Formalizing ODP Computational Viewpoint Specifications in Maude

Raúl Romero and Antonio Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{jrromero,av}@lcc.uma.es

Abstract

The ODP computational viewpoint describes the functionality of a system and its environment, in terms of a configuration of objects which interact at interfaces. Computational objects provide a functional decomposition of the system, independently of its distribution. Although several notations have been proposed to model this ODP 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 explore the use of Maude as a formal notation for writing 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 modelling the ODP computational viewpoint concepts, allows the execution of the specifications produced, and offers good tool support for reasoning about them.

1. Introduction

One of the common ways of dealing with the inherent complexity of specifying distributed systems is by dividing the design activity into a number of areas of concern, each one dealing with a specific aspect of the system. Current software architectural practices define several distinct viewpoints of systems in order to accomplish such specification decomposition. Examples include the viewpoints described in IEEE Std. 1471 [16], the "4+1" view model [18], the Zachman's framework [28], or the ODP Reference Model [17]. In particular, we are interested in the Reference Model of Open Distributed Processing (RM-ODP) framework, which provides five generic and complementary viewpoints on the system and its environment: *enterprise*, *information*, *computational*, *engineering* and *technology* viewpoints. They enable different abstraction viewpoints, allowing participants to observe a system from different suitable perspectives [19].

The *computational* viewpoint describes the functionality of the ODP system and its environment through the decomposition of the system, in distribution transparent terms, into objects which interact at interfaces. In the computational viewpoint, applications and ODP functions consist of configurations of interacting computational objects.

Although the ODP reference model provides abstract languages for the relevant concepts, it does not prescribe particular techniques to be used in the individual viewpoints. The viewpoint languages defined in the reference model are abstract languages in the sense that they define what concepts should be supported, not how they should be represented. 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, 4, 17, 19]. For example, formal description techniques such as Z and Object-Z have been proposed for the information and enterprise viewpoints [27], and LOTOS, SDL or Z for the computational viewpoint [17, 26].

In this paper we explore a new alternative for specifying the computational viewpoint. We propose Maude [5], an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. We have already used Maude for modelling the ODP enterprise viewpoint [10], and the information viewpoint [9]. Here we shall show how rewriting logic and its underlying membership equational logic [22], and in particular Maude, provide the expressiveness required for modelling 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 [14], model checking [11], or theorem proving [7]. Maude offers a comprehensive toolkit for automating such kinds of formal analysis of specifications.

The structure of this document is as follows. First, Sections 2 and 3 serve as a brief introduction to the ODP computational viewpoint and Maude, respectively. Then, Sec-

tion 4 presents our proposal, describing how to write computational specifications in Maude. Section 5 is dedicated to a case study that illustrates our approach. Section 6 relates our work to other similar proposals and, finally, Section 7 draws some conclusions and outlines some future research activities.

2. The Computational Viewpoint

The computational viewpoint is directly concerned with the distribution of processing but not with the interaction mechanisms that enable distribution to occur. The computational specification decomposes the system into objects performing individual functions and interacting at well-defined interfaces.

The heart of the computational language is the object model which defines the form of interface an object can have; the way that interfaces can be bound and the forms of interaction which 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 with 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) without specifying the actual degree of distribution in the computational specification.

2.1. 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, which are shown in Figures 1 and 2.

Objects and interfaces. ODP systems are modelled in terms of *objects*. An object contains information and offers services. A system is composed as a configuration of interacting objects. In the computational viewpoint we talk about *computational objects*, which model the entities defined in an computational specification. Computational objects are abstractions of entities that occur in the real world, in the ODP system, or in other viewpoints [17].

Computational objects have *state* and can interact with their environment at *interfaces*. An interface is an abstraction of the behaviour of an object that consists of a subset

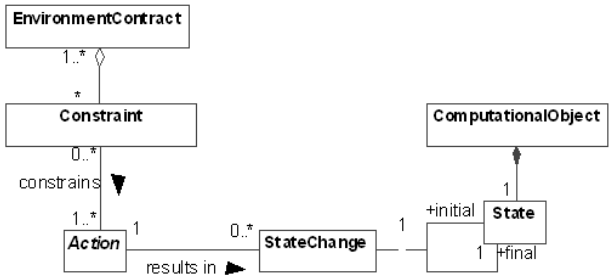

Figure 1. Computational objects and actions

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 (see Fig. 1). In ODP, an object may have multiple interfaces.

