Modeling the ODP Computational Viewpoint with UML 2.0

José 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 interacting at interfaces, independently of their distribution. Up until UML version 2.0, both the lack of precision in the UML definition and the semantic gap between the ODP concepts and the UML constructs hindered its application for ODP computational viewpoint modeling. With the advent of UML 2.0 the situation may have changed, because its semantics have been more precisely defined and it now incorporates a whole new set of concepts more apt for modeling the structure and behavior of distributed systems. In this paper we explore the benefits provided by the new extension mechanisms of UML and, more specifically, we present a UML profile for modeling the ODP computational viewpoint concepts. We also show a case study that illustrates how our proposal is applied to a multimedia distributed system.

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 [8], the "4+1" view model [13], the Zachman's framework [26], or the Reference Model of Open Distributed Processing (RM-ODP) [9]. In particular, we are interested in the latter framework, the RM-ODP, 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 perspectives [14].

These viewpoints are sufficiently independent to simplify reasoning about the complete specification of the system. The architecture defined by RM-ODP tries to ensure the mutual consistency among the viewpoints, and the use of a common object model provides the glue that binds them all together.

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 notations 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 used, 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 [2, 4, 9, 14]. For example, formal description techniques such as Z and Object-Z have been proposed for the information and enterprise viewpoints [25], and LOTOS, SDL (Specification and Description Language) or Z for the computational viewpoint [9, 24].

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.

Until de advent of UML version 2.0, both the lack of precision in the UML definition and the semantic gap between the ODP concepts and the UML constructs hindered its application in this context. The UML (1.4) Profile for EDOC [17] tried to bridge this gap. But from our perspective, the gap was so big that the Profile ended up being too large and difficult to understand and use by both ODP and UML users. With the advent of UML 2.0 the situation may have changed, since not only its semantics have been more precisely defined, but it also incorporates a whole new set of concepts more apt for modeling the structure and behavior of distributed systems.

In this paper we explore the use of UML 2.0 for modeling the ODP computational viewpoint concepts, and we briefly present a UML (2.0) Profile that allows the graphical description of ODP computational specifications. Our work is intended for two main kinds of audiences. First, it provides ODP modelers with a graphical notation for expressing their ODP system specifications. And second, it provides UML system modelers with a UML Profile that can help them structure their (large) specifications according to a mature and standard proposal.

The structure of this document is as follows. First, sections 2 and 3 serve as a brief introduction to the computational viewpoint and UML 2.0, respectively. Section 4 presents our proposal, describing how to model computational specifications in UML, which is illustrated in Section 5 with a small example.Then, Section 6 discusses some of the issues that we have discovered when trying to use UML 2.0 to represent the ODP concepts. Finally, Section 7 compares our work to other similar proposals, draws some conclusions and outlines some future research activities.

2 Computational Viewpoint in RM-ODP

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 that 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 between engineering and technology specifications (including programming languages and communication mechanisms) thus allowing open interworking and portability of components in the resulting implementation.

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.

Objects and interfaces. ODP systems are modeled 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 a computational specification. Computational objects are abstractions of entities that occur in the real world, in the ODP system, or in other viewpoints [9].

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. ODP objects 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.

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". <X> can be anything that has a type. 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.), 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).

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 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 be expressed in terms of signals [9].

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 Quality of Service (QoS) constraints, usage and management constraints, etc. These 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.

An environment constraint can thus describe both requirements placed on an object's environment for the correct behavior, and constraints on the object behavior in the correct environment.

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; 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.

3 Unified Modeling Language 2.0

UML is a visual modeling language that provides a wide number of graphical elements for modeling systems, which are combined in diagrams according to a set of given rules. The purpose of such diagrams is to show different views of the same system or subsystem and indicate what the system is supposed to do.

There are mainly two types of diagrams: *structural* and *behavioral*. The former ones focus on the organization of the system. Structural diagrams include package diagrams, object diagrams, deployment diagrams, class diagrams and composite structure diagrams. Behavioral diagrams reflect the system response to inner and outer requests and its evolution in time, and include activity diagrams, use cases, statecharts (and protocol state machines) and interaction diagrams (e.g., sequence, communication and timing diagrams).

One of the major improvements of UML 2.0 [19, 20] is the addition of new diagrams and the enhancements made to existing ones: UML 2.0 structure, composite, communication, timing and interaction overview diagrams allow solving many of the UML 1.x limitations. Most of these improvements have been influenced by the integration of the mature SDL language within UML. In addition, UML 2.0 now provides better constructs for modeling the software architecture of large distributed systems, with concepts such as components and connectors, and has promoted the use of OCL (*Object Constraint Language*), now fully aligned with UML 2.0 [18]. Finally, the language extension mechanisms have been greatly enhanced too, with the more precise definition of UML Profiles to allow the customization of UML constructs and semantics for given application domains. These new concepts and mechanisms of UML 2.0 constitute the basis of our proposal.

4 Modeling Computational Viewpoint Concepts in UML 2.0

The UML 2.0 Profile for the ODP Computational Viewpoint (which is fully described in [21]) is made up of three main parts. First, it defines the ODP computational viewpoint metamodel, which is an evolution of the metamodel presented in [23], and defines the semantics, properties and related elements of each metaclass. Second, ODP concepts are mapped to UML elements. This mapping contains information about every ODP computational concept, the UML base element that represents such a computational concept, and the stereotype that extends the metaclass so that the specific domain terminology can be used.

