Advance Access publication on September 12, 2007 doi:10.1093/comjnl/bxm062

Modeling ODP Computational Specifications Using UML

JOSÉ RAÚL ROMERO[1], JOSÉ M. TROYA[2] AND ANTONIO VALLECILLO[2,*]

[1]*Department of Informática y Análisis Numérico, University of Córdoba, Spain*
[2]*Department of Lenguajes y Ciencias de la Computación, University of Málaga, Spain*
**Corresponding author: av@lcc.uma.es*

The open distributed processing (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. Quality of service (QoS) contracts and service level agreements are an integral part of any computational specification, which are specified in ODP in terms of environment contracts. Up until unified modeling language (UML) version 2, 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 the situation has 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 for modeling the ODP computational specifications and, in particular, we show how ODP environment contracts can be modeled with this approach.

Keywords: RM-ODP; computational specifications; environment contracts; QoS; UML

Received 19 July 2006; revised 2 July 2007

1. INTRODUCTION

We are witnessing an increasing interest in the Software Engineering community toward the use of models for developing software systems. Models allow to state features and properties of systems accurately, at the right level of abstraction, and without delving into the implementation details. Shifting intellectual property and business logic from source code into models allows organizations to focus on the important aspects of their systems, which have traditionally been blurred by the usage of standard programming languages and underlying technologies. Model engineering is an emergent discipline that considers models as first-class entities enabling new possibilities for creating, analyzing and manipulating systems through various types of tools and languages. Each model usually addresses one concern, and the transformations between models provide a chain that enables the automated implementation of a system from its corresponding models.

Models are specially important in the case of large-scale heterogeneous distributed systems, which are inherently much more complex to design, specify, develop and maintain than classical, homogeneous, centralized systems. One way to cope with such complexity is by dividing the design activity according to several areas of concerns, each one focusing on a specific aspect of the system, as described in IEEE Std. 1471 [1]. Following this standard, current architectural practices for designing open distributed systems define several distinct viewpoints. Examples include the viewpoints described in the '4 + 1' view model [2], the Zachman's framework [3] or the reference model of open distributed processing (RM-ODP) [4].

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

organizations investigating RM-ODP as a promising alternative for specifying their IT systems, and for structuring their large-scale distributed software designs. Examples of such projects include the DASIBAO (Démarche d'Architecture des Systémes d'Information BAsée sur ODP) methodology for specifying IT systems, developed by EDF (Electricité de France) [5]; the Reference Architecture for Space Data Systems (RASDS), developed by the Consultative Committee for Space Data Systems (CCSDS, which includes space agencies such as NASA/JPL, JAXA, ESA, etc.) [6]; the projects developed by the Interoperability Technology Association for Information Processing (INTAP) in Japan (http://net.intap.or.jp/e/); or the Synapses project for enabling EU healthcare professionals to share patient records and medical data irrespective of the systems that hold them (https://www.cs.tcd.ie/synapses/public/). RM-ODP has also been successfully used for building financial systems (see, e.g. [7, 8]) and in government [9].

The RM-ODP provides five generic and complementary viewpoints on the system and its environment: *enterprise*, *information*, *computational*, *engineering* and *technology*. They allow different participants to observe a system from different perspectives [10]. 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 and a common foundation defining concepts used in all of them (composition, type, subtype, actions, etc.) provide the glue that binds them all together.

One of these viewpoints, the *computational viewpoint*, describes the functionality of the system and its environment through the decomposition of the system, in distribution transparent terms, by means of objects which interact with interfaces. More precisely, the ODP computational viewpoint focuses on the software architecture of the system, how the system services are implemented, and the service level agreements that govern such functional description. An integral part of the ODP computational specifications are the environment contracts, which determine the quality of service (QoS), usage and management constraints of an object and its environment, in terms of requirements placed on an object's environment for the correct behavior of the object; and constraints on the object behavior in a correct environment. The correct and complete specification of environment contracts is critical in many important distributed application domains, such as embedded systems, multimedia applications or e-commerce services and systems.

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. This lack of precise notations for expressing the different models involved in a multi-viewpoint specification of a system is a common feature for most enterprise architectural approaches, including the Zachman framework, the '4 + 1' model, or the RM-ODP. These approaches were consciously defined in a notation- and representation-neutral manner to increase their use and flexibility. However, this makes more difficult, among other things, the development of industrial tools for modeling the viewpoint specifications, the formal analysis of the specifications produced and the possible derivation of implementations from the system specifications (see, e.g. [11]).

Several notations have been proposed for the different ODP viewpoints by different authors, which seem to agree on the need to represent the semantics of the ODP viewpoints concepts in a precise manner [4, 10, 12, 13]. For example, formal description techniques such as Z and Object-Z have been proposed for the information and enterprise viewpoints [14], and LOTOS, *Specification and Description Language* (SDL) or Z for the computational viewpoint [4, 15].

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 as a modeling language for expressing ODP system specifications. Until the advent of UML version 2, 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 [16] 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, the situation seems to have changed, since not only its semantics have been more precisely defined, but also incorporates now a whole new set of concepts more apt for modeling the structure and behavior of open distributed systems. In particular, UML provides now more appropriate support for describing architectural components and connectors, structured classifiers, improved state machine specifications and sequence diagrams, and better mechanisms for defining domain-specific languages by means of UML profiles.

These improvements, together with the wide adoption of UML by industry, the number of available UML tools and the increasing interest for model-driven development and the model driven architecture (MDA) initiative, motivated both ISO/IEC and the ITU-T to start a joint project, launched in 2004, to define a standard for the use of UML for ODP system specifications [17]. This document (usually referred to as UML4ODP) defines a set of UML profiles, one for each viewpoint language and one to express the correspondences between viewpoints, by which ODP modelers can use the UML notation for expressing their ODP specifications in a standard graphical way, and UML modelers could use the

RM-ODP concepts and mechanisms to structure their large UML system specifications according to a mature and standard proposal.

The work presented in this paper has been developed within the scope of that project. More precisely, this paper is an extension of an initial work presented at EDOC 2005 [18], which provided the input to the UML4ODP standard to develop the representation of the computational viewpoint specifications using UML. However, both that initial work and the standard (in its current version) offer very limited support for expressing environment contracts. Thus, this paper not only describes the approach to express ODP computational specifications in UML, but also covers more in detail the specification of environment contracts. As mentioned earlier, these contracts are critical in most commercial systems and therefore deserve to count on the appropriate mechanisms for modeling them.

The structure of this document is as follows. First, Sections 2 and 3 serve as a brief introduction to the computational viewpoint and UML, respectively. Section 4 presents both the UML4ODP standard and our proposal to model environment contracts, describing how to model computational specifications with UML. This 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 to represent the ODP concepts, and Section 7 compares this work with other similar proposals. Finally, Section 8 draws some conclusions and outlines some future research activities.

2. COMPUTATIONAL VIEWPOINT IN RM-ODP

The computational viewpoint is directly concerned with the functional aspects of the system, independently from their distribution. 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 ODP object model, which defines: the form of interfaces that objects 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.