Binding objects are computational objects which support a binding between a set of other computational objects. They help compose (synchronize) 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.

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 is then 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 are modelled by other two signals, the one that is sent by the server (the *termination submit*), and

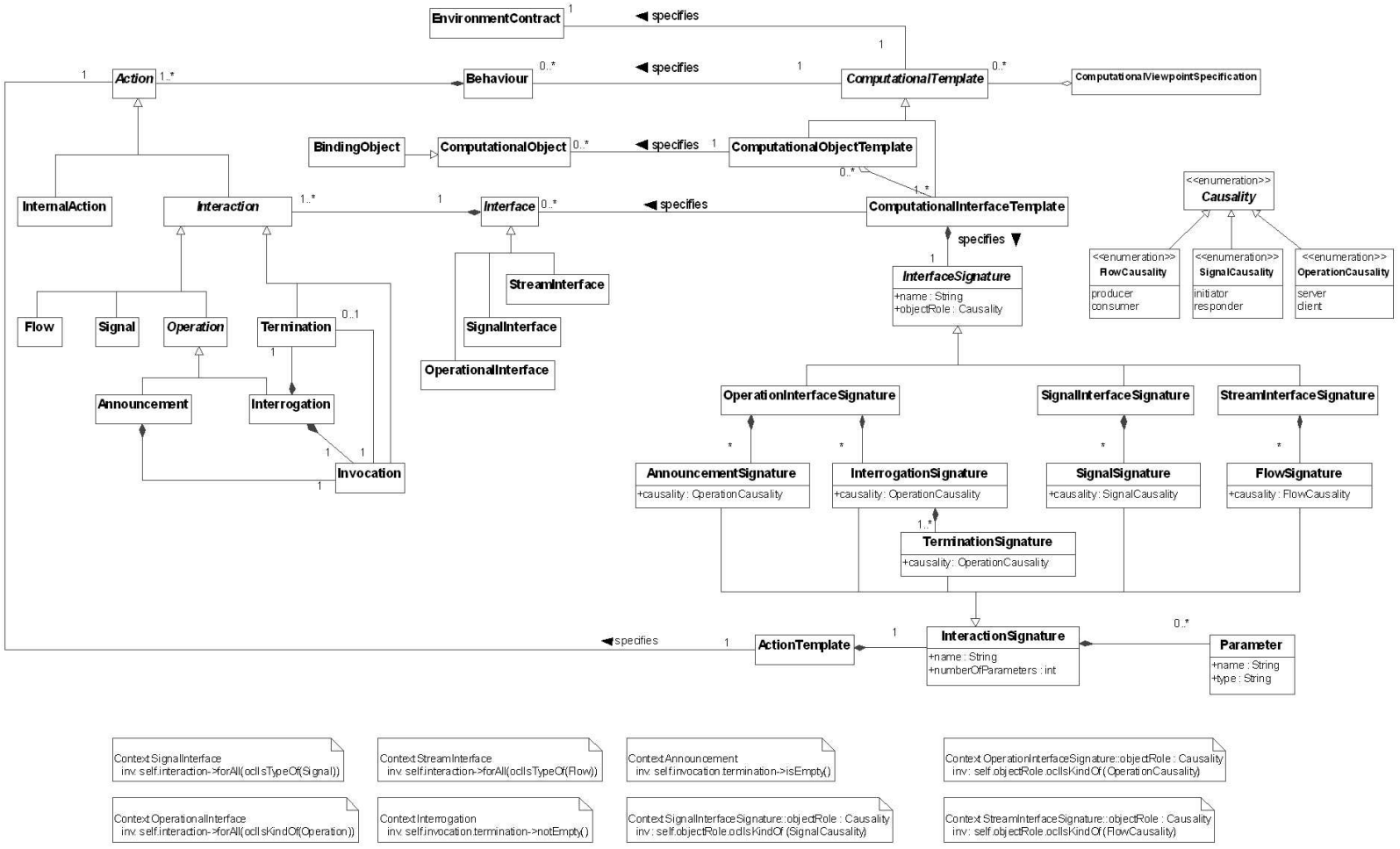

Figure 2. Computational viewpoint concepts

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 are also expressed in terms of signals [17].

