, $C_1 \ldots C_n$, $C'_{i_1} \ldots C'_{i_k}$ and $C''_1 \ldots C''_1$ are classes, $i_1 \ldots i_k$ is a subset of $1 \ldots n$, and Cond is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1 \ldots M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1} \ldots O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1 \ldots Q_p$ are created; and (e) new messages $M'_1 \ldots M'_q$ are created, i.e., they are sent. Rule labels and guards are optional.

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

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

When several objects or messages appear in the left-hand side of a rule, they need to synchronize in order for such a rule to be fired. These rules are called synchronous, while rules involving just one object and one message in their left-hand sides are called asynchronous rules.

Maude distinguishes two kinds of inheritance, namely class inheritance and module inheritance. Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration $C<C'$, indicating that C is a subclass of C′, is a particular case of a subsort declaration $C<C'$, by which all attributes, messages, and rules of the superclasses, as well as the newly defined attributes, messages and rules of the subclass characterize its structure and behavior. ODP's notion of subtyping—*A* is a subtype of *B* if every <X> that satisfies *A* also satisfies *B*—corresponds to Maude's class inheritance. On the other hand, the ODP's notion of inheritance, that allows the suppression and modification of the attributes and methods of the base class [19, Part 2-9.21] corresponds to Maude's module inheritance. Throughout the paper, by inheritance we shall mean Maude's notion of class inheritance, i.e., ODP's subtyping. Multiple inheritance is also supported in Maude [8].

4.2. Timed specifications: Real-Time Maude

In Maude, the formal specification of distributed object-oriented real-time systems is performed with Real-Time Maude [31,32]. This is an extension of Maude that offers a good treatment of time and a wide range of flexible techniques, including timed rewriting for simulation purposes, search and time-bounded linear temporal logic model checking.

The specification formalism is based on real-time rewrite theories, which extend rewriting logic. Real-Time Maude specifications are also executable. This helps debugging and simulating system's progress with time. However, an execution runs only one of the possible concurrent behaviors of the system. Real-Time Maude provides additional useful tools to gain further assurance about a system specification, such as model-checking, which allow to explore different temporal behaviors of the system from a given initial state. A time-bounded linear temporal logic model checker also permits the analysis of all possible behaviors considering certain time and machine performance constraints.

To specify real-time rewrite theories, Real-Time Maude extends the different types of modules available in Maude (functional modules, system modules, and object-oriented modules) to their corresponding timed versions. In particular, Real-time Maude extends the previous notion of object-oriented module to timed object-oriented module, whose syntax is `tomod` ... `endtom`. The sort that specifies time domain is `Time`, which can be considered a commutative monoid `(Time, 0, +, <)`. Some predefined modules specifying useful time domains, namely natural numbers and nonnegative rational numbers, are provided too. These modules define a supersort `TimeInf` that extends the sort `Time` with an infinity value `INF`.

A real-time rewrite theory may contain two possible types of rules:

- *Instantaneous rewrite rules*, which are considered as ordinary Maude rules and assumed to take zero time. (Their syntax is explained in Section 4.1.)
- *Tick rewrite rules*, which actually model elapse of time in a system. These rules should normally have either of the following admissible forms:

  `crl [l] : {t}=>{t'} in time R if` (*R* lt *u*) ∧ *cond* .
  `crl [l] : {t}=>{t'} in time R if` (*R* le *u*) ∧ *cond* .
  `crl [l] : {t}=>{t'} in time R if` *cond* .

  where `R` is a new timed variable that does not occur in *t* and which is not initialized in *cond*; and `{t}` represents the global state of the system in a given time:

```
op {_} : System -> GlobalSystem
     [ ctor] .
```

A rule `{t}=>{t'}` in time `R` models a transition from the global state `{t}` to the global state `{t'}` taking time `R`. The constrains `R lt` *u* and `R le` *u* guarantee, respectively, that the time taken by the transition is less than `(lt)` or less or equal than `(le)` *u*.

In Real-Time Maude, the state of a system is represented by terms of sort `System` so that tick rules ensure that time advances uniformly in all parts of the system.

The user may specify the data type of time values as considered most appropriated. Therefore, time can be either discrete, which is recommended to specify real-time systems, or dense, which is often used to model hybrid systems.

One common way of specifying in Real-Time Maude how the system evolves with time is in terms of operations `delta` and `mte` [31]:

```
op delta : Configuration Time
      -> Configuration .
op mte : Configuration -> TimeInf .
```

Sort `Configuration` is declared to be a subsort of `System`. Thus, the `delta` operation determines, for each object and message in the given configuration, its evolution after a period of time has passed. The second operation, `mte` (maximum time elapse), determines the maximum amount of time within which no timed action occurs in the system.

Given `delta` and `mte` operations, a unique tick rule is enough to manage time:

```
var C : Configuration .
var R : Time .

crl[tick] :
  { C} => { delta(C, R)}
  in time R if R <= mte(C)
 [nonexec] .
```

The timed behavior of the objects in the system will be specified in terms of these two operations. To simplify their definition, the following equations are provided. These definitions need to be completed for each particular system, by providing the appropriate additional equations (see Section 6.3 for illustrative examples).

```
vars C1 C2 : NEConfiguration .
var R : Time .
var O : Object .
var Msg : Message .

eq delta(none, R) = none .
eq delta(C1 C2, R)
  = delta(C1, R) delta(C2, R) .
eq mte(none) = INF .
eq mte(C1 C2)
  = min(mte(C1), mte(C2)) .
eq delta(O, R) = O [ owise] .
eq delta(Msg, R) = Msg .
eq mte(O) = INF [ owise] .
eq mte(Msg) = INF .
```

The sort `NEConfiguration` is a subsort of `Configuration` taking non-empty configurations. The last four equations define the `delta` and `mte` operations for messages and for those objects not affected by time. Thus, according to these equations, such objects and messages will not be changed by the `delta` operation, and their maximum time elapse will be infinite. Notice the use of the otherwise attribute (`owise`); the otherwise equations will be used only if no other equation can be used. As part of his specification, the user will give equations for the `delta` and `mte` operators on those objects affected by the passage of time.

5. Writing computational specifications in Maude

We present our approach for modeling objects and interfaces, and their behavior in the following two sections. Then, Section 5.3 introduces our Computational Infrastructure, which provides basic functionality giving support to some basic mechanisms and structuring rules for the computational viewpoint. Section 5.4 gives a discussion on the modeling of contracts.

5.1. Modeling objects and interfaces

Our approach to modeling ODP computational specifications is based on the use of computational object templates for specifying system objects, and computational interface templates for specifying interfaces. Both computational objects and interfaces will be instantiated from those templates. Interactions will be specified in terms of signals.

Computational object templates will be represented by Maude classes. In Maude, each class is defined by a name and a set of attributes (of certain sort) that describes the state of the instances of the class.