This section summarizes how the main concepts of the ODP computational language are mapped to UML 2.0 concepts. In the following, we will distinguish between the ODP and the UML concepts by writing ODP concepts in *italic typeface* and UML concepts in sans-serif typeface.

4.1 Computational objects and interfaces

Computational object templates and objects. A key concept of the ODP computational viewpoint is the *computational object*. Each *computational object* is instantiated from its corresponding *computational object template*.

A *computational object template* will be mapped to a UML component. UML components represent autonomous system units, that encapsulate state and behavior—their granularity is arbitrary, as the ODP Reference Model requires for *computational objects*—and interact with their environment in terms of provided and required interfaces. In UML, components are classifiers. A UML classifier can have a set of features, that characterize its instances.

ODP *computational objects* will then be mapped to UML component instances.

Computational interfaces. *Computational objects* interact with their *environment* at *interfaces*. These are instantiated from *computational interface templates*, which comprise the *interface signature* (*signal*, *operation* or *stream* as appropriate), a *behavior* specification and an *environment contract* specification.

There are no exact terms in UML 2.0 to provide one-to-one mappings for these ODP concepts. However, the semantics provided by other modeling elements can be used with slight customizations.

According to ODP, 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". Thus, the *interfaces* (considered as instances of *computational interface templates*) of a *computational object* constitute a partition of the *interactions* of that *object*.

Then, if we consider *computational interfaces* as interaction points at which *computational objects* interact, we find that this concept corresponds to the UML concept of interaction point, i.e., a port at the instance level. More precisely, according to the UML 2.0 specification, the required interfaces realized by a port characterize the services that the owning component expects from its environment and that it may access through this interaction point. Similarly, the provided interfaces characterize the behavioral features that the owning component offers to its environment at this interaction point. A behavioral port may be used in order to specify some behavior (e.g., a protocol state machine) associated to some interface.

In ODP, a *computational interface template* comprises an *interface signature*, which is defined as the set of *action templates* associated with the interactions of an *interface*. Each of these *action templates* comprises the *name* for the *interaction*, the number, names and types of the parameters and an indication of *causality* with respect to the *object* that instantiates the *template*.

Then, an ODP *computational interface signature* will be mapped to a set of UML interfaces (see Section 6.4), each of which is defined as a kind of classifier that represents a declaration of a set of coherent public features and obligations. This means that each interface can be considered as the specification of a contract that must be fulfilled by any instance of a classifier that realizes the interface (e.g., the UML component instance that represents the *computational object*, through its corresponding interaction point).

Different stereotypes will be used to distinguish the interfaces that represent the different kinds of *computational interface signatures*. Thus, ODP *signal*, *operation*, and *stream interface signatures* will be represented by interfaces stereotyped ≪CV_SignalInterfaceSignature≫, ≪CV_OperationInterfaceSignature≫ and ≪CV_StreamInterfaceSignature≫, respectively.

All these stereotypes are inherited from a common abstract stereotype ≪CV_InterfaceSignature≫ that represents ODP *computational interface signatures*, and will be constrained by the appropriate OCL constraints that define the kinds of *interactions* that each interface can declare (see Section 4.5). An abstract stereotype, which cannot be instantiated, is very useful when defining UML profiles for avoiding repetition in multiple stereotypes that logically have common properties.

Causalities. As described in [22], in the ODP computational viewpoint *causalities* can be considered at two different levels: at the *action template level* (interaction role) and at the *object*'s *interface signature* level (object role). In UML, that interaction role is expressed by the kind of interface that represents the *interaction*, i.e., depending on whether it is a provided or a required interface. The object role is implicitly defined for *operations*. However, for *signal* or *stream interface signatures*, which comprise *signals* or *flows* that may go in different directions, there is no clear relationship between the *causality* of the *interface signature* and the *causalities* of the individual *interactions* that comprise the *interface signature*. In those cases, the tag objectRole, attached to ports stereotyped ≪CV_CompInterfaceTemplate≫, will be used to indicate the *causality*.

Binding objects. In ODP, a *binding object* is considered as a *computational object* that supports a *binding* between a set of other *computational objects* and that is subject to special provisions as specified by the *binding rules*. Thus, a *binding object* is a special kind of *computational object* and, therefore, can be modeled as a component instance stereotyped ≪CV_BindingObject≫ (this stereotype is inherited from ≪CV_Object≫).

4.2 Interactions

In ODP, the basic one-way communication mechanism from an *initiating object* to a *responding object* is the *signal*, which represents a single basic *interaction* between them. *Operations* and *flows* are also *interactions*, although they can be handled in terms of *signals*, as previously mentioned in Section 2.1. Both synchronous and asynchronous interactions are possible in UML and in ODP [15].

An ODP *signal* will be mapped to a UML message, stereotyped ≪CV_Signal≫, which is the specification of the conveyance of information from one instance to another. In UML, a message can specify either the raising of a UML signal or the call of an UML operation.

ODP *flows* can be considered an *interaction* consisting of *signals*. Thus, *flows* will be generally represented in UML by interactions or just by single messages stereotyped ≪CV_Flow≫. Since a *flow* represents in ODP an abstraction of a sequence of *interactions* from a *producer* to a *consumer*, the appropriate UML behavioral diagram needs to be associated to the corresponding port.