Computational templates. Computational objects and interfaces can be specified by templates. In ODP, a *template* is the specification of the common features of a collection of Xs 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 *behaviour* specification; and an *environment contract* specification.

An *interface signature* consists of a name, a causality role (producer, consumer, etc.), and 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).

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's interfaces and its environment. They may specify quality of service constraints, usage and management constraints, etc.

2.2. 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; (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; (c) *failure* rules that apply to all computational objects and identify the potential points of failure in computational activities.

A computational specification defines as well an initial set of computational objects and their behaviour. 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.

3. Rewriting Logic and Maude

Maude [5, 6] is a high-level language and a high-performance interpreter and compiler in the OBJ [13] 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.

Rewriting logic [21] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of (conditional) equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite *rules* of the form $t \rightarrow t'$, where t and t' are Σ-terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

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

In a concurrent object-oriented system, the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The general form of such rewrite rules is

`crl` $[r]$:
$M_1 \ldots M_m < O_1 : C_1 | atts_1 > \ldots < O_n : C_n | atts_n >$
`=>` $< O_{i_1} : C'_{i_1} | atts'_{i_1} > \ldots < O_{i_k} : C'_{i_k} | atts'_{i_k} >$
$< Q_1 : C''_1 | atts''_1 > \ldots < Q_p : C''_p | atts''_p >$
$M'_1 \ldots M'_q$
`if` $Cond$.

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

For instance, the following Maude definitions specify a class `Account` with an attribute `balance` of sort integer, a message `withdraw` with an object identifier 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 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.

```
class Account | balance : Int .
msg withdraw : Oid Int -> Msg .
msg transfer : Oid Oid Int -> Msg .
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 .
```

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 [17, 2-9.21] corresponds to Maude's module inheritance. Throughout the paper, by inheritance we will mean Maude's notion of *class* inheritance, i.e. ODP's subtyping. Multiple inheritance is supported in Maude [5].

4. Writing Computational Specifications in Maude

This Section describes how the computational language concepts can be represented in Maude.

4.1. Modelling objects and interfaces

Basically, as described in [17, Part 3], applications and ODP functions consist of configurations of computational objects interacting at their interfaces.

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 describe the state of the objects of the class.

Computational objects will then be represented by Maude objects. In Maude, each object belongs to a class—although it may change during the object's lifetime—, 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 exhibits:

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

Predefined sort `Configuration` allows to store configurations of Maude objects and messages. Attribute `conf` will store the set of Maude objects representing the interfaces of the computational object. In addition, this attribute may contain the Maude messages that represent the object's internal actions, which do not go through the object interfaces.

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

Binding objects, as computational objects, will be represented by Maude objects. They will adhere to the particular provisions binding objects should be subject to, as described in [17, 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 represented by Maude messages. Each Maude message has a name and a set of parameters of some

sorts, as specified in its message declaration (`msg`). Thus, Maude message declaration will represent signal signatures, while message instances will represent concrete signals. Maude messages will be named after the signal they represent. The identifiers of the objects and interfaces involved in the signal will be added by the computational infrastructure to the envelop of the message that transports such signal.