Computational objects will then be represented by Maude objects. In Maude, each object belongs to one class (although it may change during the object's lifetime) that in our model corresponds to the computational object template specifying the object. All computational object templates will inherit from class `CV-Object`, which describes the common features that any computational object should exhibit:

```
class CV-Object |
  conf : Configuration .
```

The attribute `conf` will store the configuration of Maude objects representing the interfaces of the computational object. (This attribute is of sort `Configuration`, instead of being a set of objects, because it may also contain the Maude messages that represent the object's pending internal actions, which do not happen at the object interfaces — see below.)

Maude configurations allow to capture not only the individual state of each object, but also the collective state of the system. Thus, at a more global level, Maude configurations of objects of class `CV-Object` will naturally represent the ODP configurations of computational objects.

In addition, a `CV-Object` can be extended to deal with real-time information. For example, the following class `CV-RTObject` defines an attribute `timer` (of sort `Time`) to store an internal timer that can be used by the object for incremental and countdown timed calculations (see Section 6.3 for examples)—if several timers where needed we could use a set of named timers, a map, or any other structure according to our needs.

```
class CV-RTObject | timer : Time .
subclass CV-RTObject < CV-Object .
```

Binding objects, as computational objects, will be represented by Maude objects. They will adhere to the particular provisions binding objects are subject to, as described in [19, 3–7.3.2], and may include control interfaces, too. Such control interfaces will be represented in Maude using the normal modeling mechanisms described here for representing interfaces.

Signals will be specified in terms of Maude messages. Each Maude message has a name and a set of typed parameters, as specified in its message declaration. Thus, Maude message declarations will represent signal signatures, while message instances will represent concrete ODP signals. Maude messages will be named after the signals they represent.

Operations can be expressed in ODP in terms of signals, and therefore they will be modeled in Maude by the messages that represent such signals. In ODP, flows can also be modeled by sets of signals, and therefore will usually be represented by sequences of Maude messages.

Interfaces will be modeled as Maude objects. The class to which any interface belongs will inherit from a general Maude class `CV-Interface`, which represents a generic interface template:

```
class CV-Interface |
  interfaceType : InterfaceType,
  objectRole    : Causality,
  uniqueId      : Oid,
  bind          : Oid .
```

The attributes of this class store the following information: the type of the interface (stream, operation or signal); the kind of role played by the object that owns the interface (producer, consumer, etc.) [34]; the interface identifier (which is unique for the interface in that computational object); and the reference to a binding object, if a binding is established (otherwise this attribute evaluates to `nil`). The incoming and outgoing messages are placed in internal in and out queues, which are automatically handled by the computational infrastructure (see Section 5.3). Predefined sorts `Oid` and `Cid` are used to represent object identifiers and class identifiers, respectively.

Fig. 1 shows a graphical representation of three computational objects A, B and C interacting at their interfaces. Interface c_1 has been zoomed out to show its internal structure, as modeled in our approach (i.e., using the queues of incoming and outgoing messages).

5.2. Modeling behavior

The functional behavior associated to interactions (signals, operations or flows) is specified by using Maude rules, which provide a powerful mechanism for modeling the conditions under which these actions may occur, and the effects of such actions. More precisely, the left- and right-hand sides of a Maude rule represent, respectively, the configuration of objects and messages before and after the state change of the system.

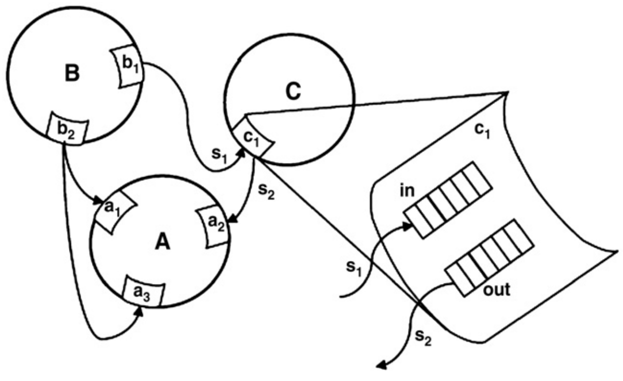

Fig. 1. A graphical representation of objects, interfaces and signals.

Rules allow us to describe the state changes of all the computational objects participating in a given action. Each Maude rule may also contain a guard, which specifies some constraints on when the interaction may occur, hence allowing the specification of many of the environment contract constraints.

5.3. A computational infrastructure

To support the computational specification of systems, we have developeda Computational Infrastructure (CI) which provides a set of common functions that implement the basic mechanisms and structuring rules described in Ref. [19, Part 3–7.2] for the computational viewpoint. They allow computational objects to instantiate interface templates (i.e., create their interfaces), send or respond to signals through any of its interfaces, bind interfaces, create other computational objects (by instantiating the appropriate object templates), etc. All these common tasks are handled by the CI, abstracting away all their details from the user specifications-and hence simplifying them.

The functions provided by the CI have been implemented in Maude, and the way to access them is by sending Maude messages. These messages can be sent by any Maude object representing a computational object, and will be handled by the CI. The following are some of the most relevant ones:

- Messages `send(IF, SIGNAL)` and `receive(O', IF', IF, SIGNAL)` allow a computational object to send and receive a signal through one of its interfaces, respectively. In them, `IF` is the local interface; `O'` and `IF'` are the remote computational object identifier and its corresponding interface; and `SIGNAL` is the Maude message representing the signal.
- Messages `bind(IF, IF')` and `unbind(IF)` allow explicit binding, where `IF` represents the identifier of the initiating interface and `IF'` represents the remote interface identifier. Multi-party binding is also allowed. Compound binding is modeled in terms of primitive bindings.
- Messages `lookup(UID, IFClass, O)` and `lookupResponse(UID, IFClass, O, IF)` can be used to obtain the appropriate interface identifier of a remote computational object `O`, where `UID` is the identifier of the required interface, `IFClass` is the class identifier (of sort `Cid`) of that interface, and `IF` is the interface identifier obtained.
- Other basic functions supported by the Computational Infrastructure include the instantiation of interface templates and computational objects:
  – message `instantiateInterface(IFTemplate, IFKind, O, Caus, UID)` instantiates an interface of object `O`, with interface identifier `UID` (for trading purposes), from the interface template `IFTemplate`, of kind `IFKind` (signal, operation, or stream) and with causality `Caus`.
  – message `instantiateObject(OID, OTemplate, ATTS)` instantiates an object with identifier `OID`, from the object template `OTemplate`, and with attributes' values given by `ATTS`.

The way in which these messages are internally handled by the CI is transparent to the specifier of the computational viewpoint of the system.

5.4. Modeling environment contracts

Environment contracts specify constraints on the interactions between interfaces and their environments. As stated in [30], "an environment contract specifies at the same time expectations from an object on its environment, and guarantees or obligations that the object will fulfil in return". Existing proposals (e.g., [14]) for expressing contracts use pre- and post-conditions, with logical predicates usually written in propositional or linear temporal logics.

