OCL for the ODP Information Viewpoint Specifications

BOUHDADI MOHAMED, EL MAATI CHABBAR, ABDALLAH AQARROUT
Department of Mathematics and Computer Science
University Mohammed V Faculty of Sciences
B.P1014  Rabat Morocco

Abstract: - Now that the Reference Model for Open Distributed Processing (RM-ODP) has stabilized, several ODP standards must be defined. These standards include ODP functions and ODP transparencies. We address in this paper issues related to the ODP information viewpoint standards. Firstly, using the ODP information concepts and based on the trends in software engineering and in the formal specification and description techniques, we justify that OCL is one of the appropriate formal languages for the ODP information viewpoint specifications. Secondly, we use OCL for the information viewpoint specification of the ODP trading function. The OCL gives the abilities to attach constraints to any meta-model, to verify their coherence and to translate the constraints into code for evaluating them on instance models.

Key-Words: - RM-ODP, Information Modeling and Specification languages, OCL Language, Trading Function

1 Introduction
The distributed processing is rapidly growing as it allows increasing the performances, the evolution, and the reuse of existent systems. However, the heterogeneity of computer systems obviously makes much more difficult this task. For this, the OMG consortium defined an architecture OMA [1] whose objective is to ensure the inter-working of applications running on heterogeneous systems. It defines the architecture CORBA [2] whose the ORB realizes the communications between client objects and server objects. It defines also a language for the definition of interfaces (IDL). CORBA specifies the implementation of the object distributed applications but does not specify how to design those applications. This is the aim of the Open Distributed Processing. The RM-ODP [3-6] defines an object model and an architecture for the construction of open distributed systems. The object model [4] defines concepts for information and for processing. The architecture [5] comprises five viewpoints, a viewpoint language for each viewpoint. The enterprise, information and computational viewpoints are distribution transparent. The engineering viewpoint focuses on the mechanisms and functions required to support distributed interaction between objects in the system. The ODP functions define the functionalities of the ODP operating system assumed to process the difficulties inherent to distribution. They include the repository functions types functions [7] and the trading function [8].

However, the RM-ODP is not prescriptive about the use of any particular notation for the viewpoints. The viewpoints are abstract in the sense that they define what concepts should be supported, not how these concepts should be represented. Furthermore, it has been acknowledged that a certain formality in the viewpoint notations is necessary. Indeed, RM-ODP is a meta-norm [9], and only provides a framework for the definition of new standards. These standards include standards for ODP functions; standards for modeling and specifying ODP systems; standards for methodology, programming, implementing, and testing ODP systems. Based on the meta-modeling approach, we defined in [10] an UML-based syntax for the ODP QoS (Quality of Service) aware enterprise language. We focus in this paper on issues related to the formal notation for the ODP information viewpoint specifications. The ODP trading function which is intended to be a standard as well is selected as case study. The trading function is based on the concept of service and service constraints related to the concept of quality of service studied in [10].

2 On ODP Systems Engineering

2.1 The RM-ODP Information Concepts
The object model [4] defines the basic concepts concerned with existence and activity: the expression of what exists, where it is and what it does. The core concepts defined in the object model are object and action. An object is a model of an entity. It is characterized by its behavior and, dually, by its states. Depending on the RM-ODP viewpoint, the emphasis may be placed on the behavior or on the states. When the emphasis is placed on behavior an object is
informally said to perform functions and offer services, these functions are specified in terms of interfaces. An object is distinct from any other object by its identity. It interacts with its environment at its interaction points which are its interfaces. An action is something which happens. The other concepts defined in the object model are derived from the concepts of object and action; those are class, template, type, subtype/supertype, subclass/superclass, composition, and behavioral compatibility. Composition of objects is a combination of two or more objects yielding a new object. An object is behaviorally compatible with a second object with respect to a set of criteria if the first object can replace the second object without the environment being able to notice the difference in the object behavior on the basis of the set of criteria. A type (of an $<x>$) is a predicate characterizing a collection of $<x>$s. A class (of an $<x>$) defines the set of all $<x>$s satisfying a type. A $<x>$ template is the specification of the common features of a collection x in a sufficient detail that an x can be instantiated using it. An $<x>$ template is an abstraction of a collection of $<x>$s. Types, classes, templates are needed for object, interface, and action. An information specification defines the semantics of the information and the semantics of information processing in terms of a configuration of information objects, the behavior of those objects and environment contracts for the system. An information object template is defined in terms of static, invariant and dynamic schemata. Invariant schema: A set of predicates in one or more information objects which must always be true. The predicates constrain the possible states and state changes of the objects to which they apply. Static schema: A set of predicates in one or more information objects, at some point in time, subject to the constraints of any invariant schemata. Dynamic schema: A specification of the allowable state changes of one or more information objects, subject to the constraints of any invariant schemata. An information object is either atomic or composite. The state of the composite object is represented by the combined state of its component information objects. The information objects resulting from the instantiation of a composite information object template only exist as part of the instantiated composite object and have no meaning outside it.