Operations can be expressed in ODP in terms of signals, and therefore they will be represented in Maude by the messages that represent such signals. In ODP, *flows* can also be modeled by signals, and therefore will usually be represented by 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,
    input         : Configuration,
    output        : Configuration .
```

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.); the interface identifier (which is unique for the interface in that computational object); the reference to a binding object, if a binding is established (otherwise this attribute evaluates to `nil`); and two attributes (`input` and `output`) that represent the sets of incoming and outgoing messages not yet consumed or delivered, respectively. Both configurations will be automatically handled by the Maude infrastructure that we have developed to implement all interaction mechanisms defined by the computational viewpoint. Predefined sorts `Oid` and `Cid` are used to represent object identifiers and class identifiers, respectively.

To support the computational specifications of systems, we have developed a "Computational Infrastructure" that provides a set of common functions that supports the basic structuring rules of the ODP computational viewpoint described in [17, 3-7.2]. This basic functions will allow every computational object to send or respond to signals, instantiate interface templates, bind interfaces, instantiate object templates, etc.

The Computational Infrastructure accepts a set of Maude messages that can be sent by any Maude object representing a computational object. In particular, it accepts two messages for sending and receiving signals: `send(IF, SIGNAL)` and `receive(O', IF', IF, SIGNAL)`, where `IF` is the local interface; `O'` and `IF'` are the remote computational object identifier and 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 modelled in terms of primitive bindings.

In order to obtain the appropriate interface identifier of a remote computational object `O`, messages `lookup(UID, IFClass, O)` and `lookupResponse(UID, IFClass, O, IF)` can be used. In them, `UID` is the identifier of the required interface, `IFClass` is the Class identifier (`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 (`instantiateInterface(...)`) and computational objects (`instantiateObject(...)`), etc. The way in which these messages are internally handled by the Infrastructure is transparent to the software engineer specifying the computational viewpoint of a system.

4.2. Modelling Behaviour

The behaviour associated to interactions (signals, operations or flows) is specified in terms of Maude rules, which provide a powerful mechanism for modeling when these actions occur, and the effects of such actions. More precisely, the left- and right-hand sides of a Maude rule will represent, respectively, the configuration of objects and messages before and after the state change of the system. Rules allow 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 implementation of many of the environment contract constraints.

4.3. Modelling Environment Contracts

Environment contracts specify constraints on the interactions between interfaces and their environments. As stated in [24], "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., [12]) for expressing contracts use pre- and post-conditions, with logical predicates usually written in propositional or linear temporal logics. Maude has very good properties as a logical and semantic framework in which many different languages, logics, and models of computation can be expressed, and in particular, for expressing propositional and temporal logic predicates [20].

The way to deal with environment contract constraints (hereinafter, ECC) will depend on whether they are expecta-

tions or obligations. Expectations depend on the behaviour 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 (e.g., 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 in two different ways. First, they can be expressed as part of the rules that specify the behaviour of the interactions of the objects, using the rules' pre-conditions and guards for restricting behavior that does not fulfill such obligations. In addition, obligations can be separately expressed and then used at the metalevel to constrain the system execution. Maude's reflective capabilities provide very powerful and flexible ways for expressing ECCs at the metalevel, and for controlling the execution of the Maude specifications using them—so invalid states are never reached, or to enforce particular executions under some circumstances. Different built-in strategies for executing specifications are available in Maude, and also facilities for defining our own rewriting strategies are available [5], thus guiding the rewrites depending on our specific needs. The detailed process for expressing constraints and invariants on the system and for defining execution strategies based on them is outside the scope of this paper, and has been reported in [8].

4.4. Reasoning about the Maude specifications

Once the system specifications are written using this modelling approach, what we get is a rewrite logic specification of the system, which can be used for formally reasoning about it. The fact that rewriting logic specifications are executable, will allow us to apply a flexible range of increasingly stronger formal analysis methods, such as runtime verification [15], model checking [11], narrowing analysis, or theorem proving [5]. Each method has its own complexity, and allows different kinds of analyses.

One interesting example of such methods and tools is the inference of properties, verifying that certain proposition holds for the system, i.e., that it can be formally deduced from the axioms defined in the Maude specification. This verification process may be achieved in different ways—and using different techniques and tools—depending on the logic in which the invariant predicates are expressed and the logic supporting the specification notation used. Notice that these two logics may not coincide; they do not coincide in fact in most cases: we may be interested, for example, in verifying whether a certain specification written in membership equational logic satisfies a given property expressed in some temporal logic.