2.1. Computational language concepts

2.1.1. 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 [4].

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 or synchronize two or more interfaces, e.g. a binding object may be responsible for ensuring that a certain level of QoS is maintained between interacting objects.

2.1.2. Computational templates

Computational objects and interfaces can be specified by templates. In ODP, an $<$X$>$ *template* is 'the specification of the common features of a collection of $<$X$>$ s in sufficient detail that an $<$X$>$ can be instantiated using it'. $<$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 a set of signal signatures, operation signatures or flow signatures as appropriate. Each of these signatures specifies the name of the interaction and its parameters (names and types).

2.1.3. Interactions

RM-ODP prescribes three particular types of interactions: *signals*, *operations* and *flows*. A signal may be regarded as a single, atomic action between computational objects and constitutes 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 types: *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, the server object does not return a termination. Terminations model every possible outcome of an operation.

Flows model streams of information, i.e. a flow is an abstraction of a sequence of interactions, resulting in conveyance of information from a producer object to a consumer object. A flow may be used to abstract over, for example, the exact structure of a sequence of interactions, or over a continuous interaction including the special case of an analog information flow. In general, a flow is seen in the

computational view as essentially unstructured, with the data transfer process seen as a single stream. The structure of the flow is normally described in the engineering view.

2.1.4. Environment contracts

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

2.2. Structure of ODP computational specifications

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

A computational specification defines as well 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, or delete computational interfaces or computational objects.

3. UML 2

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 one shows the system features that do not change with time. 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).

UML version 2 [19] counts on new diagrams (e.g. composite, communication, timing and interaction overview diagrams) and has enhanced some of the UML 1.X ones (e.g. sequence diagrams). Some of the UML version 2 improvements have been influenced by the integration of the mature SDL language within UML. In addition, UML 2 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 2 (*Object Constraint Language*), which claims to be aligned with UML 2 [20]. Finally, the language extension mechanisms have greatly been 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 version 2 constitute the basis of this proposal.

4. MODELING COMPUTATIONAL VIEWPOINT CONCEPTS IN UML 2

The UML Profile for the ODP computational viewpoint, as described in the UML4ODP standard [17], is made up of three main parts. First, it defines the ODP computational viewpoint metamodel as well as the semantics, properties and related elements of each metaclass. Second, ODP concepts are expressed as 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. Finally, a set of OCL constraints (on the stereotyped elements) captures the structuring rules that should be followed by those models that claim conformance to the ODP computational specifications. Note that the description of the metamodel is not strictly necessary to define the profile, because metamodels and profiles are different ways of defining domain-specific languages. However, thinking about the metamodel greatly helped to define the ODP concepts and to understand the existing relationships among them, as a previous step to map them to UML concepts.

This section summarizes how the main concepts of the ODP computational language are expressed in terms of the UML 2 concepts. The interested readers can consult [17, 18] for a more detailed description of these mappings.

It is important to notice that the mappings between the ODP and the UML elements are not always straightforward. In fact, the use of UML for ODP system modeling is not free from problems, as we shall discuss later in Section 6. For instance, the object models followed by UML and ODP do not match completely (e.g. UML is class-based, whereas ODP is object-based; their behavioral models are different; etc.), and there are some ODP concepts that do not have a direct mapping onto UML (e.g. there is no single UML element that can naturally represent an ODP *interface signature*). In some cases, we had to find compromise solutions between the definition given by the computational language for some ODP concepts and the UML semantics for the corresponding notational constructs. In such cases, our priority was always focused on improving the

usability, the simplicity and the readability of specifications produced, while respecting the semantics of both the ODP computational language and the UML.

In the following, when there is some possibility of confusion between the ODP and the UML concepts, we will distinguish them by writing ODP concepts in *italic typeface* and UML concepts in sans-serif typeface.

4.1. Computational objects and interfaces

A key concept of the ODP computational viewpoint is the *computational object*. An ODP *computational object* is generally specified in terms of its template, which is expressed by a UML component stereotyped as ≪CV_Object≫. 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*.

ODP *computational objects* are then expressed as UML instance specifications of the UML components that represent their *templates*.

4.1.1. 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 to provide one-to-one mappings for these ODP concepts. However, the semantics provided by other modeling elements can be used. 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.' 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 specification, the required interfaces realized by a port characterize the services that the owning component requires from its environment. Similarly, the provided interfaces characterize the behavioral features that the component offers to its environment at this interaction point. A behavioral port may be used in order to specify some behaviors (e.g. a protocol state machine) associated to some interfaces.

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* is expressed as a set of UML interfaces (see also 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 specification 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* are expressed as interfaces stereotyped ≪CV_SignalInterfaceSignature≫, ≪CV_OperationInterface Signature≫ and ≪CV_StreamInterfaceSignature≫, respectively.

4.1.2. 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. As a special kind of *computational objects*, *binding objects* are specified by component instance specifications stereotyped ≪CV_BindingObject≫ (this stereotype inherits from ≪CV_Object≫).

According to the structuring rules of the ODP computational viewpoint [4, Part 3–7.2.3], there are two different kinds of explicit *bindings* among *computational objects*: primitive and compound bindings. *Primitive bindings* communicate two *objects* directly through their *interfaces*, and are expressed in UML in terms of assembly connectors, stereotyped ≪CV_PrimitiveBinding≫, between the ports of the components that represent the interacting *computational objects. Compound bindings* link two or more *computational objects* via a *binding object*, and are therefore expressed in terms of the component that represent the *binding object*, which is connected to the interacting *objects* using *primitive bindings*.

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. Both synchronous and asynchronous interactions are possible in UML and in ODP [21].

An ODP *signal* will be expressed as 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 a UML operation.

The interactions that comprise ODP *operations* are also expressed in terms of UML messages, which represent the corresponding *announcements*, *invocations* or *terminations*.

Please note that, according to the interaction rules [4, Part 2–7.2.2.5], at a lower level of abstraction, *operations* can also 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*). Similarly, *terminations* are modeled by other two *signals*, the one that is sent by the server (the *termination submit*), and the one that finally reaches the client (the *termination deliver*). This way of modeling *operations* is not contemplated here. However, should there be the need to use it, the *signals* that comprise the specification of an *operation* can naturally be expressed in UML in terms of UML MessageEvents (from Communications).

An ODP *flow* is specified by a UML property, stereotyped as ≪CV_Flow≫. The property shall belong to a UML interface stereotyped as ≪CV_StreamInterfaceSignature≫, which represents the *stream interface signature* where the *flow* is defined. The name of the property expresses the name of the *flow*. The type of the property expresses the *flow signature*, which shall be expressed by a UML interface, stereotyped as ≪CV_FlowSignature≫. The *causality* of the *flow* (*consumer* or *producer*) is expressed by the tag definition, causality, of stereotype ≪CV_Flow≫.