The way to deal with environment contract constraints will depend on whether they are expectations or obligations. Expectations depend on the behavior of the environment, which is usually out of our control. Therefore, one way to deal with them is by introducing the appropriate rules for allowing the observation of the possible violations of such constraints. Those watchdog rules will determine the appropriate corrective (penalty or incentive) actions or for raising failures.

Obligations impose constraints on the actions that the system is forced to undertake as part of its intended behavior. Obligations can be expressed in Maude as part of the rules that specify the behavior of the interactions of the objects, using the rules' left-hand sides and guards for restricting behavior that does not fulfill such obligations. As we shall illustrate in the case study in the next section, this approach is quite flexible and allows us to express very interesting properties in a simple way.

6. A case study

In order to illustrate our proposal, let us specify a simple example, adapted from [4]: a multimedia distributed system that consists of a video streamer that transmits video frames to a sink, through a series of intermediate channels and transformers (see Fig. 2). Initially, the stream `source` (i.e., an instance of class Source) receives an external request for playing video. Immediately, it begins emitting video frames to the compressor (`Compressor`), which converts the received data into compressed packets that are sent through a network (we model it as a `binding` object, of class Binding). The flow continues until reaching the decompressor (`Decomp`), which translates the received packets into video frames again. The decompressor sends each processed frame to the buffer (`Buffer`), where flows are stored until they are finally delivered to the target object (`Sink`). This process continues until the source object receives the proper request for stopping the video streams.

Computational objects interact at interfaces. Thus, video flows (`str` messages) are sent and received through a stream interface, which is instantiated from the interface class `IStream`. The request for playing (`play`) or stopping the video (`stop`) are received by the `Source` object through its operational interface `IPlayer`. Similarly, the `Buffer` object may emit control signals associated to certain events, such as

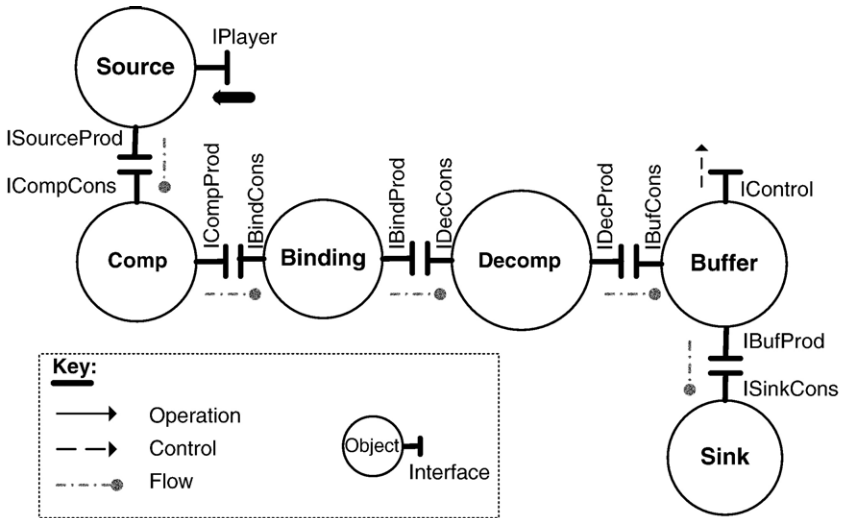

Fig. 2. Example of a simple video stream system.

bufferEmpty or bufferFull. These are sent through an interface instantiated from the interface class IControl.

From a behavioral point of view, the computational objects Compressor and Decompressor have two states (Idle and Busy). They receive a flow at their consumer interface and process it before they send it again through the producer interface, if appropriate. The Sink object has just one state, receiving packets. Similarly, the Binding object, which represents the existing connection, has one single state, in which the transmission of every received flow from the source to the sink is achieved. The corresponding state diagrams are shown in Fig. 3.

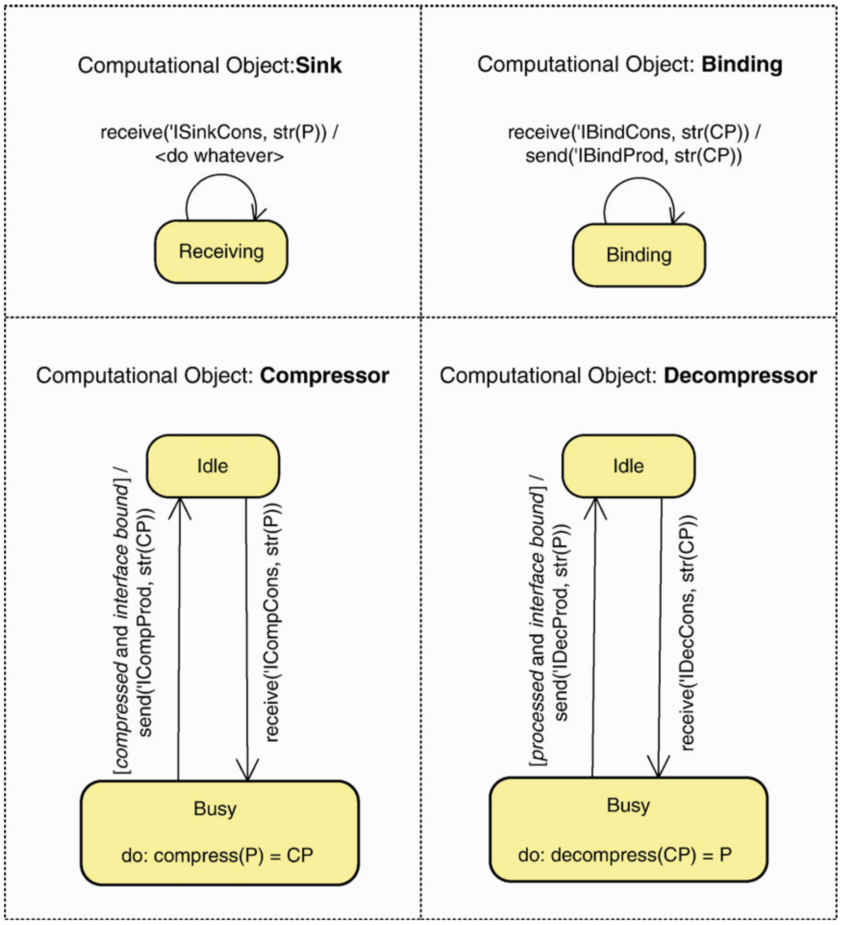
Computational Object:Sink
receive('ISinkCons, str(P)) / <do whatever>
Receiving
Computational Object: Binding
receive('IBindCons, str(CP)) / send('IBindProd, str(CP))
Binding
Computational Object: Compressor
Idle
[compressed and interface bound] / send('ICompProd, str(CP))
receive('ICompCons, str(P))
Busy
do: compress(P) = CP
Computational Object: Decompressor
Idle
[processed and interface bound] / send('IDecProd, str(P))
receive('IDecCons, str(CP))
Busy
do: decompress(CP) = P

Fig. 3. State diagrams of Sink, Binding, Compressor, and Decompressor.