In the case of executable specifications of systems, we can also use a dynamic approach to verify a given specification against a given proposition, by checking that the execution traces of the specification satisfy such a condition. Then, we talk about *model checking*, if we study and check all the possible system execution traces, or about *monitoring*, if we just consider the actual execution trace of the system, checking that the condition always holds for such a trace. As previously mentioned, such conditions can be expressed in different logics. We have already experimented with predicates expressed in propositional logic and linear temporal logic [8]. The temporal logic we have considered is the same that Maude uses in its model checker [11], and the approach used to deal with temporal logic is similar to the one proposed by Havelund and Roşu in [15] for monitoring Java programs.

Another interesting alternative is the "connection" of the formal specification of a system written in Maude with its CORBA or Web-based distributed implementation. Even if a system has been formally specified, and its specification validated for correctness, it is usually very hard to prove that a given implementation of a system conforms to its specification, and that the properties proved for the specification still remain at the implementation level. Probably, the main reason is because specifications and implementations live in separate *worlds*. Reference [2] reports about the way in which Maude objects can be transparently replaced by their CORBA or WSDL implementations, so that objects in both worlds coexist, while still being able to reason about the system. In addition to the usual advantages provided by the use of formal specifications, by allowing objects in any of these worlds (specification and implementation) to seamlessly interoperate we can obtain several interesting advantages, such as building prototypes in which specifications and final implementations are combined, directly using Maude specifications for testing component implementations, and checking the specification of a new component against a running system, without having to fully implement the new component.

5. An Example

In order to illustrate our proposal, we will specify a simple example in Maude. It is a multimedia system composed of *listeners* that want to receive *audio frames* (e.g., listen to a radio program) from a given *audio streamer* (e.g., a radio station or some kind of audio emitter). Apart from these two objects, *binding objects* are in charge of the actual transmission of the audio frames to all listeners attached to a given channel, and a *service manager* object controls the selection of channels by the listener and the configuration of the corresponding binding objects (see Figure 3). For brevity we will concentrate here on the basic system functionality,

omitting many other details that a real system may exhibit.

As mentioned in previous sections, one relevant aspect of a computational viewpoint specification is the decomposition of the system into a configuration of computational objects that interact at interfaces. In our example, a `Listener` computational object will ask the `ServiceManager` for joining the `BindingObject` that distribute the `AudioFrames` from `AudioStreamer` serving the selected channel. Audio frame distribution is performed by means of flows.

The objects of the system interact at several interfaces. Operation interface `IAudioChannel` defines an operation `selectAudioStreamer` (not show in the diagram) that allows listeners to register for a particular audio channel so they can start receiving audio frames from the corresponding binding object. This interrogation may have two terminations: `selectAudioStreamerResponse` and `selectAudioStreamerFailure`, depending on the success or failure of the registration process. Control interface, `IControl`, is used by service managers to ask binding objects to create a new stream source and to include the listener in the list of objects they send audio frames to (using the interrogation `newStreamPort`). Control interfaces are modeled as normal operation interfaces.

Operation interface `IRegistry` allows the `AudioStreamer` to register the control interface of its binding object in the interacting `ServiceManager`. This process is carried out by the interrogation `addAudioStreamer`, which is followed by two possible terminations: `addAudioStreamerResponseOK` (the service manager accepts the registration) and `addAudioStreamerToManyStreamersFailure` (if the maximum number of streamers connected to the service manager has been reached). Another operation interface is defined, `IBinding`, which provides the announcement `registerBOInterfaces`, that allows the binding object to specify its main interfaces to the `AudioStreamer` that instantiated it.

Finally, stream interface `IAudioStream` is used to transfer audio frames between audio sources and the binding objects, and then from the binding objects to their subscribed listeners, multicasting the frames received from the audio sources to them.

Figure 3 shows and informal diagram with the main objects and interfaces of the system.

5.1. Formalizing objects and interfaces

Once we have identified the elements of the system, we are ready to translate them into Maude. Our Maude specifications will be divided into three main parts: computational templates, behaviour specifications and an initial configuration.

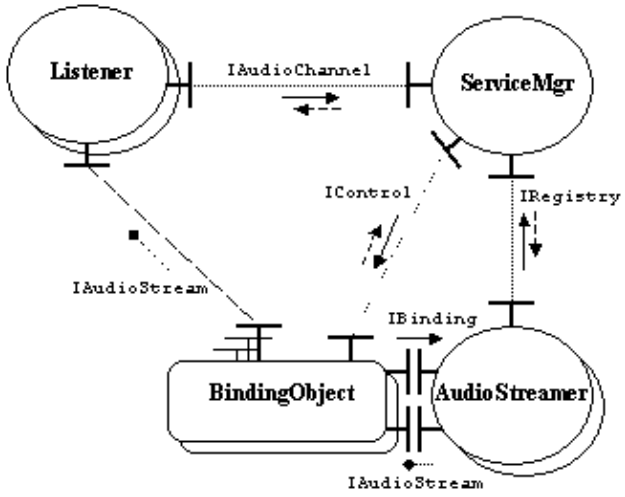

Figure 3. Example of a simple audio server system

The following Maude classes represent the computational object templates that specify the objects in the system. All these classes are subclasses of `CV-Object`.