A *computational signature* can be expressed by a UML reception, a UML operation or interface, depending on the sort of *signature*. UML receptions are used to express signatures of computational *interactions*, which are expressed by individual signals (*signals*, *announcements*, *invocations* and *terminations*). UML operations can be used to express ODP *interrogation signatures* that are composed of an *invocation signature* and a *termination signature*. Finally, UML interfaces are used for expressing *flow signatures*.

4.3. Environment contracts

An *environment contract* is a contract between an object and its environment. Unlike contracts in other ODP viewpoints, mainly in the enterprise and information viewpoints, which govern part of the collective behavior of sets of objects, *environment contracts* place constraints on the behavior of individual computational objects, and usually include QoS, usage and management aspects. QoS aspects are mainly focused on three main issues: time (e.g. latency), volume (e.g. throughput) and reliability (e.g. percentage of media frames lost).

In the UML4ODP standard, an *environment contract* of a *computational object* is simply expressed by a set of UML constraints (stereotyped ≪CV_EnvironmentContract≫) applied to the component that expresses the *computational object*. Although this is correct, we feel that it cannot be expressive enough to represent, in a standard manner, many of QoS constraints required in the environment contracts of, let us say, a complex multimedia application.

In general, the ODP reference model does not prescribe how *environment contracts* must be specified. This is why there are not many works that deal with the modeling of ODP environment contracts, even less in UML. Moreover, each system modeler might like to specify the contract constraints in the way that best suits his/her particular application, and therefore the UML elements (and their semantics) required to model different environment contracts can change from one application to another.

Our proposal is based on reusing existing UML mechanisms and notations for modeling QoS and other environment contract constraints, on top of the UML4ODP profiles. The possibility offered by UML 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/her preference.

In general, two standard profiles are currently the most widespread and used specifications concerning QoS aspects: the UML Profile for Schedulability, Performance and Time Specifications [22], and the UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms [23]. Both documents are OMG standards and propose a set of constructs that provide support to describe the main QoS elements in their respective domain. The first one is mainly focused on time-related properties, and allows the specification of performance and schedulability requirements. The UML profile for QoS was specially conceived for large distributed systems, and allows the description of QoS requirements and properties, including extra-functional requirements in analysis models (delay, jitter, throughput, etc.), risk assessment concepts and high-reliability requirements. The core concepts and general model of both UML profiles are inspired on the ISO general QoS architecture [24], that defines the fundamental QoS terms, concepts and mechanisms. In this paper, we will use the OMG's UML Profile for QoS to illustrate how an additional profile can perfectly work to express environment contracts and, particularly, QoS constraints on the computational elements.

The profile uses *QoS characteristics* to define collections of QoS concepts with precise meaning. Specifically, a QoS characteristic is a quantifiable non-functional aspect within a certain domain of values. QoS characteristics are defined independently of the means in which they are represented or controlled. Since a specification of a QoS characteristic is a constraint over its domain, these QoS characteristics are used to model the QoS requirements by identifying restrictions on a certain element or service with respect to the value domain. QoS characteristics also include a set of *quality attributes* that are the *dimensions* in which to express a quality satisfaction. Then, *quality levels* serve to express the quantifiable level of satisfaction of a non-functional property.

Sometimes, specifications need to support different models of execution with different *QoS levels*. Each mode is executed

depending on different *QoS constraints* associated, and may exhibit different functionality. Thus, a QoS constraint define any kind of restriction that an element or service impose on QoS characteristics. Changes between levels are described in terms of *QoS transitions*. Three different kinds of QoS constraints can be distinguished: (i) those that define the level of quality of the services that must be provided to achieve the clients' requirements (*QoS required*); (ii) those that define the set of services provided, and establish the limits for supported values (*QoS offered*); and (iii) the agreements between the required and provided QoS constraints, for both the provider and the client (*QoS contract*).

Since a QoS constraint is expressed by a stereotyped UML constraint, this could also be compatible with the use of the stereotype ≪CV_EnvironmentContract≫. If needed, we could even import the profile and extend it so that any QoS constraint, or QoS level and transition might be depicted as a computational *environment contract*. However, this is not strictly necessary because any non-functional information reflected in the model can be considered as an *environment contract* [4, Part 2–11.2.3].

4.4. Structure of computational specifications

As mentioned in Section 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 computational specification is composed of a configuration of *computational objects*, which need to be instantiated from their corresponding computational templates. These templates are modeled using component diagrams, which comprise the notational elements for representing at least the *object templates*, *interfaces templates*, *interface* and *interaction signatures* and *bindings*.

A computational specification also comprises the specification of the *internal actions* for those *objects*, which will be modeled using behavioral models associated to the UML components that represent those *objects*'. In particular, two kinds of UML diagrams will be used to model the *objects* behavior: activity diagrams, which focus on the sequence and conditions for coordinating low-level behaviors; and statecharts, which show how events cause changes in the *objects' states*.

After providing the *computational objects*' structure and individual behavior, the *interactions* that occur between the objects can be modeled using interaction diagrams. These diagrams emphasize *object* interactions by describing how messages are passed between objects and cause invocations of other behaviors. Interaction diagrams come in different types. Sequence diagrams are used to model message interchanges between a number of lifelines, each of which represents the interacting ODP *interface instances*. They also allow to specify fragments of interactions that can be reused to specify more complex interactions, using options, alternatives, exceptions and other sequence composition operations. 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 [19]. They define interactions in a way that facilitates the overview of the control flow within system. Each node within the diagram can represent another interaction diagram.

Finally, the specification of the *objects*' *environment contracts* complete the computational specification of the system, using the mechanisms described in the previous section. UML 2 also offers new diagrams, which allow the representation of some QoS constraints in a more natural way. In particular, timing diagrams can also be 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 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. Note that profiles do not allow modifying existing metamodels. Rather, a profile is intended to provide a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain.

Table 1 shows a summary of the stereotypes defined in the UML Profile for the ODP computational viewpoint. Three important concepts are described in this table: (i) the ODP computational language concept; (ii) the UML base element that should be used to model that concept; and (iii) the name of the stereotype that should be applied to the UML element. The tag definitions and constraints that complete the definition of the profile have been omitted, but the interested readers can found them in [17].