2.2 Software Engineering Methods versus ODP Systems: Theory and Practice

The standardization of ODP explicitly request the application of FDT, but the philosophy of ODP is much more complex as those of OSI. This becomes obvious, if we have a look at the need for support of the various viewpoints and for support of ODP object-oriented concepts. The languages Z [12], SDL [13] and LOTOS [14] [15] are used in [6] for the specification of ODP concepts. Elsewhere, up to now no formal method is likely to be suitable for specifying and verifying every aspect of an ODP system. The inherent characteristics of ODP systems imply the need to integrate different specification languages, and to handle non-behavioral properties of ODP systems. Also, it is recognized to integrate the well-established verification techniques; that is, the theorem proving and model checking techniques. We need to support all different kinds. Methods and tools should work in conjunction with each other. More precisely, we can build meta-method which itself produces methods customized for a particular problem domain. This represents the objective of the inter-working in the formal methods [16] which could complement less formal methods that are used in the overall system development process.

Elsewhere, the UML [16] is de facto standard for object-oriented modeling. An important aspect of the language is the need to provide a precise description of its semantics [17], [18], [19], [20], [21]. The intention is that this should act as an unambiguous description of the language. The task of formalizing UML has been addressed using various available formal techniques. Most of these attempts are complementary. We use the Precise UML (pUML) group formalization strategy [22]. The approach taken is to give a meta-model description of the language (describing the syntax of the UML within the UML itself). This is presented in terms of three views: the abstract syntax, well-formedness rules, and modeling elements semantics. The abstract syntax is expressed using a subset of UML static modeling notations. The well-formedness rules are expressed in OCL [11]. There has been an amount of research for applying the UML as a syntactic notation to the ODP viewpoints, [20], [21], [22].

2.3 Why to use OCL for the ODP information viewpoint specifications

We use OCL for the informational specification even if formal methods are also convenient such as Z and Object-Z [24], for several reasons. Firstly, the disadvantage of traditional formal languages is that they are usable to persons with a string mathematical background. Secondly, like Z, OCL can specify the ODP information concepts. Indeed, an invariant schema is the specification of the types of one or more information objects that will always be satisfied whatever behavior the objects may exhibit. A static schema is the specification of the state of one or more information objects at some particular point in time. These types are
subtypes of one or more of the types specified in the invariant schema. Behavior in an information specification can be modeled as transitions from one static schema to another, which reclassification of instances from one type to another. OCL can be used to specify invariants on classes and types in the class model, to describe pre- and post conditions on operations and methods. Hence, we deduce that OCL can be used to define the information semantics and semantics of information specification.

Thirdly, in the current context of integration and unification of formal methods and software engineering methods, OCL will serve as a common denominator for formal method semantics and software engineering method semantics. Indeed, we used the meta-modeling approach in our work [10] in order to define a syntax of a language for the ODP QoS-aware enterprise viewpoint specifications. We based on [17] where OCL is defined as a language for specifying the context constraints of the syntax of diagrammatical languages such as UML.

Elsewhere, the UML meta-model itself has little or no semantic content. UML is not adequate for ODP systems if you wish to be sure that the language is unambiguous and well-formed, or if you wish to build ODP automatic tools which are able to make use of semantic content. In order to give a semantics to a modelling language (which may not be directly executable), there are, essentially two approaches: an axiomatic approach, which states what sentences in the languages can be derived from other sentences; and a denotational approach, where expressions are mapped to the « instances » they denote. A denotational approach [23] would be realised by a) a definition of the form of an instance of every UML language element (e.g. the objects that could be denoted by a class, the links that could be denoted by associations, etc.) and b) a set of rules which determine which instances are and are not denoted by a particular language element. The pUML adopts a denotational, meta-modelling approach to the semantics. There are three main steps to the approach:

-Define the meta-model for the language of the model: classes, roles, models.
-Define the met-model for the language of the instances: objects, links, and snapshots.
-Define the mapping or the meaning function (also within the meta-model) between these two languages. We need a language to specify the meaning function and the two meta-models. Another advantage of OCL is that unlike Z, OCL language can be used to specify the meaning function if a denotational semantics definition.