```
class Listener |
    channelSelected : Qid,
    channelStreamer : Oid,
    manager : Oid .
class ServiceMgr |
    maxStreamers : Int,
    streams : PFun(Qid | Oid) .
class AudioStreamer |
    streamConsumerIF : Oid,
    name : Qid,
    serviceManager : Oid,
    nextFrame : AudioFrame .
class BindingObject |
    consumers : Set(Oid) .
```

Class `Listener` contains three attributes: the name of the audio channel selected, the interface identifier of the stream producer instance in the binding object, and the object identifier of the service manager that serves the listener.

Class `ServiceMgr` contains information about the maximum number of streamers allowed (`maxStreamers`) and a partial function `streams` that provides the control interface associated to a given channel name.

Class `AudioStreamer` has four attributes: the consumer interface identifier in the binding object, the channel name, the identifier of the service manager in which the audio streamer is registered, and the next `AudioFrame` to be transmitted.

Finally, class `BindingObject` has only one attribute, `consumers`, that stores the set of current listeners.

Computational interface templates are specified in Maude modules. Each module contains the classes representing the interface and the Maude messages representing its associated interactions. The interfaces of our example do not have attributes, apart from those they inherit from the base class `CV-Interface`.

```
(omod IAUDIOCHANNEL-SPEC is
  pr CV-INTERFACE .

  class IAudioChannel .
  subclass IAudioChannel < CV-Interface .
  msg selectAudioStreamer : Qid -> Msg .
  msg selectAudioStreamerResponse :
      Qid Oid -> Msg .
  msg selectAudioStreamerFailure :
       Qid -> Msg .
endom)
```

The rest of the interface templates are defined analogously.

5.2. Formalizing behaviour

As mentioned in Section 2.2, an ODP computational specification should also define the behaviour for each computational object, specify how interactions are achieved, and how the system evolves as result of the interactions. In Maude, this is specified in terms of rules.

As an example, we will show the specification of the channel selection process, by which a listener selects an audio streamer by its channel name. First, it interrogates its service manager about the producer stream interface of the corresponding binding object. Then, the service manager locates the control interface of the binding object that serves that stream, and asks the binding object to add the new listener to its list of consumers, and to create a new stream interface for producing the stream to the listener. This selection operation may fail if the channel does not exist. For the sake of brevity, we will just show those rules referring to the service manager (`ServiceMgr`).

First, let us specify the case in which the selection process fails because the requested channel does not exist, i.e., has not been previously registered in the service manager by any audio streamer.

The following rule specifies the behaviour if a `selectAudioStreamer(Q)` message is received at the operational interface `IAudioChannel` for a channel named `Q`, that is not registered. The rule shows that the same interface `I` is used again to send the failure termination back, if there is still a binding object (`BO`) associated to the interface.

```
crl [selectAudioStreamer-failure] :
 < O : ServiceMgr |
  streams : PF,
  conf : (
  receive(O', I', I, selectAudioStreamer(Q))
  < I : IAudioChannel | bind : BO,
        objectRole : server,
        uniqueId : 'IAudioChannel >
  CONF ) >
=>
  < O : ServiceMgr |
     conf : (
     send(I, selectAudioStreamerFailure(Q))
     < I : IAudioChannel | >
     CONF ) >
if BO =/= nil and search(PF, Q) == null .
```

A similar rule will specify the behaviour of the service manager when there is no binding object associated to that interface. In that case, no response would take place.

The behaviour of the service manager when the channel is found, is specified by three Maude rules, which correspond to the different states through which the service manager object goes in that case. Roughly speaking, the service manager receives the `selectAudioStreamer` request from the listener. Then, it searches for the control interface of the binding object associated to the provided channel and binds to it. The service manager changes to a new state, in which it waits for the binding to be established. This state will be reached when an internal message, called `ServiceMgr::getNewStreamPort` is received, and some other required conditions —as specified by the preconditions of the Maude rule—are satisfied.

The following Maude rules specify such a behaviour.