5. A CASE STUDY

Let us show in this section how the UML Profile for the ODP computational viewpoint can be used, and how some QoS constraints can be modeled. We will illustrate it using a simple example of a typical 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

TABLE 1. Summary of the ODP computational viewpoint profile

ODP concept	UML base element	Stereotype
Computational object template	Component (from PackagingComponents)	≪CV_Object≫
Computational interface template	Port (from Ports)	≪CV_Interface≫
Signal interface signature	Interface(s) (from Interfaces)	≪CV_SignalInterfaceSignature≫
Operation interface signature	Interface(s) (from Interfaces)	≪CV_OperationInterfaceSignature≫
Stream interface signature	Interface(s) (from Interfaces)	≪CV_StreamInterfaceSignature≫
Interrogation signature	Operation (from Communications)	≪CV_InterrogationSignature≫
Announcement signature	Reception (from Communications)	≪CV_AnnouncementSignature≫
Invocation signature	Reception (from Communications)	≪CV_InvocationSignature≫
Termination signature	Reception (from Communications)	≪CV_TerminationSignature≫
Signal signature	Reception (from Communications)	≪CV_SignalSignature≫
Flow signature	Interface (from Interfaces)	≪CV_FlowSignature≫
Flow type	Property (from Kernel)	≪CV_Flow≫
Computational object	InstanceSpecification (from Kernel)	≪CV_Object≫
Binding objects	InstanceSpecification (from Kernel)	≪CV_BindingObject≫
Signal interface	Port (interaction point) (from Ports)	≪CV_SignalInterface≫
Operation interface	Port (interaction point) (from Ports)	≪CV_OperationInterface≫
Stream interface	Port (interaction point) (from Ports)	≪CV_StreamInterface≫
Signal	Message (from BasicInteractions)	≪CV_Signal≫
Announcement	Message (from BasicInteractions)	≪CV_Announcement≫
Invocation	Message (from BasicInteractions)	≪CV_Invocation≫
Termination	Message (from BasicInteractions)	≪CV_Termination≫
Primitive binding	Connector (from BasicComponents)	≪CV_PrimitiveBinding≫
Environment contract	Constraint (from Kernel)	≪CV_EnvironmentContract≫

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 Fig. 1.

We also need to specify the *environment contracts*. In this case, suppose that the following QoS constraints apply to the system.

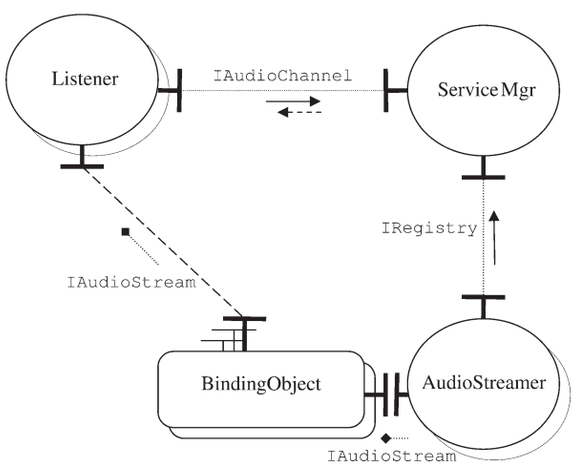

FIGURE 1. An audio stream application (informal spec).

(i) The response time from the service manager will always be <100 ms when processing the listeners' requests.
(ii) Listeners will be able to accept audio frames from the binding objects (i.e. the channels) at a rate of $\sim$232 bits/s.
(iii) The behavior of Listeners may be affected by their workload. Thus, two QoS levels will be distinguished for these objects, normal and overloaded, depending on the capacity of their internal buffer. Working in normal mode they will be able to process frames at a rate of 256 kbps, while this rate will drop to 128 kbps when overloaded, i.e. when the size of their buffer exceeds a certain threshold.

5.1. Computational templates

From the snapshot in Fig. 1, we can identify several *computational object templates* and *computational interface templates*.

More specifically, four *object templates* can be identified: Listener, ServiceMgr, AudioStreamer and Binding. These all are represented by stereotyped UML components as shown in Fig. 2. Three *computational interface templates* are identified too. In this case, UML stereotyped ports are used to express not only the different kind of *interface templates*, but also the

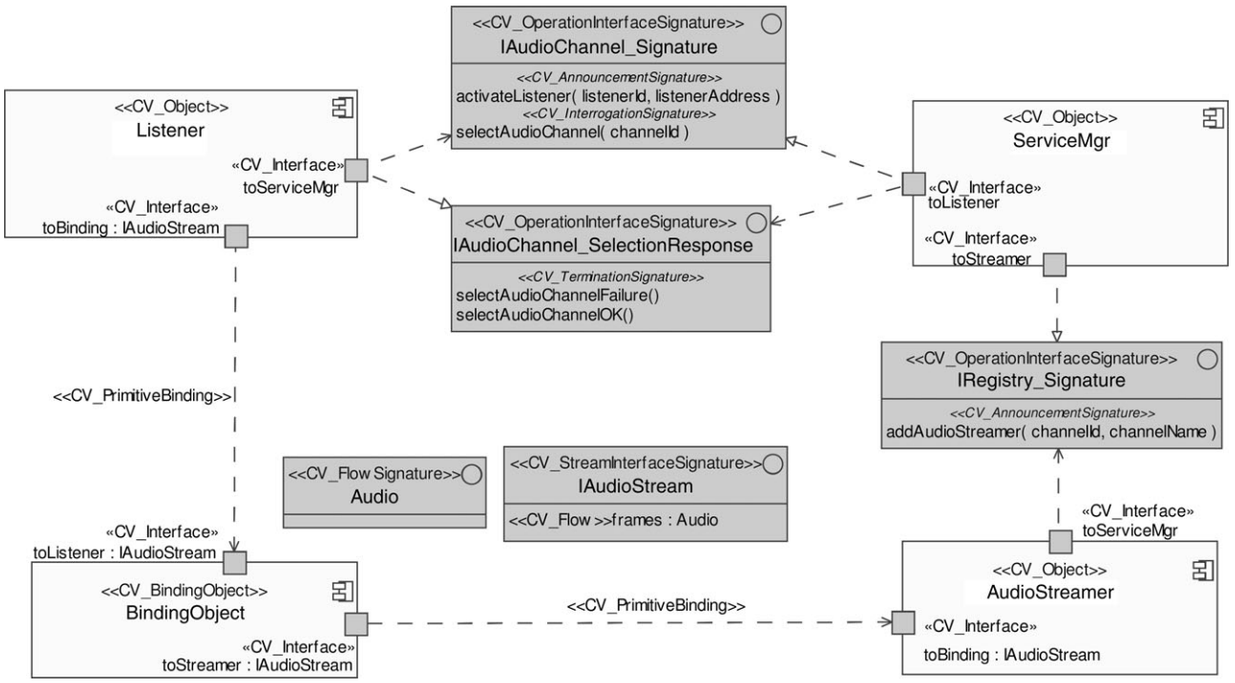

FIGURE 2. Computational templates for the audio stream application.

object templates of which these *interface templates* are part. More specifically, these *interfaces templates* are: IAudioChannel, IRegistry and IAudioStream. The first two are *operation interfaces* and the last one is a template for *stream interfaces*. (For clarity, the information about the *role* and *causality* of these ODP *interfaces* has been omitted in the diagram.)

UML interfaces shown in Fig. 2 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≫—declares the signatures for both the activateListener *announcement* and the selectAudioChannel *invocation*. This port (which represents an ODP *interface signature*) has another UML interface, IAudioChannel_SelectionResponse, which models the corresponding *terminations*. In this example, we have decided to model the *interrogations* using separate *invocations* and *terminations*. Alternatively, they could have been modeled using UML operations that jointly represent the corresponding ODP *interrogations*.