In the case of *operations*, complete messages can be used to represent *interrogations*, i.e., one message representing the ODP *invocation* followed by another message representing the corresponding ODP *termination* in the opposite direction.

Stereotypes ≪CV_Invocation≫, ≪CV_Termination≫ and ≪CV_Announcement≫ will be used to qualify the messages representing the respective *signals*.

In ODP, the duration of *operations* is arbitrary unless otherwise specified by the appropriate *environment contracts* associated to the *objects* and *interfaces* involved. The same happens in UML. In fact, version 2.0 includes new features to specify both time intervals and duration intervals, observations and constraints in behavioral models.

In ODP, in order to specify a *signal* we need to provide its *signature* and its *behavior*.

An *interaction signature* will be represented by an UML reception, which consists of a declaration stating that the interface classifier is prepared to react to the receipt of a signal. As mentioned in [20], "by declaring a reception associated to a given signal, a classifier specifies that its instances will be able to receive that signal, or a subtype thereof, and will respond to it with the designated behavior." In ODP, each *interface signature* comprises a set of *interaction signatures* that conform to the *interface type*. This means that we need to define the proper set of ODP *interactions* as public features of the appropriate UML interface classifier. The appropriate stereotypes indicate the kind of *signature* being represented:

- ≪CV_SignalSignature≫ for *signal signatures*;
- *Interrogation signatures* comprise an *invocation signature* (≪CV_InterrogationSignature≫) and the *signature* of its corresponding *termination* (stereotyped ≪CV_TerminationSignature≫);
- ≪CV_AnnouncementSignature≫ for an *announcement signature*, which comprises just the *signature of its invocation* (≪CV_AnnouncementSignature≫); and
- ≪CV_FlowSignature≫ for a *flow signature*.

The behavior of *interactions* refers to the communication process between *computational objects*, which will be expressed in UML with behavioral diagrams [3]:

- Interaction models describe how messages are passed between objects and cause invocations of other behaviors.
- Activity models focus on the sequence, input/outputs and conditions for invoking other behaviors.
- Finally, state machine models show how events (e.g., signal events) cause changes to the object state and invoke other behaviors.

Which of them to choose is a matter of the system perspective that the modeler needs to specify, since each of these models is focused on a different aspect of the system dynamics.

As a matter of fact, in UML a message can refer to both a signal raising and an operation call. From the ODP perspective, every *interaction* can be expressed in terms of *signals*, which are definitely best suited for many kinds of application domains, e.g., multimedia. However, there are cases in which system designers might prefer to adopt an object-oriented approach that represents the exchange of information in terms of *operation interactions* between *computational objects*. In this case, modeling these *interactions* as UML operations might be simpler.

Bindings between *interfaces* can be represented in UML in terms of links between the ports comprised by the components that represent the interacting *computational objects*.

4.3 Environment contracts

Environment contracts place constraints on the *behavior* of *computational objects*, and usually include QoS, usage, and management aspects. The ODP Reference Model does not prescribe how an *environment contract* must be specified; it just defines this concept and its basic contents.

QoS aspects are mainly focused on three main issues: time (e.g., latency, jitter, etc.), volume (e.g., throughput) and reliability (e.g., permitted percentage of media frames lost). These QoS constraints are generally expressed in terms of *QoS contracts* specified as part of the *environment contract* of the *computational templates*. These QoS contracts usually are expressed as invariants $Req(A) \Rightarrow Prov(A)$, where $Req(A)$ indicates the QoS required by A from other objects and its environment (e.g., the supporting infrastructure or middleware) and $Prov(A)$ indicates the QoS provided by A to its environment. In addition, QoS constraints may imply usage and management constraints too, such as location constraints and distribution transparency constraints.

Each system modeler might like to specify their own constraints in the way that best suits their particular application, and therefore the UML elements (and their semantics) required to model different *environment contracts* can change from one application to another. Thus, instead of incorporating these kind of concepts into our UML Profile, we have decided to use separate profiles for representing QoS and other extra-functional aspects of *environment contracts*. The possibility offered by UML 2.0 to apply multiple profiles to a package—as long as they do not have conflicting constraints—will allow the specifier use the QoS profile(s) of his preference, on top of the ODP Computation Viewpoint profile we are describing here. Figure 1 shows an example in which the QoS constraints are expressed using the OMG's UML Profile for QoS and Fault Tolerance Characteristics and Mechanisms [16]. (Please notice that the application of a profile allows the use of its stereotypes, but does not necessarily require their use.)

4.4 Computational specifications

As described in 2.2, a computational specification describes the functional decomposition of an ODP system, in distribution transparent terms. In UML, the computational specification will be represented by a set of diagrams that model both structural and behavioral aspects of the system. These diagrams will use the elements provided by the applied profiles (using their specified semantics).

A *configuration* of *computational objects* and their interacting *interfaces* will be modeled by component diagrams (at the instance level).