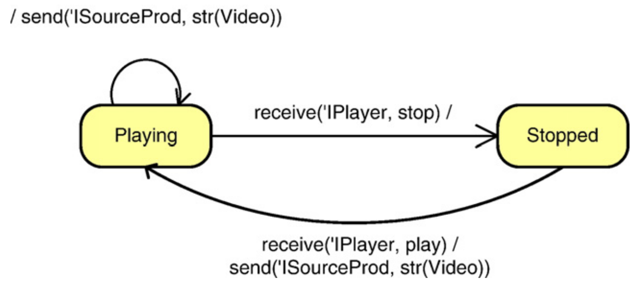
/ send('ISourceProd, str(Video))
Playing
receive('IPlayer, stop) /
Stopped
receive('IPlayer, play) /
send('ISourceProd, str(Video))

Fig. 4. State diagram of the `Source` computational object.

The `Source` computational object toggles between two states, as shown in Fig. 4: `Playing`, in which the object continuously emits video frames to the compressor; and `Stopped`, in which the object does nothing but waiting for a video request. Finally, the `Buffer` computational object has three possible states (`Empty`, `Full` and `Normal`), depending on whether the buffer array becomes empty, full, or it is still admitting incoming messages. As mentioned above, the computational object instantiates a control interface for signaling certain events (i.e., buffer empty or full), which is useful for QoS purposes.

Fig. 5 shows the state diagram of the `Buffer` computational object.

In this example we will consider the following representative QoS constraints (adapted from [4]) that will help us illustrate our modeling approach:

(1) The network connection (i.e., the `Binding` object) between the `Source` and the `Sink` objects loses a message with probability 0.05.
(2) The `Compressor` and `Decompressor` objects process each frame in less than 5 milliseconds.
(3) The network connection takes at least 7 ms. to transport each frame.
(4) The data `Source` generates frames at a rate of at least 30 frames per second.

Once we have identified the system elements, we are ready to formalize them in Maude. Our computational specifications will be divided into four main parts: computational templates (§ 6.1), behavioral specifications (§ 6.2), environment contracts (§ 6.3), and initial configuration (§ 6.4).

6.1. Specifying objects and interfaces

The following Maude classes represent the computational object templates that specify the objects in the system. All these classes are subclasses of `CV-RTObject`, since we will impose some real-time constraints on them.

```
class Source |
  currentId : Nat,
  currentFrame : Video,
  lastSent : Nat .
class Buffer |
  maxSize : Int,
  packets : Set{ Packet} .
classes Compressor Decompressor
        Binding Sink .
subclass Source < CV-RTObject .
subclass Sink < CV-RTObject .
subclass Binding < CV-RTObject .
subclass Buffer < CV-RTObject .
subclass Compressor < CV-RTObject .
subclass Decompressor < CV-RTObject .
```

Class `Source` has three attributes: the identifier of the last video frame generated (`currentId`), the content of that frame (`currentFrame`), and the identifier of the last frame sent (`lastSent`). As shown in Fig. 4, any `Source` object can toggle between two possible states (`Playing` and `Stopped`). We can specify object states in different ways, e.g., by some particular property related to each of the states-one way or another depending on the concrete case; by using an attribute that can range over constants representing the different states — e.g., an attribute `state` that can take the values `stopped` and `playing`; etc. In this paper we will specify the possible states of an object (of a given class A) by declaring a subclass of A for each possible state. One object will always belong to one of these subclasses, and will change its classifier when it goes

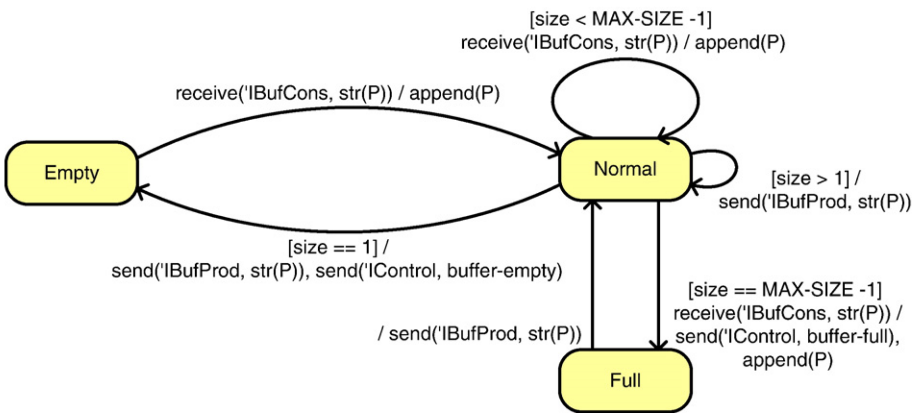
[size < MAX-SIZE -1]
receive('IBufCons, str(P)) / append(P)
receive('IBufCons, str(P)) / append(P)
Empty
Normal
[size > 1] /
send('IBufProd, str(P))
[size == 1] /
send('IBufProd, str(P)), send('IControl, buffer-empty)
[size == MAX-SIZE -1]
receive('IBufCons, str(P)) /
send('IControl, buffer-full),
append(P)
/ send('IBufProd, str(P))
Full

Fig. 5. State diagram of the `Buffer` computational object.

from one state to another one. Thus, we define two subclasses of class Source (Source-Stopped and Source-Playing) as follows.

```
classes Source-Stopped
        Source-Playing .
subclasses Source-Playing < Source .
subclasses Source-Stopped < Source .
```

The source object will not emit any video frame while it belongs to class Source-Stopped, and will change its class to Source-Playing once a play request is received. In this state, it emits the requested video stream. This is an elegant and natural approach for specifying object behaviors, when they are different at different states. For example, a play request will be only accepted by an object of class Source-Stopped. Likewise, only objects of class Source-Playing will accept stop requests.

Classes Compressor and Decompressor have no attributes, since their objects will just process (compressing and decompressing, respectively) the flow received at their consumer interfaces, sending those transformed packets through their producer interfaces. The following classes specify the two states of a compressor, in terms of subclasses. The Decompressor object is analogously specified.

```
classes Compressor-Idle
        Compressor-Busy .
subclass Compressor-Idle
           < Compressor .
subclass Compressor-Busy
           < Compressor .
```

Classes Binding and Sink do not have any attribute either, apart from those inherited from CV-RTObject. A Binding object acts as a pipeline by communicating the source with the Sink, i.e., representing the communication link between both sides of the network. A Sink object just consumes every video frame received through its consumer interface.