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*. Fig. 3 shows a configuration consisting of two audio streamers (i.e. 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 UML component that represents 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

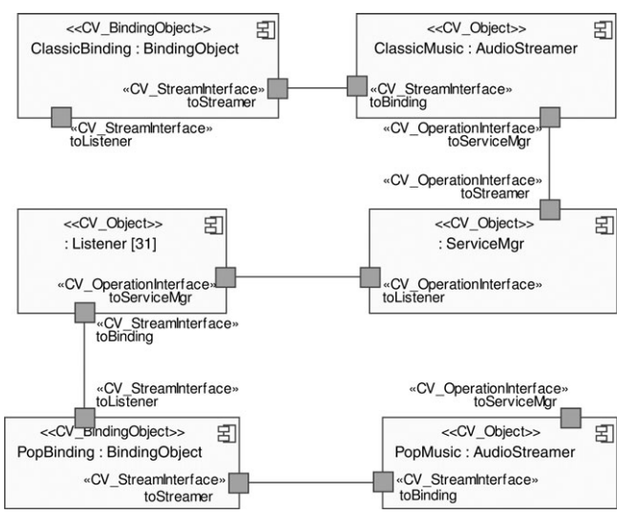

FIGURE 3. Computational object configuration.

explicit the existence of such *computational objects* in the system (e.g. both audio streamers have a unique name in the model).

Computational interfaces are represented by ports attached to the corresponding component instances. Note that, at the object level, the stereotype itself indicates the kind of *interface* that this port represents. Link between ports represent the binding between *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 from the information available in 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 *computational objects* can instantiate, bind to or destroy individual *interfaces*. This issue will be discussed later in Section 6.3.

5.3. Behavior

Computational templates can include the specification of behavior. In this case, state machines (e.g. 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, Fig. 4 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*.

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.

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.

(i) We can use the normal UML mechanisms 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).
(ii) Alternatively, we can use specific UML profiles for representing particular constraints.

Each kind of constraints of an *environment contract* may require a different way of modeling.

In our example, we will show how the UML profile for QoS [23] can be used for modeling the two QoS constraints presented above, as explained in Section 4.3.

The first constraint establishes a limit of time between a listener's request and the corresponding service manager response, i.e. the serviced manager's turn-around time. We first select one of the QoS characteristics defined in the profile (in this case, turn-around) and specialize it to our specific circumstances by defining a new QoS characteristic (ServiceMgrTurnAround), which assigns the appropriate values to the template parameters, as shown in Fig. 5.

Once we have defined this QoS characteristic, we need to attach it to the appropriate model elements. In this case, as shown in Fig. 7, we attach the ≪QoSOffered≫ constraint to the corresponding IAudioChannel *computational interface template* of the *ServiceManager* component.

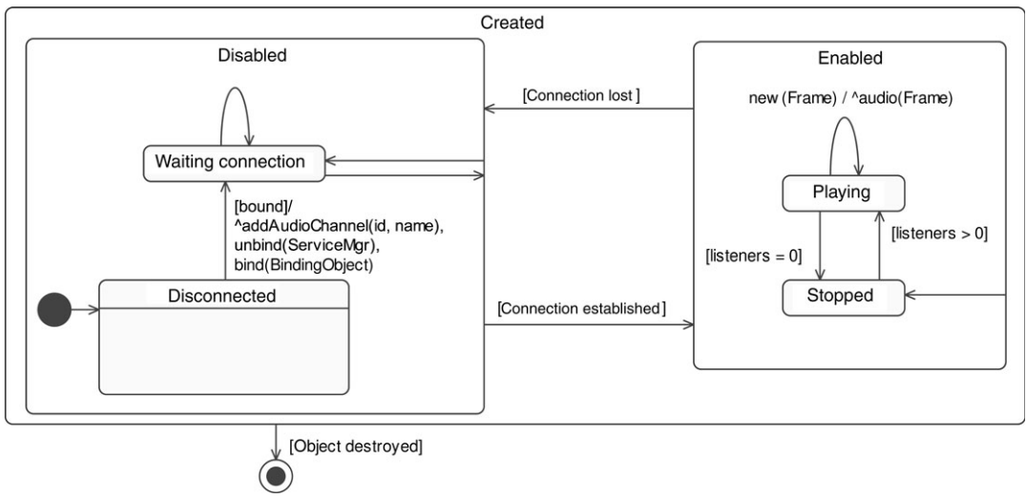

FIGURE 4. AudioStreamer statechart.

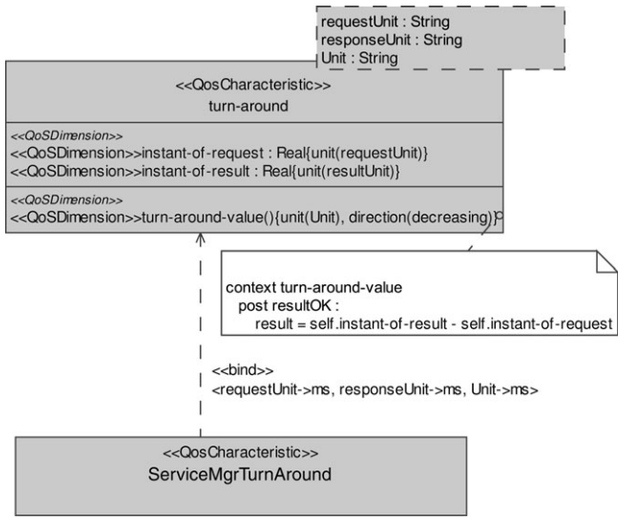

FIGURE 5. QoS characteristic: ServiceMgrTurnAround.

Similarly, we need to specify the minimum rate at which a listener is able to receive audio frames from the channel, i.e. the communication throughput. Then, we define our own MinStreamerThroughput characteristic and, as prescribed in [23], bind it to the communication-throughput QoS characteristic (see Fig. 6). Since that characteristic is not parameterized, no template parameters need to be considered.

This QoS characteristic is used by the ≪QoSRequired≫ constraint attached to the AudioStream *computational interface template* in the IAudioStream *computational object template*, as shown in Fig. 7. (Please notice that Figs. 2 and 7 are the same, apart from the QoS notes attached to some of their elements.)

Finally, Fig. 8 specifies two states for Listener objects, depending on the QoS level they are able to provide. Thus, depending on how full their buyer is, they toggle between the normal and overload states, as prescribed by the third of the QoS constraints defined for this example.

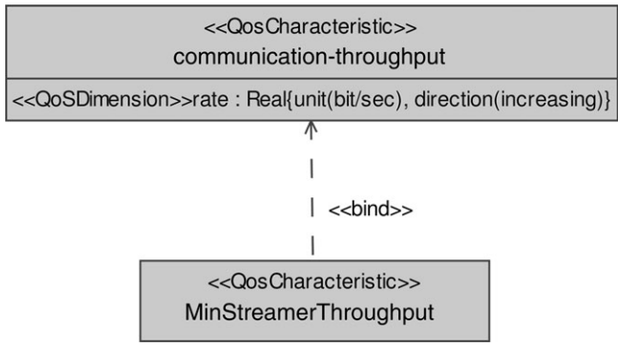

FIGURE 6. QoS charact.: MinStreamerThroughput.