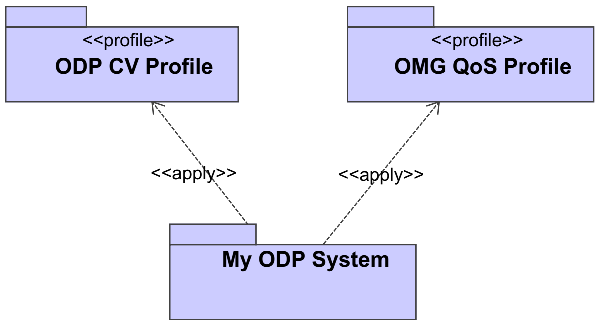


Figure 1. Profile applications

The *internal actions* of those *objects* will be represented by behavioral diagrams associated to the UML components that represent those *objects*. Activity models, which focus on the sequence and conditions for coordinating low-level behaviors, and statecharts, which show how events cause changes of the objects' state, can be used to represent the specification of such *internal actions*. UML 2.0 distinguishes between two different kinds of state machines. Behavioral state machines can be used to specify behavior of (objects of) a class. Protocol state machines, which are defined in the context of one classifier, are used to express usage protocols."

The *interactions* that occur between the *computational objects* can be modeled using interaction diagrams, which come in different flavors. Usually, sequence diagrams are used to model message interchanges between a number of lifelines, each of which represents the interacting UML interface instances. Communication diagrams may also be useful. They provide the vision of how messages are passed from one component instance to another and how they make their sequencing explicit. In addition, interaction overview diagrams are a new variant of activity diagrams [20]. They define interactions in a way that facilitates overview of the control flow. Timing diagrams can be also useful to represent the *interactions* among *computational objects* when some timed simple constraints need to be observed or applied.

4.5 Summary of the mappings

The fact that most ODP concepts can be represented by UML 2.0 concepts without changing their original semantics (maybe imposing some additional constraints on them, at most) enables the use of a UML Profile as the right kind of mechanism for our purposes [7]. Note that the profile mechanism does not allow for modifying existing metamodels. Rather, a profile is intended to provide a straightforward mechanism for adapting an existing metamodel with

Table 1. Computational Viewpoint profile

ODP Concept	UML Base Element	Stereotype	Comments
Computational object template	Component	≪CV_CompObjectTemplate≫	A component should never interact directly at interfaces with its environment but through its non-empty set of ports.
Computational interface template	Port	≪CV_CompInterfaceTemplate≫	Two tags are attached to the stereotyped port: (a) objectRole, which represents the causality of the computational object as a whole; and (b) type, which represents the kind of computational interface (signal, operation or stream).
Signal interface signature	Interface(s)	≪CV_SignalInterfaceSignature≫	An interface comprises receptions specifying signals with the same direction.
Operation interface signature	Interface(s)	≪CV_OperationInterfaceSignature≫	An interface comprises receptions specifying interactions for the same direction.
Stream interface signature	Interface(s)	≪CV_StreamInterfaceSignature≫	An interface comprises receptions specifying flows with the same direction.
Announcement signature	Reception	≪CV_AnnouncementSignature≫	
Interrogation signature	Reception	≪CV_InterrogationSignature≫	
Termination signature	Reception	≪CV_TerminationSignature≫	
Signal signature	Reception	≪CV_SignalSignature≫	
Flow signature	Reception	≪CV_FlowSignature≫	
Environment contract	Package	≪CV_EnvironmentContract≫	Different environment contracts can be represented by separate packages.
Computational object	InstanceSpecification	≪CV_Object≫	It refers to the instance of a component representing a computational object template.
Binding object	InstanceSpecification	≪CV_BindingObject≫	Inherited from ≪CV_Object≫
Signal interface	Port (interaction point)	≪CV_SignalInterface≫	The name of the port references the interface template from which it is instantiated.
Operation interface	Port (interaction point)	≪CV_OperationInterface≫	The name of the port references the interface template from which it is instantiated.
Stream interface	Port (interaction point)	≪CV_StreamInterface≫	The name of the port references the interface template from which it is instantiated.
Signal	Message	≪CV_Signal≫	
Flow	Interaction / Message	≪CV_Flow≫	A sequence of signals or the raise of a single signal.
Announcement	Message	≪CV_Announcement≫	
Invocation	Message	≪CV_Invocation≫	
Termination	Message	≪CV_Termination≫	

constructs that are specific to a particular domain.

As a summary, Table 1 shows the most important stereotypes defined in the UML Profile for the ODP Computational Viewpoint [21]. Note that three important concepts are described in this table: (a) the concept from the ODP computational language; (b) the UML base element that should be used to model that concept; and (c) the name of the stereotype that should be applied to the UML element. The tags, meta-attributes and constraints that complete the definition of the profile have been omitted, but the interested reader can find them in [21].

5 A case study

In the following, we show how the UML Profile for the ODP Computational Viewpoint could be applied to a computational specification. The case study shown has been adapted from [23] and consists of 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. A snapshot of the system is shown in Figure 2. For brevity, we will just concentrate on a reduced subset of the basic system functionalities, omitting many other details that the modeler should have to take into consideration when specifying the real system.

5.1 Computational templates

From the snapshot in Figure 2, we can identify several computational object templates, each of which comprises a set of computational interface templates.