Class Buffer has two attributes: maxSize, which represents the maximum number of packets that can be stored in the Buffer array; and packets, a collection of video frames that are ready to be delivered to the Sink object. A Buffer object has three possible states, which are specified by three Maude subclasses of the Buffer class:

```
classes Buffer-Empty
        Buffer-Full
        Buffer-Normal .
subclass Buffer-Empty < Buffer .
subclass Buffer-Full < Buffer .
subclass Buffer-Normal < Buffer .
```

Computational interface templates are specified using Maude modules. Each module contains the classes representing the interface and the Maude messages representing its associated interactions. These interfaces have no attributes, apart from those inherited from CV-Interface. Our application defines three computational interface templates. For example, the module IPLAYER below declares the class IPlayer, that represents the computational interface template IPlayer, from which operational interfaces will be instantiated.

```
omod IPLAYER is
  pr CV-INTERFACE .
  class IPlayer .
  subclass IPlayer < CV-Interface .
  msgs play stop : -> Msg .
endom
```

Control (from class IControl) and stream (from class IStream) interfaces are analogously specified in their respective Maude modules. IControl declares messages bufferEmpty and bufferFull, and IStream declares a single message str.

6.2. Specifying behavior

As mentioned in Section 5.2, an ODP computational specification should define the behavior for each computational object, and should specify how interactions are achieved and how the system evolves as result of these interactions. In Maude, this is specified in terms of rewrite rules, in which the left-hand side of the rule corresponds to the initial object state and also contains the events required to trigger the transition. The right-hand side describes the final object state. For example, the following rule (stop-playing) describes the behavior of a Source object, in the Playing state, when a stop message is received at its IPlayer computational interface.

```
rl[stop-playing] :
  < O : Source-Playing |
    conf :
     (< I : IPlayer | >
      receive(O', I', I, stop)
      CONF) >
  => < O : Source-Stopped |
       conf : (< I : IPlayer | >
              CONF) >.
```

As we can see, the message representing the stop signal is consumed, and the state of the object changes to Stopped (note that the change of state is modeled with a reclassification of the object).

Transitions constrained by guards may be modeled by using conditional rewrite rules. For example, the state transition from Stopped to Playing may happen only if the IStream interface is bound:

```
crl[ start-playing] :
  < O : Source-Stopped |
      conf :
       (receive(O', I', I, play)
        < I : IPlayer | >
        < I" : IStream |
           interfaceType : stream,
           objectRole : producer,
           bind : B>
        CONF)>
  => < O : Source-Playing |
      conf :
       (< I : IPlayer | >
        < I" : IStream | >
        CONF)>
  if (B =/= nil).
```

Likewise, the following rule models the dog-ear transition in the Playing state, which represents the continuous emission of video frames. This is specified by sending str flows through the IStream interface, whose object causality (role) is producer. This rewrite rule will be continuously ready for execution (and hence periodically executed due to Maude's fairness for executing rules) until a stop interaction is received.

```
  crl[ play-video-stream] :
    < O: Source-Playing |
      currentFrame : V,
      currentId : N,
      conf :
       (< I : IStream |
            interfaceType : stream,
            objectRole: producer,
            bind : B >
        CONF)>
  => <O : Source-Playing |
      currentFrame: nextFrame(V),
      currentId : N+1,
      conf:
       (< I : IStream | >
        send(I, str(# N V #))
        CONF)>
  if (B =/= nil).
```

6.3. Specifying extra-functional requirements

Let us show in this section how to express the environment contracts described at the beginning of Section 6. First, as we discussed in Section 4.2, we need to define the delta and mte operations for all those objects affected by the passage of time. Since every computational object is an instance of a class inheriting from class CV-RTObject, we need to specify how the timers increase. In our case, the time elapse affects equally to every computational object in the system, and therefore we can specify the elapse of an amount of time just by incrementing the timers accordingly.

```
  eq delta(< O : CV-RTObject |
            timer : R >, R')
    = < O: CV-RTObject |
        timer : (R plus R')> .
```

Notice that the equations given in Section 4.2 plus this equation will make the application of the delta function on a computational system produce the increment of the timers of all the computational objects in it. Of course, if we were specifying a system in which different objects react differently as time passes, we could consider a different delta equation for each timed computational object.

Requirement (1) establishes that the binding connection can lose a video frame with probability 0.05, what we model by considering that the Binding object does not deliver sometimes the received stream. Assuming a function random that returns a natural number smaller than its argument, we just need to generate a natural number smaller than 100; if the result is greater or equal than 5 then we send the message, otherwise we do not send it. Requirement (3) also affects the Binding objects, since it says that the network connection takes at least 7 ms to transport each frame. This requirement has been modeled by the condition in the following transmission rule.

```
  crl[ transmit-flow] :
    < O : Binding | timer : R,
      conf :
       (< I : IStream | >
        receive(O', I', I, P)
        CONF) >
    => <O : Binding |
        timer : 0,
        conf :
          if random(100) >=5
          then (< I : IStream | >
                send(I", P)
                CONF)
          else (< I : IStream | >
                CONF)
          fi >
    if R > 5.
```

Let us now consider Requirement (2), which states that the Compressor and Decompressor objects process each frame in less than 5 milliseconds. We focus on one of them, namely the Compressor. The specification of the Decompressor follows the same pattern.

As we discussed in Section 6.1, the two states of a compressor are modeled by the two subclasses Compressor-Idle and Compressor-Busy. We model the transitions from one state to the other with two instantaneous rewrite rules, and consider the compression time as the time elapsed from the application of one rule to the other. To do it we use the timer of the Compressor object as described below.

While the compressor object is waiting for something to happen, as an instance of Compressor-Idle, the attribute timer is not considered at all. However, if a flow package str(NCP) is received, then the timer counter is reset to zero and the object state toggles to Compressor-Busy.

```
rl[ receive-video-stream] :
   < O : Compressor-Idle |
      conf :
       (< I : IStream |
           uniqueId : 'ICompCons >
        receive(O', I', I, str(NCP))
        CONF) >
=> < O : Compressor-Busy |
      timer : 0 >.
```

Notice that a compressor object can receive a new frame only if it is of class Compressor-Idle; the signal must come through its consumer interface, the one with identifier 'ICompCons.

The *working* compressor should process the received package before its timer reaches the maximum allowed time and send it again through the producer interface (which should be bound to the consumer interface of some other object). This is specified in the following rule, that defines also how the state toggles to Compressor-Idle.

```
crl[ compress-video-stream] :
   < O : Compressor-Busy |
      timer : R,
      conf :
       (< I" : IStream |
           uniqueId : 'ICompProd,
           bind : B'>
        receive(O', I', I, str(NCP))
        CONF) >
   => < O : Compressor-Idle |
        timer : 0,
        conf :
         (< I" : IStream | >
          send(I" ,
            str(compress(NCP)))
          CONF) >
   if (B' =/= nil) .
      *** the producer interface is bound
```

To make sure that the compression task does not take more than 5 ms, we define the mte operator for Compressor-Busy objects. With the following rule, we make sure that the instantaneous rewrites rules, and in particular the compress-video-stream rule, are used in the specified time.

```
eq mte(< O : Compressor-Busy |
           timer : R >)
   = 5 - R .
```

The last requirement (number (4)) establishes that the Source object should generate at least 30 frames/s. One way of achieving this is by making the Source object generate at least a frame every 33 ms, that is, a new frame must be generated at most 33 ms after the previous one. We also specify this kind of requirements using timers.