6. FURTHER ISSUES

UML is a notation general enough to model most kinds of object-oriented systems. However, it presents some semantic differences with ODP, which need to be taken into account when modeling ODP viewpoint languages. This section discusses the issues that we have discovered when modeling the ODP computational language using the UML 2 notation. None of them are really critical, but they deserve some attention for the sake of precision.

6.1. General issues

The major issues come from the differences between the underlying object-oriented models of UML and the 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. In 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 [25]. For example, the interaction model of the UML is based on message exchange between objects, whereas interactions in the computational viewpoint are pieces of shared behavior. Furthermore, ODP object 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 some 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. However, this is probably a more critical issue in the enterprise or information viewpoints than in the computational viewpoint, in which we are normally concerned with the behavior of isolated interacting objects. Actually, both *environment* [4, Part 2–8.2] and *environment contract* [4, Part 2–11.2.3] are mainly defined for individual objects (although objects, at different abstraction level, may be composite).

This is why in our approach their representation is done for individual objects, too. Other representations are of course possible. In order to provide an alternative representation of environment contracts that refer to the collective behavior of objects in UML, it would be necessary to use class diagrams, at least to describe the ontology of the objects under consideration and of the relationships between these objects. Although these class diagrams may not, strictly speaking, be a fragment of the computational viewpoint, nevertheless, the computational specification ought to refer to the semantics of these diagrams: as in any specification, independently of the notation used, the ontology, i.e. the domain model, is essential for understanding the rest including the computational

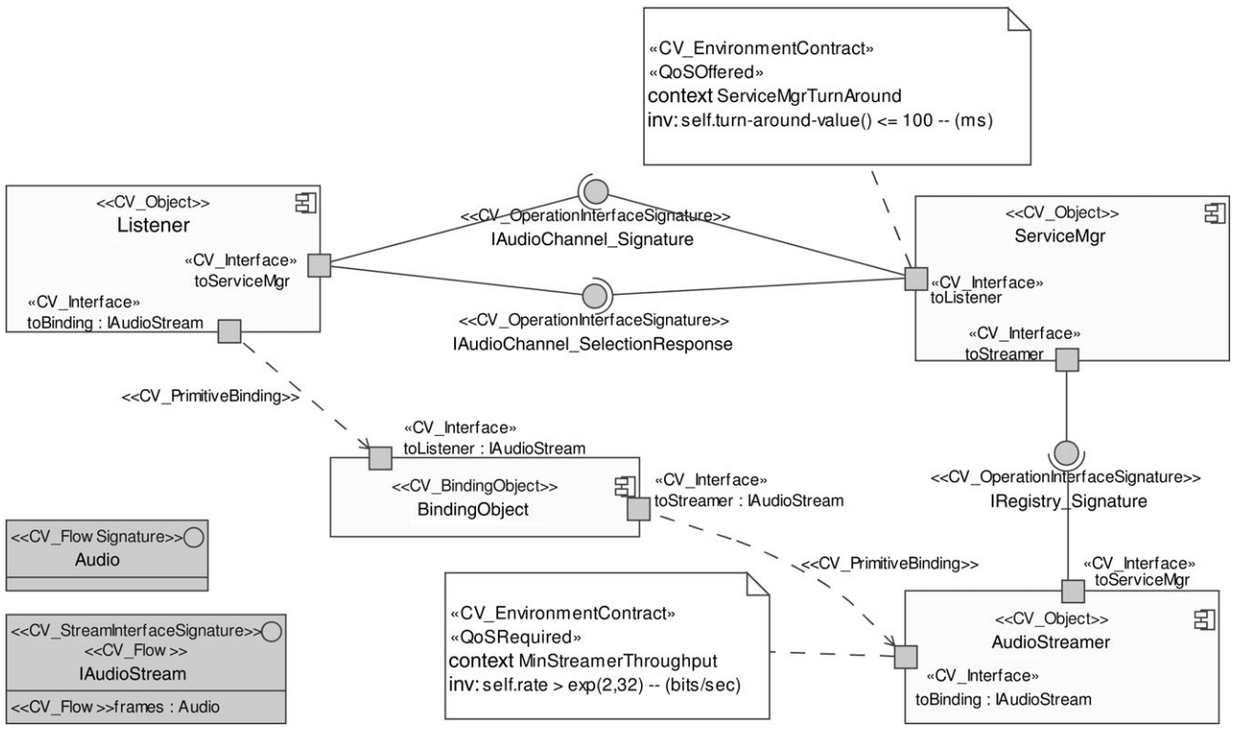

FIGURE 7. Computational template diagram with QoS constraints.

objects and their behavior. For examples of these alternative representations see, e.g. the UML Profile for EDOC (on Relationships) [16]; the book 'Business models' by Kilov [26]; or the third volume ('Domains, Requirements, and Software Design') of the book 'Software Engineering' by Bjorner [27].

6.2. Terminology Conflicts

Another issue may arise from the different meaning assigned to some common terms to UML and ODP. Examples include the previously mentioned terms, class and type. UML classes (and types) may correspond to ODP *types*; UML concrete classes or components may correspond to ODP *templates*, but there is no UML concept 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 they sit at the classifier level

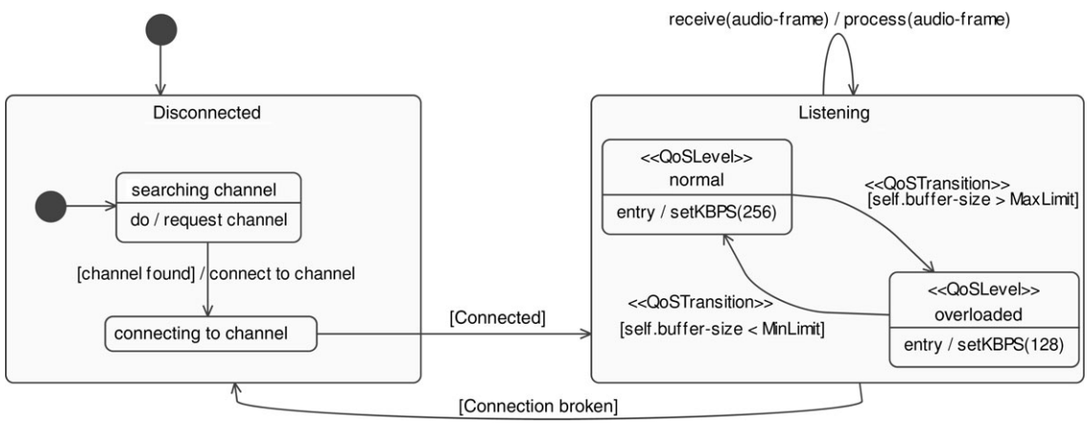

FIGURE 8. Statechart of the Listener with different states depending on the QoS level.

(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 the UML4ODP profile, UML interfaces express ODP *interface signatures*.