More specifically, four object templates can be identified: Listener, ServiceMgr, AudioStreamer and BindingOb-

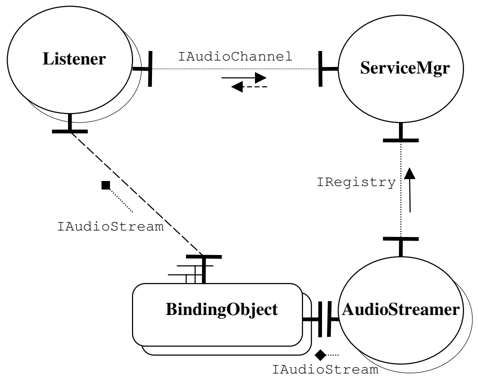


Figure 2. An audio stream application

ject. These all are represented by stereotyped UML components as shown in Figure 3. Three computational interface templates are identified too. In this case, UML stereotyped ports are used to express not only the different kind of existing interface templates (with an indication of type and causality), but also the object templates of which these interface templates are part. More specifically, these interfaces templates are: IAudioChannel, IRegistry and IAudioStream. The first two are operational interfaces and the last one is a template for stream interfaces.

UML interfaces shown in Figure 3 indicate the corresponding signature for each interface template, according to the type of interfaces they will instantiate. For example, the UML interface IAudioChannel_Signature—stereotyped ≪CV_OperationInterfaceSignature≫ because it is an operation interface signature—declares the signatures for both the announcement activateListener and the interrogation selectAudioChannel. Since no tagged values appear attached to the UML interface, the default value for the tag isTermination is applied (false). This port (representing an *ODP interface signature*) has another UML interface, IAudioChannel_SelectionResponse, that represents the corresponding set of terminations for the invocation selectAudioChannel, indicated by the value of the tag invocation of stereotype ≪CV_OperationInterfaceSignature≫. Note that, contrary to UML, in an ODP specification both incoming and outgoing communications require to be explicitly modeled. Consequently, different UML interfaces are needed for the same ODP signature, one for each communication direction. The expected interaction role will determine the kind of association between the UML interface, representing a part of the interface signature, and the UML port, representing the interface template.

5.2 Computational objects and interfaces

As mentioned in Section 2.2, a computational specification describes the functional decomposition of an ODP system in terms of a configuration of computational objects. Figure 4 shows a configuration consisting of two audio streamers (radio stations). One of them is currently connected to the service manager. A given number of listeners are connected to the binding object that manages the distribution of audio packets from one of those streamers. Each of these computational objects is represented in UML by the corresponding instance of the component that specifies its computational object template. Although in general there is no need to name the component instances, sometimes we need to do it in order to distinguish them, and to make explicit the existence of such computational objects in the system (e.g. both audio streamers have a unique name in the model).

Computational interfaces also appear in the diagram, represented by ports attached to the component instances. Note that, at the object level, the stereotype itself indicates the kind of interface that this port represents. For example, the component instance :ServiceMgr contains two ports. Both are stereotyped ≪CV_OperationInterface≫, indicating that the represented computational interface is operational. A link between two ports represents the binding between those computational interfaces.

There is no explicit need to indicate the type of the port that represents the computational interface template from which it is instantiated: indicating the name that identifies such interface template in the object template is enough. Likewise, any additional information can easily be obtained

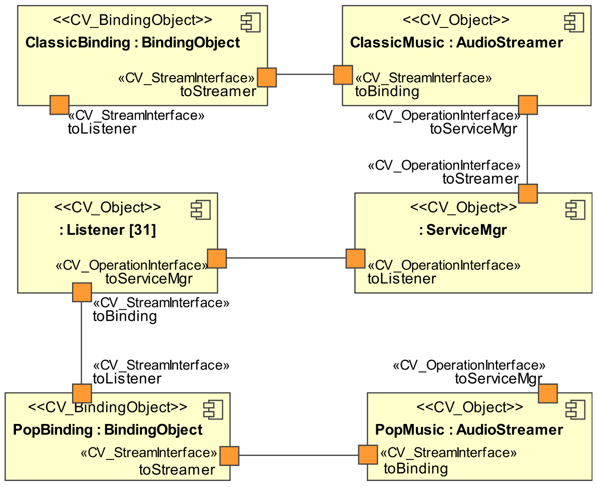
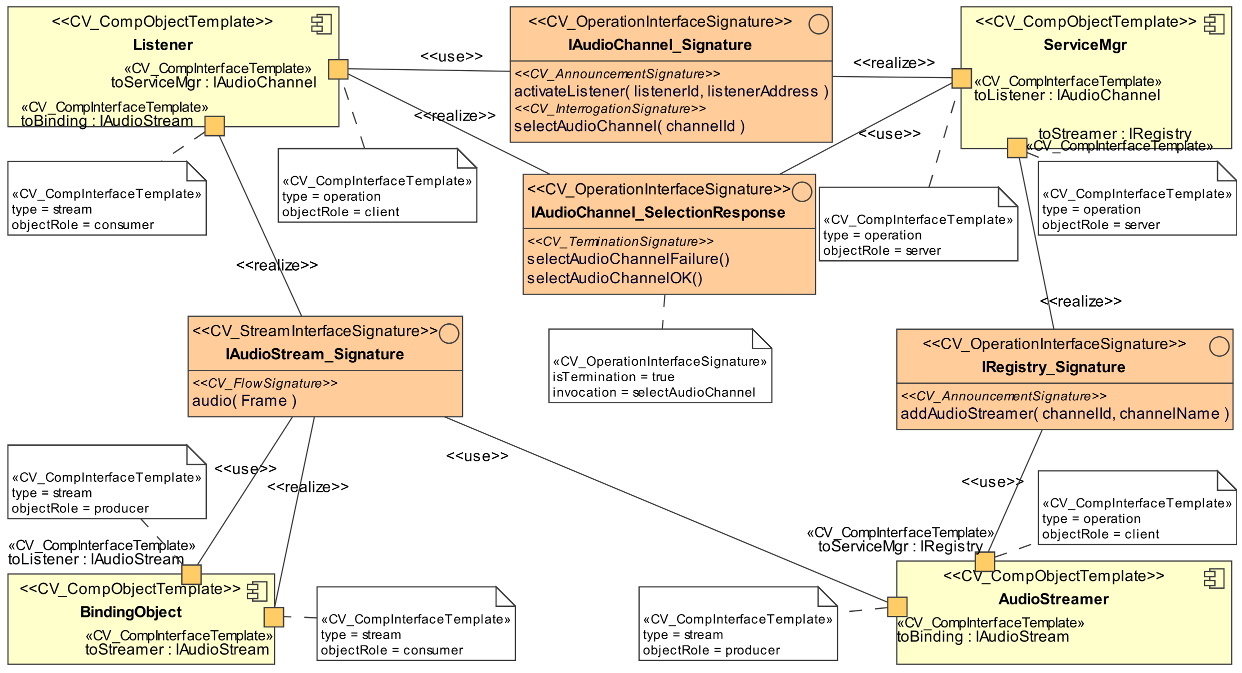