The Source object has two states (Playing and Stopped). While stopped, time does not affect the object and hence its timer is not relevant for the rules. However, while playing, the object should generate a new frame at least every 33 ms, emitting frames through the producer interface. Rule play-video-stream specifies the generation of frames:

```
crl[ play-video-stream] :
   < O : Source-Playing |
     currentVideo : V,
     currentId : N,
     conf :
       (< I : IStream |
          interfaceType : stream,
          objectRole : producer,
          bind : B>CONF) >
   => < O : Source-Playing |
        timer : 0,
        lastSent : N,
          currentVideo : generate(V),
             *** next frame generated
          currentId : s(N),
             *** next frame identifier
          conf :
            (< I : IStream | >
             send(I, str(# N V #))
             CONF) >
   if (B =/= nil) .
      *** the producer interface bound
```

As for the Compressor, the requirement is specified by the effects of operation mte on objects of class Source-Playing.

```
eq mte(< O : Source-Playing |
           timer : R >)
   = 33 - R .
```

6.4. Execution, search, and model-check of the system specifications

Once the system specifications are written using this modeling approach, what we get is a rewriting logic specification of the system. Since the rewriting logic specifications produced are executable, this specification can be used as a prototype of the system, which allows us to simulate it. Notice however that when executing the system we only get one of the possible executions. After illustrating its execution, we will see below how we can explore all possible rewrites from a particular initial configuration by using Real-Time Maude's searching and model-checking facilities.

Either for executing, searching, or model-checking we need an initial configuration. We may create such an initial configuration just by giving a Maude term representing it, or more appropriate, by using the messages provided by the CI,

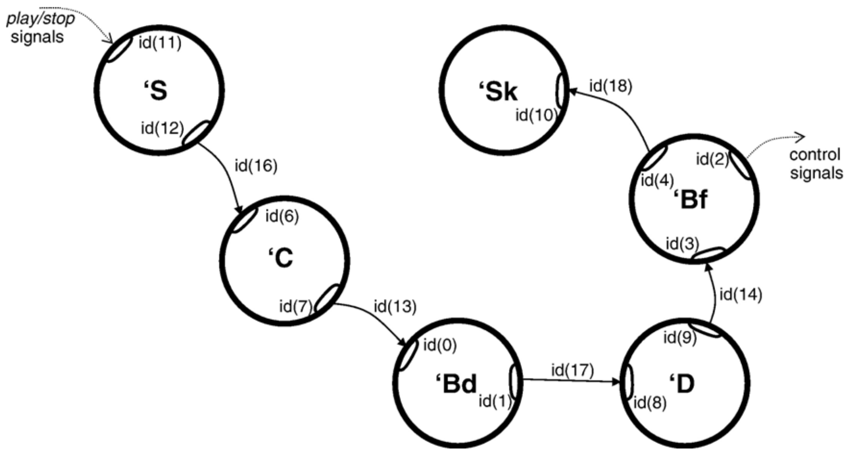

Fig. 6. A graphical representation of the initial computational state.

namely `instantiateObject`, `instantiateInterface`, `bind`, etc.

A graphical representation of one initial state, as created using the CI messages, is depicted in Fig. 6 – for the sake of simplicity, other infrastructure-specific messages and objects have been omitted – which reflects the structure of the system in Fig. 2. This configuration consists of a *Source* object `'S`, one compressor object `'C`, one binding object `'Bd`, one decompressor object `'D`, one buffer object `'Bf`, and one sink object `'Sk`. To understand the way in which the rewrite rules specifying the behavior of the system are executed, it is interesting to see the term representing such an initial state, which is shown in Fig. 7. Let us call `InitialConf` to that initial configuration.

The Real-Time Maude toolkit allows the analysis of our timed specifications using a set of commands for performing high-performance rewriting. In this way, an execution simulates *one* of the possible behaviors of the system from the initial

```
< 'S : Source-Playing | timer: 0, currentId: 1,
     currentVideo: 1241, lastSent: 0,
     conf : (< id(11) : IPlayer| bind : id(15),uniqueId:'IPlayer,
                 interfaceType: signal,objectRole: responder>
             < id(12):IStream| bind: id(16),uniqueId: 'ISourceProd,
                 interfaceType:stream,objectRole: producer>) >

< 'C : Compressor-Idle | timer : 0,
     conf : (< id(6):IStream| bind: id(16),interfaceType : stream,
                 objectRole: consumer,uniqueId: 'ICompCons>
             < id(7):IStream| bind : id(13),interfaceType : stream,
                 objectRole: producer,uniqueId: 'ICompProd>) >

< 'Bd : Binding| timer : 0,
     conf : (< id(0):IStream| bind : id(13),interfaceType : stream,
                 objectRole: consumer,uniqueId: 'IBindCons>
             < id(1):IStream| bind : id(17),interfaceType: stream,
                 objectRole: producer,uniqueId: 'IBindProd>) >

< 'D : Decompressor-Idle | timer : 0,
     conf : (< id(8):IStream| bind : id(17),interfaceType:stream,
                 objectRole: consumer,uniqueId: 'IDecCons>
             < id(9):IStream| bind : id(14),interfaceType : stream,
                 objectRole: producer,uniqueId: 'IDecProd>) >

< 'Bf : Buffer-Empty|maxSize: 15, packets: empty-set,timer: 0,
     conf : (< id(2):IBufferControl | bind : nil, uniqueId: 'IControl,
                 interfaceType : signal,objectRole: initiator>
             < id(3):IStream| bind: id(14),interfaceType: stream,
                 objectRole: consumer,uniqueId: 'IBufCons>
             < id(4):IStream| bind: id(18),interfaceType: stream,
                 objectRole: producer,uniqueId: 'IBufProd>)>

< 'Sk : Sink | timer : 0,
     conf : (< id(10):IStream| bind: id(18),interfaceType: stream,
                 objectRole: consumer,uniqueId: 'ISinkCons>) >
```

Fig. 7. A textual representation of the initial computational state.

configuration. Since we are dealing with timed modules, it is natural to relate the rewriting to the time elapsed in the system, instead of considering a *n*-step rewrite execution.

As described in Ref. [32], Real-Time Maude allows timed rewrite and timed fair rewrite (depending on how the rules are selected for execution) to simulate one sequence of possible rewrite steps starting with an initial configuration (the initial state of the system). Since our specification contains a large number of elements, it is recommended to execute a timed bounded rewriting-otherwise we may get a nonterminating execution. As a result of that timed rewrite, what we get is the last state of the simulated behavior, together with a time stamp with the total duration of the rewrite sequence. For example, we may use the following `tfrew` command to execute a fair rewrite of the system:

```
$ (tfrew[ 200000] InitialConf
    in time <= 300 .)
```

where `[ 200000]` establishes the upper bound on the number of rewrite steps to perform, `InitialConf` (of sort `GlobalSystem`) contains the initial configuration of the system, and `in time <=300` fixes the maximum time elapse for this execution.