Finally, for testing ODP systems [4], the current testing techniques [25], [26] are not widely accepted. However, a new approach for testing, namely agile programming [27], [28] or test first approach [29] is being increasingly adopted. The principle is the integration of the system model and the testing model using UML [30]. This approach is based on the executable UML [31]. In this context OCL is used to specify the properties to be tested. OCL also serves to attach constraints to UML meta-models in order to verify the coherence of meta-models and to translate the constraints into code for evaluating them on instance models.

3 The Informational Specification of the ODP Trading Function

The ODP trader is chosen as case study because it appears as a first main application of ODP. Furthermore there are informal specifications of the five viewpoints given in the standardization documents of the ODP trader. The ODP trading function allows objects to offer their services and match their needs against offered services. It is performed by an object called trader. Offering a service is called export, discovering a service is called import. To export, an object gives a trader a description of a service together with the location of a computational interface at which that service is available. To import, an object asks the trader a service having some characteristics, the trader checks against the descriptions of services and responds the importer with the location of the selected service interfaces. Due to the sheer number of service offers that will be offered, the trading service will be split up and the service offers will be partitioned. Hence, the trading system consists of a collection of inter-working linked traders; each of them manages a partition of service offers.

3.1 Informational Specification

<table>
<thead>
<tr>
<th>Template Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invariant schema:</strong></td>
</tr>
<tr>
<td>Offers: Collection(Offer),</td>
</tr>
<tr>
<td>Nodes: Collection(Node),</td>
</tr>
<tr>
<td>Edges: Collection(Edge),</td>
</tr>
<tr>
<td>Partitions: Collection(Partition)</td>
</tr>
<tr>
<td><strong>Initial schema:</strong></td>
</tr>
<tr>
<td>Information(),</td>
</tr>
<tr>
<td>Offers = {},</td>
</tr>
<tr>
<td>Nodes = {},</td>
</tr>
<tr>
<td>Edges = {},</td>
</tr>
<tr>
<td>Partitions = {}</td>
</tr>
<tr>
<td><strong>Dynamic schema:</strong></td>
</tr>
<tr>
<td>Export(): adds a service offer to service offer space of the trading system,</td>
</tr>
</tbody>
</table>
Withdraw(): withdraws a service offer
ModifyOffer(): changes the service property and service offer property values associated with a service offer whilst preserving the service offer identifier.

AddEdge(): adds an edge,
RemoveEdge(): removes an edge,
ModifyEdge(): changes the property of an edge,
AddNode(): adds a node to the trading system nodes,
RemoveNode(): removes a node,
Import(): searches for the subset of service offers which satisfy some matching, scoping and some preference constraints.

The information of the trading system is a composite object described by the object template Information. The component objects of the information object of the system are:

**Template Offer**
- ServiceDescription: Service;
- ServiceOfferIdentifier: Reference(Offer);
- ComputingInterfaceIdentifier: Reference(Interface);
- OfferProperties: Collection(Property);

**Template Service**
- ServiceSignature: Collection(Operation);
- ServiceProperties: Collection(Property)

**Template Partition**
- NodeRef: Reference(Node);
- NodeOffers: Collection(Offer);

**Template Edge**
- FirstNode: Reference(Node);
- SecondNode: Reference(Node);
- EdgeProperties: Collection(Property);

**Template Node**
- NodeRef: Reference(Node);
- NodeProperties: Collection(Property);

**Template Property**
- Name: CharString;
- Type: MetaType;
- Mandatory: Boolean;
- Readonly: Boolean;

**Template Operation**
- Name: CharString;
- TypeReturn: MetaType;
- Parameters: Collection(Parameter);

**Template Parameter**
- Name: CharString;
- Type: MetaType;
- PassageMode: Enumeration("in", "out", "in/out");

Export(in NewOffer: Offer, in Anode: Node, out OfferRef: Reference(Offer) )

Pre:
1. self.nodes → includes(Anode)
2. self.offers → forAll(p/ p.ServiceOfferIdentifier = OfferRef).