<<CV_BindingObject>>
ClassicBinding : BindingObject
«CV_StreamInterface» toStreamer
«CV_StreamInterface» toListener
<<CV_Object>>
ClassicMusic : AudioStreamer
«CV_StreamInterface» toBinding
«CV_OperationInterface» toServiceMgr
«CV_OperationInterface» toStreamer
<<CV_Object>>
: Listener [31]
«CV_OperationInterface» toServiceMgr
«CV_StreamInterface» toBinding
<<CV_Object>>
: ServiceMgr
«CV_OperationInterface» toListener
«CV_StreamInterface» toListener
«CV_OperationInterface» toServiceMgr
<<CV_BindingObject>>
PopBinding : BindingObject
«CV_StreamInterface» toStreamer
<<CV_Object>>
PopMusic : AudioStreamer
«CV_StreamInterface» toBinding

Figure 4. Computational object configuration



<<CV_CompObjectTemplate>>
Listener
«CV_CompInterfaceTemplate» toServiceMgr : IAudioChannel
«CV_CompInterfaceTemplate» toBinding : IAudioStream
<<CV_OperationInterfaceSignature>>
IAudioChannel_Signature
<<CV_AnnouncementSignature>>
activateListener(listenerId, listenerAddress)
<<CV_InterrogationSignature>>
selectAudioChannel(channelId)
<<CV_CompObjectTemplate>>
ServiceMgr
«CV_CompInterfaceTemplate» toListener : IAudioChannel
toStreamer : IRegistry «CV_CompInterfaceTemplate»
<<use>>
<<realize>>
<<realize>>
<<use>>
«CV_CompInterfaceTemplate» type = stream objectRole = consumer
«CV_CompInterfaceTemplate» type = operation objectRole = client
<<CV_OperationInterfaceSignature>>
IAudioChannel_SelectionResponse
<<CV_TerminationSignature>>
selectAudioChannelFailure()
selectAudioChannelOK()
«CV_CompInterfaceTemplate» type = operation objectRole = server
«CV_CompInterfaceTemplate» type = operation objectRole = server
<<realize>>
<<realize>>
<<CV_StreamInterfaceSignature>>
IAudioStream_Signature
<<CV_FlowSignature>>
audio(Frame)
«CV_OperationInterfaceSignature» isTermination = true invocation = selectAudioChannel
<<CV_OperationInterfaceSignature>>
IRegistry_Signature
<<CV_AnnouncementSignature>>
addAudioStreamer(channelId, channelName)
«CV_CompInterfaceTemplate» type = stream objectRole = producer
<<use>>
<<realize>>
<<use>>
<<use>>
«CV_CompInterfaceTemplate» type = operation objectRole = client
«CV_CompInterfaceTemplate» toListener : IAudioStream
«CV_CompInterfaceTemplate» toServiceMgr : IRegistry
<<CV_CompObjectTemplate>>
BindingObject
«CV_CompInterfaceTemplate» toStreamer : IAudioStream
«CV_CompInterfaceTemplate» type = stream objectRole = consumer
«CV_CompInterfaceTemplate» type = stream objectRole = producer
<<CV_CompObjectTemplate>>
AudioStreamer
«CV_CompInterfaceTemplate» toBinding : IAudioStream

Figure 3. Computational templates for the audio stream application

from the information available about the computational template.

In UML, interfaces are non instantiable elements, and ports are instantiated and destroyed only once the components that comprise them are instantiated or destroyed. However, in ODP, *Template Rules* [9, Part 3 – 7.2.5.1] provide mechanisms that allow computational objects to instantiate, bind to or destroy computational interfaces. So, it means that these functionalities should be explicitly detailed by the modeler according to the specific requirements of each ODP system.

5.3 Behavior

For the case study, we need to specify different behavior aspects of the computational elements. So, activity, communication, interaction and sequence diagrams might be useful to represent both the internal actions of the computational objects and the interactions that occur among them. Our Profile provides stereotypes that should be applied to the UML messages modeled in the behavioral diagrams.