As mentioned above, rewrite commands allow us to execute just one sequence of rewrite steps, which may not be enough in many situations. Real-Time Maude also takes advantage of Maude's search capabilities to provide timed search commands that also allow us to analyze all possible behaviors starting from an initial configuration. More precisely, the timed search obtains states that are reachable in a certain time interval from the initial state. For example, suppose we want to search for those executions in which the buffer gets full, starting from the initial configuration `InitialConf`, and having a execution bound of 150 time units. The `tsearch` command can be used for such a search as follows.

```
$ (tsearch InitialConf =>*
     {< O:Oid : Buffer-Full |
        ATTS:AttributeSet >
      CONF:Configuration}
    in time <150 .)
```

The result will be a collection of states (solutions) that fulfil the given search pattern within the specified time frame. In this way we can search for executions leading to undesirable states that we want to avoid in the system, or for situations that violate any of the properties that we want to prove on the system. For example, we can check that the frames are received in the same order as they were emitted, by searching for states in which a frame is received with a smaller number than any of its predecessors. And if frames are timestamped, we can also check some jitter or throughput properties of the system, e.g., that the delay of frames between their emission at the source and their reception at the sink does not exceed a given threshold; or that the delay between two consecutive frames arriving at the sink is not above another threshold.

It always makes sense to give an upper bound to the time interval to explore in order to restrict the number of states of the search space. This is not only due to the intrinsic non-terminating nature of most real-time systems, but also because of the state explosion of the search tree, that will easily overflow the computational capabilities of any computer.

Other interesting tools provided by Real-Time Maude allow us to perform untimed searches (`utsearch`) or to find the shortest and largest time to reach a given configuration (`find earliest` and `find latest`, respectively). The search commands are very useful, and usually help to uncover many undesired situations. However, by searching we cannot reach any definitive conclusion, we can only gain certain level of confidence in the specification. Maude offers a linear temporal logic explicit-state model checker [13], which is properly extended by Real-Time Maude to provide time-bounded model checking as well as untimed model checking for the system to be analyzed. Although the model checker has the restriction that the reachable states from a given initial state must be finite, one may use abstraction techniques [28] to make it finite. Just as for search, Real-Time Maude provides timed and untimed model checking commands.

7. Related work

Formal description techniques are being extensively employed in ODP and have proved valuable in supporting the precise definition of reference model concepts [7,37]. Among all the works currently available, the most widely accepted notations for formalizing the computational viewpoint are Z, LOTOS, and SDL.

Initially, LOTOS and SDL were chosen because they are notations specifically designed to deal with computational descriptions of systems. A formal semantics of some of the concepts of the computational viewpoint language can be found in Part 4 of the RM-ODP [19], using these notations. Z also offers interesting benefits for describing ODP constructs [21], and particularly for writing computational viewpoint specifications, as described in [38]. However, Z is not object-oriented, does not allow modularity, and has some limitations for expressing invariants and constraints using temporal logic, which is a logic in which many environment contracts are expressed.

Object-Z solves most of the Z limitations since it is object oriented, allows modularity, and incorporates a subset of temporal logic for expressing environment contract constraints. However, the use of Object-Z for writing computational specifications also presents some shortcomings when compared to Maude. In particular, Maude offers far more tool support than Object-Z does. Even if some animation can be obtained with Object-Z, it does not reach the level that can be obtained with Maude's execution facilities and strategies. Besides, object templates are usually represented in Object-Z as classes, which is the natural way of doing it. However, Object-Z does not allow dynamic reclassification of objects, which may be the case under some particular circumstances (for instance, it may be required for representing systems in dynamically configurable networks). This is not an issue in

Maude, since the class of an object can be changed during its lifetime.

Najm and Stefani have also used rewriting logic for specifying this viewpoint, at two different levels. First, to specify the computational language concepts themselves, providing formal semantics for both the concepts (e.g., object, binding object, interaction, etc.) and the mechanisms (e.g., internal and external concurrency, message exchange, etc.) used in this viewpoint [29,30]; and second, to write some aspects of the computational specifications, e.g., environment contracts [14]. Our work is more in line with Najm's latter approach, since we rest on the semantics of Maude (for object, class, message, etc.) to build our specifications. The benefits of our contribution, when compared to that of Najm, is that we use a concrete executable language (Maude) and its associated toolkit, instead of an abstract notation for writing the system specifications and a separate calculus for contracts. Furthermore, Maude offers object-oriented modules and constructs, which provides a more natural way to model ODP systems. And finally, the real time facilities provided by Maude are also very valuable for expressing some kinds of computational contracts, as we have shown here.

However, the formality and intrinsic difficulty of most formal description techniques have encouraged the quest for more user-friendly notations. In this respect, the general-purpose modeling notation UML (Unified Modeling Language) is clearly the most promising candidate, since it is familiar to developers, easy to learn and to use by non-technical people, offers a close mapping to implementations, and has commercial tool support. A very interesting and complete proposal by Akehurst et al. [2] uses UML to address computational viewpoint designs, complementing the UML diagrams with the Component Quality Modeling Language (CQML, [1]) for expressing environment contracts constraints. OMG has also proposed the UML Profile for EDOC [33], whose Component Collaboration Architecture (CCA) provides a set of elements and mechanisms very well suited to write ODP computational specifications. However, the lack of precision in the UML 1.5 definition and the semantic gap between the ODP concepts and the UML constructs is so big that the UML proposals may be too large and difficult to understand and use by both ODP and UML users, hence limiting its widespread adoption.

With the advent of UML 2.0 the situation has changed, since not only its semantics have been somehow more precisely defined, but it also incorporates a whole new set of concepts more apt for modeling the structure and behavior of distributed systems. This is why we have explored the use of UML 2.0 for modeling the ODP computational viewpoint concepts, and defined a UML Profile for the graphical description of ODP computational specifications [36]. This has been integrated into a more general standardization effort by ITU-T and ISO/IEC to define the use of UML for ODP system specifications [20]. That proposal solves many of the limitations of previous works, but still only allows the graphical representation of the ODP computational specifications, without any possibilities for deriving implementations, quick-prototyping, nor access to any kind of analysis tool for reasoning about the specifications produced. In this respect, that proposal and the work presented here can complement each other perfectly well.

Finally, other works closely related to the one presented here are those that use rewriting logic and Maude for specifying ODP viewpoints. In our group we have already formalized the enterprise and information viewpoints [11,12], showing how the simplicity and power of rewriting logic can naturally express ODP specifications. Our mid-term goal is, once we count with modeling approaches for representing the ODP viewpoints, to tackle the more ambitious goal of formally specifying and reasoning about ODP correspondences and viewpoint composition and consistency [5,16]. It has been shown that rewriting logic has very good properties as a logical framework, in which representing many different languages and logics, and as a semantic framework, in which giving semantics to them [24,27]. Thus, we think that Maude is a promising option as a unifying framework for the specification of RM-ODP viewpoints in which consistency checks can be rigorously studied.