In addition, it is well known that RM-ODP substantially uses the concepts of abstraction levels and viewpoints, and while UML may be using viewpoints in a reasonable manner (although it is not always clear whether and how different diagrams interrelate and interoperate), it is certainly not clear how UML uses the concept of 'abstraction levels', which is essential for large and complex specifications.

6.3. Interface instantiation

Other problems are due to 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 [4, 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 cannot 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

This problem happens when modeling ODP *interface signatures* that comprise incoming and outgoing interactions. UML interfaces can only group interactions with the same causality, and therefore we may obtain a very unnatural decomposition of the same *interface signature* specification: provided interfaces will represent the *incoming interactions* and required interfaces will specify the *outgoing ones*.

But if several UML interfaces are required to model a single ODP *computational signature* (some for the outgoing interactions, some 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 (depending on their causalities).

This is a similar problem to that of modeling Web services descriptions with UML, because WSDL also allows operations with different causalities to co-exist within the same interface description.

We have tried to illustrate this issue in the example, Section 5.1, Fig. 2, by splitting the IAudioChannel *interface signature* into two separate UML interfaces, one with the *invocations* and other with the *terminations*. In most cases, the use of UML operations can help to overcome this problem. However, it is not so easy to solve when *interrogations* with different causalities need to co-exist in the same *interface*—as it happens, for instance, in the case of interactions that contain 'call-backs'.

6.5. UML semantic variation points

One of the problems for defining precise mappings between UML and ODP (or any other language) concepts is due to the UML semantic variation points, which allow for 'semantic specialization' of the UML concepts. Semantic variation points, as defined by UML, explicitly identify the areas where the semantics are intentionally underspecified to provide leeway for domain-specific refinements of the general UML semantics (e.g. using stereotypes and profiles). Examples include the order and way in which part instances of an aggregation are created, or the dispatching method by which a particular behavior is associated with a given message. These issues depend on the higher-level formalism used and are not defined in the UML specification.

In general, the presence of numerous variation points in the UML semantics (and the fact that they are defined informally using natural language), make it impractical to define formal mappings. In this approach, we decided that, in case of variation points of a UML concept representing an ODP concept, the semantics of the ODP concept should be used (this is one of the benefits of using ODP, which has more precise semantics). For instance, ODP defines several failure rules, that UML leaves underspecified (e.g. the behavior of an invocation of a UML operation when a precondition is not satisfied is a semantic variation point). Other example of semantic variation point in UML happens in the specification of the rules for the redefinition of operations in case of specializations, i.e. the rules regarding invariance, covariance or contravariance of types and preconditions, which determine whether the specialized classifier is substitutable for its more general parent. This is precisely defined in ODP, and therefore the ODP semantics should apply in this case.

6.6. Metamodeling choices

This final issue does not have to do with UML itself, but on how the RM-ODP foundational concepts (defined in Part 2) and the specific viewpoint language concepts have been structured. We initially had two choices: (i) define a single metamodel with the foundational concepts of RM-ODP (i.e. the core model), and then extensions of that metamodel with the

particular concepts and mechanisms of every viewpoint language; or (ii) define individual metamodels with all concepts related to every particular viewpoint language, and then establish correspondences between them.

We decided to go for the second option, i.e. to model each viewpoint language individually. This has proved to have some interesting benefits. For instance, viewpoint languages are self-contained. Furthermore, although the core elements are common to all viewpoints, most of them have some individual particularities in each viewpoint language, which required extensions in all viewpoint language metamodels, anyway. In addition, some concepts may have different representations in different viewpoint languages when represented in UML, e.g. in the UML4ODP proposal, objects in the Enterprise and Information viewpoints are specified in terms of UML classes, whereas Computational and Engineering objects are specified in terms of UML components, which are more appropriate UML elements for capturing the semantics of the corresponding ODP concepts. Of course, this option may present some limitations, too. For instance, the common structure underlying all viewpoint specifications may be lost (or at least may not be explicitly visible), so that viewpoint correspondences might be non-trivial to establish.

In any case, we think that the advantages outweigh the limitations, especially when the viewpoints are normally specified individually, and then ODP correspondences need to be defined between their elements anyway. As we have just mentioned, in this way the individuality of each viewpoint language can be respected, and the UML modeling elements more apt for representing each viewpoint concept can be decided in each case.

7. RELATED WORK

Most of the existing proposals for modeling the ODP viewpoint languages have usually used formal description techniques, which have proved valuable in supporting the precise definition of reference model concepts [13, 25, 28]. 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 [29, 30]. More specifically, in [31] we showed how Maude (a formal language based on rewriting logic) can be used to formalize the computational viewpoint. It has been shown that rewriting logic has very good properties as a logical framework, in which representing many different languages and logics, and as a semantic framework, in which giving semantics to them [32]. Thus, Maude seems to be a promising option as a unifying framework for the specification of environment contracts using different notations and logics. 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.

UML 1.X has also 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 [16], whose component collaboration architecture 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 by Akehurst *et al.* [33] uses UML to address computational viewpoint designs, complementing the UML diagrams with the component quality modeling language [34] 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 provided by UML 2. In addition, although this proposal tries to model the computational viewpoint language, it seems to present some semantic differences with the ODP standard (e.g. there is no distinction between interface templates and signatures, and therefore they are treated equally).

8. CONCLUSIONS

The increasing interest in model-driven engineering and in the MDA initiative for coping with the complexity of distributed systems has urged the need to count with precise models of the different aspects (or viewpoints) that constitute a system's specifications. The RM-ODP is also becoming increasingly relevant nowadays, mainly because the size and complexity of current IT systems is challenging most of the current software engineering methods and tools. Thus, RM-ODP is gaining recognition as an effective approach for specifying large-scale distributed system, and the industry is showing interest on it.

The work presented here provides a notation and a Profile for modeling the ODP computational concepts in UML 2, which aims at enabling model-driven support for specifying and implementing ODP systems. This paper is an extension of an initial work reported at EDOC 2005 [18] that now covers the environment contracts, which are an essential part of the computational specification of any system. Moreover,

this work is in line with the ITU-T and ISO/IEC joint standardization efforts to provide a UML representation of the ODP concepts. This proposal has been presented here using a rather small but illustrative example. Other examples, together the electronic versions of the metamodel and profile, are available from the RM-ODP web site (http://www. rm-odp.net).

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, its connection to formal notations and tools might bring along many advantages. For instance, formal analysis of the system can be achieved from the UML environment (such as model checking, theorem proving, etc.), leveraging the system analyst from most formal technicalities. In this sense, we are working on the provision of bridges between the UML specification and the Maude language, so that the Maude formal toolkit can be used with the UML models produced for the ODP system. Our first results in this respect are encouraging, and have already been reported in [11].

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their insightful and constructive comments and suggestions. 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. In particular, we would like to thank Peter Linington, Akira Tanaka and Bryan Wood for sharing their expertise and knowledge with us.

FUNDING

Ministerio de Educación y Ciencia de España (Spanish Research Project TIN2005-09405-02-01).