Computational templates may include the specification of behavior, too. State machines (including the protocol state machines attached to the ports that represent the computational stream interface templates) provide the natural mechanism for modeling the state changes caused by events in computational objects. For example, Figure 5 shows the statechart that models the basic behavior of an AudioStreamer computational object. Note that not all the events and calls specified are represented by the computational interfaces at which this object interacts. This is because the interface signature just specifies public features of the computational object. In UML, an ODP internal action (e.g., new(Frame) in Figure 5) can be represented as a functionality of any internal part of the component. In addition, the *Structuring Rules* in the ODP Standard already provide a set of common functionalities inherent to all computational objects (e.g., bind in Figure 5).

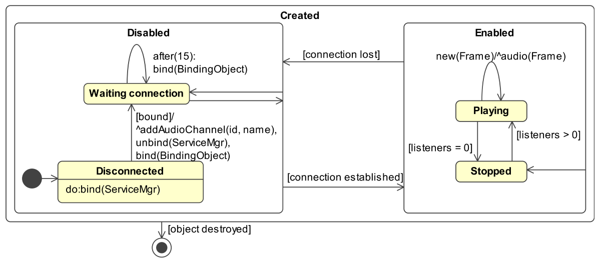


Figure 5. AudioStreamer statechart

5.4 Environment contracts

As previously mentioned in Section 4.3, environment contracts place constraints on the behavior of computational objects. These constraints usually refer to some QoS aspects, which are particularly relevant for multimedia applications. Note that the application of several profiles is possible if the rules and constraints imposed by them are compatible. Thus, the application of the profile presented here and the use of UML allows us to specify these constraints in, at least, two different ways:

(a) We can use the normal mechanisms provided by the UML to specify, for example, certain time aspects such as the duration intervals between messages in the interaction diagrams. These diagrams also allow the specification of conditions and alternative behaviors, if required.

(b) We can use specific UML Profiles for representing particular constraints. For example, the required *communication throughput* of an audio streamer object could be specified using the constructs provided by an external profile, such as the UML Profile for modeling QoS characteristics, which seems to be well suited for multimedia domains.

6 Some issues for discussion

UML is a notation general enough to model most kinds of object-oriented systems. However, it presents some semantic differences with ODP that may cause problems in some cases. This section discusses the issues that we have discovered when modeling the ODP computational language using the UML 2.0 notation.

6.1 General issues

The major issues come from the differences between the underlying object-oriented models of UML and ODP.

The traditional UML object model assumes a single hierarchy of subclasses of isolated objects exchanging messages, in which classes are first-class citizens, and objects are just instances of classes that own attributes (to hold the objects' state), operations, and invariants. By contrast, a more general object model, such as the one followed by ODP, does not require invariants and operations to be owned by a single object; rather, it uses *collective state* for invariants, and *collective behavior* for operation and interaction specifications [12]. For example, the interaction model of the UML is based on message exchange between objects, whereas interactions in the enterprise viewpoint are pieces of shared behaviour. Furthermore, ODP types are predicates on the objects, and classes are just collections of objects, promoting objects as first-class citizens.

These are subtle differences, but they raise interesting issues when trying to model some configurations of ODP objects or ODP interactions in UML, e.g., ODP synchronous interactions that simultaneously involve more than one object; or the automatic reclassification of objects, which may be required under some particular circumstances (e.g., it may be needed for representing systems in dynamically configurable networks).

6.2 Terminology conflicts

Another issue may arise from the different meaning assigned to some common terms in UML and ODP, that can be the cause of confusion between UML and ODP modelers. Examples include the previously mentioned terms, class and type. UML classes (and ≪types≫) are mapped in our proposal to ODP types; UML concrete classes are mapped to ODP templates; there is however no UML concept directly related to the concept of ODP class (an ODP class is a collection of objects that satisfy a given type).

In the computational language, the main problem may be related to the concept of interface. An ODP interface is a collection of interactions, i.e., it sits at the instance level. However, UML interfaces "specify" a set of public features and obligations, and therefore sit at the model level (a component implements an interface, i.e., an implementation relationship between a component and an interface implies that the component supports the set of features owned by the interface). This is why in our profile, UML interfaces correspond to ODP interface signatures.

6.3 Interface instantiation

Other kind of problems has to do with the differences between the rules that constrain the instantiation of the ODP computational interfaces, and the rules that govern the instantiation of their corresponding ports and interfaces in UML.

The ODP structuring rules [9, Part 3 – 7.2.5.1] establish that any computational object can dynamically instantiate interface templates. This would translate in UML into the capability of a component instance to dynamically instantiate ports. However, the semantics of UML prescribe that if a component classifier instance is created, then the instances of each of its contained ports are also created. This implies that ports can not be created or destroyed except as part of the creation or destruction process of the owning component.

This issue is not very critical, though, because a UML component is a composite structure, and UML allows to dynamically create, destroy and assign the objects (or other internal component instances) that implement the services provided by the port.

6.4 No UML interfaces for two-way interactions

In the ODP computational viewpoint, signals may be the natural way for expressing most of the computational object interactions, including operations and flows. This is specially relevant for, e.g., multimedia domains. Flows can be defined in terms of signals and, as described in [9, Part 1–7.2.2.5], signal interfaces can be used as a basis for explaining QoS characteristics, compound binding between different kinds of interfaces, etc.