```
crl [selectAudioStreamer-response] :
 < O : ServiceMgr |
  streams : PF,
  conf : (
  receive(O', I', I, selectAudioStreamer(Q))
  < I : IAudioChannel | objectRole : server,
        uniqueId : 'IAudioChannel >
  < I'' : IControl |    objectRole : client,
        uniqueId : 'IControl, bind : nil >
  CONF ) >
=>
  < O : ServiceMgr |
    conf : (
      bind(I'', search(PF, Q))
      < I : IAudioChannel | >
      < I'' : IControl | >
      ServiceMgr::getNewStreamPort(Q)
      CONF ) >
if (search(PF, Q) =/= null) .
```

As we can see, after receipt of a message with a request to select an audio streamer, a `bind` message is produced to achieve the binding of the control interface.

The second rule describes the behaviour of the service manager once the control interface is bound. (Signal `ServiceMgr::getNewStreamPort` represents an internal action, used to control the state of the object.)

```
crl [selectAudioStreamer-contact-with-BO] :
   < O : ServiceMgr |
     conf : (
       < I'' : IControl | bind : BO,
              objectRole : client,
              uniqueId : 'IControl >
       ServiceMgr::getNewStreamPort(Q)
       CONF ) >
=>
   < O : ServiceMgr |
     conf : (
       send(I'', newStreamPort)
       ServiceMgr::respondToListener(Q)
       < I'' : IControl | >
       CONF ) >
if (BO =/= nil) .
```

As a result of the rule, the service manager sends to the binding object a message (`newStreamPort`). That signal is an interrogation that requests the binding object to create a new stream interface and reply with its identifier. The service manager enters a new state (`ServiceMgr::respondToListener`), indicated by the internal action with the same name.

The final Maude rule specifies the behaviour of the service manager when the response of the `newStreamPort` signal is received from the binding object:

```
crl [selectAudioStreamer-respondToListener]:
 < O : ServiceMgr |
   conf : (
     receive(O', I'', I,
             newStreamPortResponse(IAS))
     < I  : IControl | objectRole : client,
            uniqueId : 'IControl >
     < I' : IAudioChannel | bind : BO,
            uniqueId : 'IAudioChannel,
            objectRole : server >
     serviceMgr::respondToListener(Q)
     CONF ) >
=>
   < O : ServiceMgr |
     conf : (
       send(I',
        selectAudioStreamerResponse(Q, IAS))
       unbind(I)
       < I : IControl | >
       < I' : IAudioChannel | >
       CONF ) >
if (BO =/= nil) .
```

As we can see, if the interface `IAudioChannel` is still bound, the response is finally sent to the listener and the control interface is unbound. In this way, the listener will know the stream interface through which the audio frames will be received. By unbinding the control interface, we ensure that the service manager continues responding to other listeners' requests.

To complete the computational specification, we still need to describe a initial configuration of computational objects. In this case it is just a matter of invoking a set of `instantiateObject(..)` operations, one for each computational object we want to include in the initial configuration. From that moment on, the Maude rewrite rules will act on that configuration of objects.

6. Related Work

Formal description techniques are being extensively employed in ODP and have proved valuable in supporting the precise definition of reference model concepts [4]. Among all the works, probably the most widely accepted notations for formalizing the 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 the computational viewpoint in these notations can be found in Part 4 of the RM-ODP [17]. Z also offers interesting benefits for writing computational viewpoint specifications, as described in [26]. However, Z is not object-oriented, does not allow modularity, and has some limitations for expressing invariants and constraints using temporal logic, which is the natural 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 may also present some shortcomings. In particular, oject templates are usually represented in Object-Z as classes, which is the natural way of doing it. However, Object-Z does not offer any mechanism for 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. Besides, 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.

Najm et al., have also used rewriting logic for dealing with 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 [23, 24]; and second, to write some aspects of the computational specifications, e.g., environment contracts [12]. 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 is that we use a precise 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.

On a different arena, UML has also been proposed for ODP computational modelling. UML has an appealing graphical syntax and wide acceptance within the software engineering community. However, their loose semantics may represent an impediment for achieving the precise specification and analysis of systems. There are proposals that try to address this issue using different approaches. For instance, the use of UML Profiles provides customized extensions to UML to deal with specific application domains and systems. This is the approach followed by the UML Profile for EDOC [25], whose Component Collaboration Architecture (CCA) provides a set of elements and mechanisms very well suited to write ODP computational specifications. Another very interesting and complete proposal [1] uses UML to address computational viewpoint designs, complementing the UML diagrams with the Component Quality Modelling Language (CQML) for expressing environment contracts constraints. Despite its lack of formal support, the graphical notations used in these proposals, and the existence of tools for drawing the UML diagrams, provide important advantages over the current formal approaches, which can not be ignored.

7. Concluding Remarks

Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. In this paper, we have explored the possibility of using Maude for specifying the computational viewpoint, showing how to build computational specifications of systems using 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.

There are several research areas that we plan to address in the short term. The first area is related to two important issues in ODP, namely the consistency checking and the composition of specifications of different viewpoints. By establishing the consistency of different viewpoints we simply mean that the specifications of the different viewpoints do not impose contradictory requirements. Checking the consistency of the specifications of different viewpoints is a difficult task, and it is even harder checking it if such viewpoints are specified in different formalisms.

Maude's intuitive style for specifying classes, objects, and rules greatly simplifies the understandability of the specifications produced. Furthermore, the process shown here for writing the Maude computational specifications of a system does not require users to have a deep knowledge of rewriting logic. Thus, it is our belief that Maude specifications could provide a useful vehicle for allowing stakeholders of a system to easily share and discuss about its computational specifications.

Having said that, we also feel that some graphical tool support may be required for the wide adoption of our proposal. In this sense, we are currently working on the smooth integration of our approach with the current proposals for modelling the ODP computational viewpoint using UML. This would allow the stakeholders of the system to use a more user-friendly graphical notation like UML to describe the system computational viewpoint, and then translate them into the corresponding Maude specifications.

Finally, tool support is an essential issue for any engineering approach to system specifications. Tool support should cover all the system specification life cycle, providing support for writing and validating them, for reasoning about their properties, and even for executing them. Access to the Maude toolkit from the UML environment is another goal of our proposal. 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.

Acknowledgements The authors would 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. Thanks also to Akira Tanaka and Dave Akehurst for their comments on previous versions of our computational language metamodel. The anonymous referees have also contributed with their insightful comments and suggestions, helping improve the contents and readability of the paper. This work has been partially supported by Spanish Project TIC2002-04309-C02-02.

References

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

[2] A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: A smooth transition. In *Proc. of JISBD'01*, Almagro, Ciudad Real (Spain), November 2001.

[3] C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and consistent semantics for ODP viewpoints. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, pages 371–386, Canterbury, 1997. Chapman & Hall.

[4] H. Bowman, J. Derrick, P. Linington, and M. W. Steen. FDTs for ODP. *Computer Standards & Interfaces*, 17:457–479, Sept. 1995.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Comput. Sci.*, 285:187–243, Aug. 2002.

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

[7] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000.

[8] F. Durán, M. Roldán, and A. Vallecillo. Invariant-driven strategies for Maude. In *Proc. of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Aachen, Germany, June 2004. Elsevier.

[9] F. Durán and A. Vallecillo. Specifying the ODP information viewpoint using Maude. In H. Kilov and K. Baclawski, editors, *Proceedings of Tenth OOPSLA Workshop on Behavioral Semantics*, pages 44–57, Florida, Oct. 2001. Northeastern University.

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

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

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

[13] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

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

[15] K. Havelund and G. Roşu. Rewriting-based techniques for runtime verification. To appear in Journal of Automated Software Engineering, 2005.

[16] IEEE. *Recommened Practice for Architectural Description of Software-Intensive Systems*. IEEE Std. 1471, 2000.

[17] 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.

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

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

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.

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

[22] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

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

[24] E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Proc. of FMOODS'97*, pages 157–176, Canterbury, 1997. Chapman & Hall.

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

[26] R. Sinnot and K. J. Turner. Specifying ODP computational objects in Z. In E. Najm and J.-B. Stefani, editors, *Proc. of FMOODS'96*, pages 375–390, Canterbury, 1997. Chapman & Hall.

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

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