REFERENCES

[1] IEEE Std. 1471 (2000) Recommended practice for architectural description of software-intensive systems. *IEEE*. New York, USA.

[2] Kruchten, P. (1995) Architectural blueprints—the '4 + 1' view model of software architecture. *IEEE Softw.*, **12**, 42–50.

[3] Zachman, J.A. (1997) *The Zachman framework: a primer for enterprise engineering and manufacturing*. Zachman International, La Cañada, CA, USA. http://www.zifa.com.

[4] ISO/IEC 10746, ITU-T Rec. X.901-4 (1997) RM-ODP. Reference Model for Open Distributed Processing. ISO and ITU-T. http://www.rm-odp.net. Geneva, Switzerland.

[5] Picault, A., Bedu, P., Delliou, J.L., Perrin, J. and Traverson, B. (2004) Specifying information system architectures with DASIBAO—a standard-based method. *Proc. ICEIS 2004*, Porto, Portugal, April, 254–264. INSTICC Publications, France.

[6] CCSDS Red Book 311.0-R-1 (2007) *Reference architecture for space data systems (RASDS)*. Consultative Committee for Space Data Systems, http://public.ccsds.org/review/default.aspx. Reston, (VA), USA.

[7] Bernet, O. and Kilov, H. (2003) From Box-and-Line Drawings to Precise Specifications: Using RM-ODP and GRM to Specify Semantics. In Kilov, H. and Baclawski, K. (eds.), *Practical Foundations of Business System Specifications*, 99–110. Kluwer Academic Publishers, Norwell, MA.

[8] Kudrass, T. (2003) Describing Architectures Using RM-ODP. In Kilov, H. and Baclawski, K. (eds.), *Practical Foundations of Business System Specifications*, pp. 231–244. Kluwer Academic Publishers, Norwell, MA.

[9] Sweeney, L.E., Kortright, E.V. and Buckley, R.J. (2001) Developing an RM-ODP-based architecture for the defense integrated military human resources system. *Proc. WOODPECKER'2001*, Setubal, Portugal, July, 110–123. ICEIS'2001 ICEIS Press.

[10] Linington, P. (1995) RM-ODP: The Architecture. In Milosevic, K. and Armstrong, L. (eds.), *Open Distributed Processing II*, Brisbane, Australia, 15–33. Chapman & Hall, London.

[11] Romero, J.R. and Vallecillo, A. (2006) On the execution of ODP computational specifications. *Proc. 3rd Int. Workshop on ODP and Enterprise Computing (WODPEC 2006)*, Hong Kong, October, 33–44. IEEE Digital Library.

[12] Bernardeschi, C., Dustzadeh, J., Fantechi, A., Najm, E., Nimour, A. and Olsen, F. (1997) Transformations and Consistent Semantics for ODP Viewpoints. In Bowman, H. and Derrick, J. (eds.), *Proc. FMOODS'97*, Canterbury, England, July, pp. 371–386. Chapman & Hall, London.

[13] Bowman, H., Derrick, J., Linington, P. and Steen, M.W. (1995) FDTs for ODP. *Comput. Stand. Interfaces*, **17**, 457–479.

[14] Steen, M.W. and Derrick, J. (2000) ODP enterprise viewpoint specification. *Comput. Stand. Interfaces*, **22**, 165–189.

[15] Sinnot, R. and Turner, K.J. (1997) Specifying ODP computational objects in *Z*. In Najm, E. and Stefani, J.-B. (eds.), *Proc. FMOODS'96*, Paris, France, March, pp. 375–390. Chapman & Hall, London.

[16] OMG ad/2001-08-19 (2001) A UML profile for enterprise distributed object computing V1.0. Object Management Group, Needham, MA.

[17] ISO/IEC FCD 19793, ITU-T X.906 (2006) Information technology—open distributed processing—use of UML for ODP system specifications. International Standards Organization. Geneva, Switzerland.

[18] Romero, J.R. and Vallecillo, A. (2005) Modeling the ODP computational viewpoint with UML 2.0. *Proc. 9th IEEE Int. Enterprise Distributed Object Computing Conf. (EDOC 2005)*, Enschede, The Netherlands, September, pp. 169–180. IEEE CS Press, Los Alamitos, CA.

[19] OMG formal/07-02-05 (2007) Unified modeling language 2.1.1 superstructure specification. OMG, Needham, MA.

[20] OMG ptc/06-05-01 (2006) Object constraint language (*OCL*) 2.0. OMG. Needham, MA.

[21] Linington, P.F. (2004) What foundations does the RM-ODP need? *Proc. 1st Int. Workshop on ODP in the Enterprise Computing (WODPEC)*, Monterey, CA, September, pp. 17–22. IEEE Digital Library.

[22] OMG formal/05-01-02 (2005) UML profile for schedulability, performance, and time specification. OMG, Needham, MA.

[23] OMG ptc/04-09-01 (2004) UML profile for modeling quality of service and fault tolerance characteristics and mechanisms. OMG, Needham, MA.

[24] ISO/IEC 15935 (1998) Open distributed processing—reference model—quality of service. Geneva, Switzerland.

[25] Johnson, D.R. and Kilov, H. (1999) An approach to a Z toolkit for the reference model of open distributed processing. *Comput. Stand. Interfaces*, **21**, 393–402.

[26] Kilov, H. (2002) *Business Models*. Prentice-Hall, NY.

[27] Bjorner, D. (2006) *Software Engineering. Domains, Requirements, and Software Design*, Vol. III. Springer Verlag, Heildelberg.

[28] Johnson, D.R. and Kilov, H. (1996) Can a flat notation be used to specify an OO system: using Z to describe RM-ODP constructs. In Najm, E. and Stefani, J.-B. (eds.), *Proc. FMOODS'96*, Paris, Fance March, pp. 407–418. Chapman & Hall, London.

[29] Durán, F. and Vallecillo, A. (2003) Formalizing ODP enterprise specifications in Maude. *Comput. Stand. Interfaces*, **25**, 83–102.

[30] Durán, F., Roldán, M. and Vallecillo, A. (2005) Using Maude to write and execute ODP Information viewpoint specifications. *Comput Stand Interfaces*, **27**, 597–620.

[31] Romero, J.R., Durán, F. and Vallecillo, A. (2007) Writing and executing ODP computational viewpoint specifications using Maude. *Comput. Stand. Interfaces*, **29**, 481–498.

[32] Meseguer, J. (2000) Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In Smith, S. and Talcott, C. (eds.), *Proc. FMOODS 2000*, Stanford, CA, September, pp. 89–117. Kluwer Academic Publisher, Norwell, MA.

[33] Akehurst, D.H., Derrick, J. and Waters, A.G. (2003) Addressing computational viewpoint design. *Proc. 7th IEEE Int. Enterprise Distributed Object Computing Conf. (EDOC 2003)*, Brisbane, Australia, September, pp. 147–159. IEEE CS Press, Los Alamitos, CA.

[34] Aagedal, J. (2001) Quality of service support in development of distributed systems. PhD Thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, Norway.