The problem that we face here is an issue of granularity, due to the fact that signals with different causalities can coexist in an ODP interface, whilst UML requires grouping incoming and outgoing messages in different UML interfaces. In our proposal, ODP interface templates are modeled by UML ports, which allows UML messages representing ODP signals of different causalities to be grouped under the same UML port. However, these UML messages need to be declared in UML interfaces, and this is where we may get a very unnatural decomposition: provided interfaces will represent the incoming interactions and required interfaces will specify the outgoing ones.

This has two main consequences. First, if several UML interfaces are needed to model just a single ODP computational signature (some for the outgoing signals, some others for the incoming ones), then several UML assembly connectors will be required to express the *same* binding between two interacting computational objects through their computational interfaces. This implies both stronger constraints on the elements defined in the ODP profile, and an unnatural grouping of messages. Secondly, think for instance of a exchange of information (i.e., an interaction) based on request and reply signals. In this case, the separation of the UML messages that represent these ODP signals in two different UML interfaces (one provided, and one required) seems to be quite artificial and confusing, apart from losing some information, e.g., how the requests and the replies are related.

7 Conclusions

Most of the existing proposals for modeling the ODP viewpoint languages have usually used formal description techniques, that have proved valuable in supporting the precise definition of reference model concepts [4, 11, 12]. Among all the works, probably the most widely accepted notations for formalizing the computational viewpoint are Z, LOTOS, and SDL. Lately, rewriting logic and Maude have also shown their adequacy for modeling the ODP languages [6, 5, 23]. However, the lack of acceptance of formal notations in industrial and commercial environments has encouraged the quest for graphical and more appealing notations for modeling the ODP viewpoint concepts.

In particular, UML 1.x has already been proposed by different authors for ODP computational modeling. It has an appealing graphical syntax and wide acceptance within the software engineering community. However, its loose semantics and lack of elements for modeling many of the specific concepts of ODP has traditionally represented an impediment for achieving the precise specification and analysis of systems. This issue has been addressed by different authors 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 [17], whose Component Collaboration Architecture (CCA) provides a set of elements and mechanisms well suited to write ODP computational specifications. However, the size and complexity of EDOC represents, from our point of view, an important limitation for its wide acceptance by the software engineering community.

Another interesting and complete proposal [1] uses UML to address computational viewpoint designs, complementing the UML diagrams with the Component Quality Modeling Language (CQML) for expressing environment contracts constraints. However, this approach, which was specially designed for multimedia distributed systems, uses UML version 1.4, so it does not take advantage of the new concepts and mechanisms mentioned above. Besides, although this proposal tries to model the computational viewpoint language, it seems to present some semantics differences with the ODP Standard (e.g., there is no distinction between interface templates and signatures, and therefore they are treated equally).

Apart from validating our proposal by specifying more kinds of examples, there are some lines of work that we plan to address shortly. In particular, once we count with a graphical notation to model the ODP computational viewpoint, we perceive that its connection to formal notations and tools might bring along many real advantages. For instance, formal analysis of the system can be achieved from the UML environment (such as model checking, theorem proving, etc.), freeing the system analyst from most formal technicalities. In this sense, we plan to provide bridges between the UML 2.0 specification and the Maude language, so that the Maude formal toolkit can be used with the UML models produced for the ODP system.

Acknowledgements The authors would like to thank the anonymous referees for their helpful comments and remarks. This work has been supported by Spanish Research 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] 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.

[3] C. Bock. UML 2 activity and action models part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

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

[5] F. Durán, M. Roldán, and A. Vallecillo. Using Maude to write and execute ODP Information Viewpoint specifications. *Computer Standards & Interfaces*, 2005.

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

[7] L. Fuentes and A. Vallecillo. An introduction to UML profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13, Apr. 2004.

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

[9] 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.

[10] ISO/IEC. *Information technology – Open distributed processing – Use of UML for ODP system specifications*. International Standards Organization, Geneva, Switzerland, 2005. ISO/IEC CD 19793, ITU-T Recommendation X.906.

[11] D. R. Johnson and H. Kilov. Can a flat notation be used to specify an OO system: using Z to describe RM-ODP constructs. In E. Najm and J.-B. Stefani, editors, *Proc. of FMOODS'96*, pages 407–418, Paris, Mar. 1996. Chapman & Hall.

[12] D. R. Johnson and H. Kilov. An approach to a Z toolkit for the Reference Model of Open Distributed Processing. *Computer Standards & Interfaces*, 21(5):393–402, Dec. 1999.

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

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

[15] P. F. Linington. What foundations does the RM-ODP need? In *Proc. of the 1st International Workshop on ODP in the Enterprise Computing (WODPEC)*, pages 17–22, Monterey, California, Sept. 2004.

[16] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, September 2004. OMG document ptc/04-09-01.

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

[18] OMG. *OCL 2.0*, Oct. 2003. Final Adopted Specification ptc/03-10-04.

[19] OMG. *Unified Modeling Language Specification (version 2.0): Infrastructure*, 2003. OMG document ptc/03-12-01.

[20] OMG. *Unified Modeling Language Specification (version 2.0): Superstructure*, 2003. Draft Adopted Specification ptc/03-08-02.

[21] J. R. Romero and A. Vallecillo. UML 2.0 Profile for the ODP Computational Viewpoint. Technical Report TR-05-03, Universidad de Málaga, Feb. 2005. Available from `http://www.lcc.uma.es/~jrromero`

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

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

[24] 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.

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

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