8. Concluding remarks

Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In this paper, we have shown how Maude can be used for specifying the ODP computational viewpoint concepts and supporting infrastructure, and how to build computational specifications of systems using the Maude concepts and rules. With them we do not only obtain a high-level computational description of the system, but also are in a position to formally reason about the specifications produced and to quick-prototype them.

In addition, our work provides support for the specification and analysis of the real-time properties of ODP systems, something not fully addressed before for this viewpoint. This is of special importance for many critical systems (e.g., embedded applications or control systems) or for widely used applications with strong QoS requirements or temporal constraints (e.g., commercial multimedia applications that charge for their services).

Tool support is an essential issue for any engineering approach to system specifications. Tool support should cover all the system specification life cycle, providing support for writing and validating them, for reasoning about their properties, and even for executing them. As we have mentioned before, access to the Maude toolkit from the UML environment is another goal of our proposal, in which we are currently working. This will allow us to model check the UML specifications, or to prove some of their properties using the Maude theorem prover, without forcing the user to have a strong background and knowledge of Maude or of any other formal notation or method.

Finally, the composition and consistency of the specifications of different viewpoints are the next research areas that we plan to address, once we are able to model ODP's enterprise, information and computational viewpoints using Maude.

Acknowledgements

The authors would like to thank the anonymous referees for their comments and suggestions, that have helped us improve the contents and readability of the paper. We would also like to acknowledge the work of many ODP experts who have been involved in investigating and addressing the problems of the computational specification of ODP systems. Although the views in this paper are the authors' solely responsibility, they could not have been formulated without many hours of detailed discussions with ISO experts on ODP. This work has been supported by Spanish Research Project TIN2005-09405-C02-01.

References

[1] J. Aagedal, Quality of Service Support in Development of Distributed Systems. PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[2] D.H. Akehurst, J. Derrick, A.G. Waters, Addressing computational viewpoint design, 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), IEEE CS Press, Brisbane, Australia, Sept 2003, pp. 147–159.

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

[4] L. Blair, G.S. Blair, A. Andersen, Separating functional behaviour and performance constraints: aspect-oriented specification, Technical Report MPG-98-07, Distributed Multimedia Research Group, Lancaster University, UK, May 1998.

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

[6] A. Bouhoula, J.-P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theoretical Computer Science 236 (1) (2000) 35–132.

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

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

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude 2.0 Manual, June 2003. Available in http://maude.cs.uiuc.edu.

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

[11] F. Durán, M. Roldán, A. Vallecillo, Using Maude to write and execute ODP Information Viewpoint specifications, Computer Standards and Interfaces 27 (6) (2005) 597–620.

[12] F. Durán, A. Vallecillo, Formalizing ODP Enterprise specifications in Maude, Computer Standards and Interfaces 25 (2) (June 2003) 83–102.

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

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

[15] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: J. Goguen, G. Malcolm (Eds.), Software Engineering with OBJ: Algebraic Specification in Action, Kluwer, 2000.

[16] M. Groβe-Rhode, Semantic Integration of Heterogeneous Software Specifications, Springer-Verlag, Berlin, 2004.

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

[18] IEEE, Recommened practice for architectural description of software-intensive systems, IEEE Standard 1471 (2000).

[19] ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing. Geneva, Switzerland, 1997. International Standard ISO/IEC 10746-1 to 10746-4, ITU-T Recommendations X.901 to X.904.

[20] ITU-T Rec. X.906 | ISO/IEC 19793. Information technology - Open distributed processing — Use of UML for ODP systems specifications. Final Committee Draft. Geneva, Switzerland, 2006.

[21] D.R. Johnson, H. Kilov, An approach to a Z toolkit for the reference model of open distributed processing, Computer Standards & Interfaces 21 (5) (Dec. 1999) 393–402.

[22] P. Kruchten, Architectural blueprints — the "4 + 1" view model of software architecture, IEEE Software 12 (6) (Nov. 1995) 42–50.

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

[24] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in: D. Gabbay, F. Guenthner (Eds.), 2 edition, Handbook of Philosophical Logic, vol. 9, Kluwer Academic Publishers, 2002, pp. 1–87.

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

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

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

[28] J. Meseguer, M. Palomino, N. Martí-Oliet, Equational abstractions, Procs. of CADE'03, Lecture Notes in Computer Science, Springer-Verlag, 2003.

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

[30] E. Najm, J.-B. Stefani, Computational models for open distributed systems, in: H. Bowman, J. Derrick (Eds.), Formal Methods for Open Object-based Distributed Systems, Chapman and Hall, 1997, pp. 157–176.

[31] P. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theorical Computer Science 285 (2002) 359–405.

[32] P.C. Ölveczky. Real-Time Maude 2.0 Manual. University of Illinois at Urbana-Champaign and University of Oslo, Oct. 2003.

[33] OMG. A UML Profile for Enterprise Distributed Object Computing V1.0. Object Management Group, Aug. 2001. OMG document ad/2001-08-19.

[34] J.R. Romero, A. Vallecillo, Action templates and causalities in the ODP computational viewpoint, Proc. of the 1st International Workshop on ODP in the Enterprise Computing (WODPEC), Sept. 2004, pp. 23–27, Monterey, California.

[35] J.R. Romero, A. Vallecillo, Formalizing ODP computational specifications in Maude, Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004), IEEE CS Press, Sept. 2004, pp. 212–233, Monterey, California.

[36] J.R. Romero, A. Vallecillo, Modeling the ODP Computational Viewpoint with UML 2.0, Proceedings of the 9th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005), IEEE CS Press, Sept. 2005, pp. 169–180, Enschede, The Netherlands.

[37] R. Sinnot, K.J. Turner, Applying formal methods to standard development: the Open Distributed Processing experience, Computer Standards and Interfaces 17 (1995) 615–630.

[38] R. Sinnot, K.J. Turner, Specifying ODP computational objects in Z, in: E. Najm, J.-B. Stefani (Eds.), Proc. of FMOODS'96, Chapman and Hall, Canterbury, 1997, pp. 375–390.

[39] M.W. Steen, J. Derrick, ODP Enterprise Viewpoint Specification, Computer Standards and Interfaces 22 (2) (Sept. 2000) 165–189.

[40] J.A. Zachman. The Zachman Framework: A Primer for Enterprise Engineering and Manufacturing. Zachman International, 1997. http://www.zifa.com.

José Raúl Romero received the M.Sc. degree in Computer Science from the University of Málaga. After several years working as an IT consultant, he is currently an Assistant Professor at the Department of Computer Science at the University of Córdoba. He is a member of the Software Engineering Research Group at the University of Málaga, and his research interests include model-driven software development, open distributed processing and the industrial use of formal methods.

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

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