Post:
(1) self.offers → includes(NewOffer)
(2) self.partitions → exists(p/ p.NodeRef=Anode.NodeRef .NodeOffers → includes(NewOffer)).

WithdrawOffer(in OfferRef: Reference(Offer) )

Pre:
self.offers → exists(p/ p.ServiceOfferIdentifier = OfferRef)

Post:
(1) self.offers → forAll(p/ p.ServiceOfferIdentifier <> OfferRef )
(2) self.partitions → forAll(p/ p.NodeOffers → Not Exists(q/ q.ServiceOfferIdentifier = OfferRef))

ModifyOffer(in OfferRef:Reference(Offer)
ServiceProperties,OfferProperties: Collection(property))

Pre:
self.offers→exists(p/p.ServiceOfferIdentifier= OfferRef)

Post:
(1) \( \text{self.offers} \rightarrow \text{Exists}(p/ \text{p.ServiceOfferIdentifier} = \text{OfferRef} \text{ and p.ServiceDescription.ServiceProperties = ServiceProperties and p.OfferProperties=OfferProperties}) \)

(2) \( \text{self.partitions} \rightarrow \text{Exists}(p/ \text{p.NodeOffers} \rightarrow \text{select}(q/ \text{q.ServiceOfferIdentifier} = \text{OfferRef} \text{ and q.ServiceDescription.ServiceProperties = ServiceProperties and q.OfferProperties = ServiceOfferProperties})) \)

AddEdge(in NodeRef1, NodeRef2: Reference(Node); in EdgeProperties: Collection(Property))

Pre:
(1) \( \text{self.nodes} \rightarrow \text{Exists}(p/ \text{p.NodeRef}=\text{NodeRef1}) \)
(2) \( \text{self.nodes} \rightarrow \text{Exists}(p/ \text{p.NodeRef}=\text{NodeRef2}) \)
(3) \( \text{self.edges} \rightarrow \text{Not Exists}((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1})) \)

Post:
\( \text{self.edges} \rightarrow \text{Exists}(p/ ((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1}))) \)

RemoveEdge (in NodeRef1, NodeRef2: Reference(Node))

Pre:
(1) \( \text{self.nodes} \rightarrow \text{Exists}(p/ \text{p.NodeRef}=\text{NodeRef1}) \)
(2) \( \text{self.nodes} \rightarrow \text{Exists}(p/ \text{p.NodeRef}=\text{NodeRef2}) \)
(3) \( \text{self.edges} \rightarrow \text{Not Exists}((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1}))) \)

Post:
\( \text{self.edges} \rightarrow \text{Not Exists}(p/ ((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1}))) \)

ModifyEdge (in NodeRef1, NodeRef2: Reference(Node) in EdgeProperties: Collection(Property))

Pre:
\( \text{self.edges} \rightarrow \text{Exists}(p/ ((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1}))) \)

Post:
\( \text{self.edges} \rightarrow \text{Exists}(p/ ((\text{FirstNode}=\text{NodeRef1} \text{ and SecondNode}=\text{NodeRef2}) \text{ or } (\text{FirstNode}=\text{NodeRef2} \text{ and SecondNode}=\text{NodeRef1}))) \)

AddNode (in NodeRef: Reference (Node); in NodeProperties: collection(Property))

Pre: self. nodes \( \rightarrow \text{Not Exists}(p/ \text{p.NodeRef}=\text{NodeRef}) \)

Post:
(1) \( \text{self.nodes} \rightarrow \text{Exists}(p/ \text{p.NodeRef}=\text{NodeRef}) \)
(2) \( \text{self.partitions} \rightarrow \text{Not Exists}(p/ \text{p.NodeRef}=\text{NodeRef} \text{ and Not(p.NodeOffers isEmpty)} \text{ and p.NodeOffer.isEmpty})) \)
(3) \( \text{self.Edges} \rightarrow \text{forAll}(p/\text{p.FirstNode} <> \text{NodeRef} \text{ and p.SecondNode nodeRef}) \)

Post:
\( \text{self.NODES} \rightarrow \text{Not Exists}(p/ \text{p.NodeRef}=\text{NodeRef}) \)

Import (in Aservice: Service; in MatchingCriteria, ScopeCriteria, referenceCriteria: Expression; out Offers: Collection (Offer))

Pre:

Post: Offers = (self.offers \rightarrow \text{select} (\text{MatchingCriteria and ScopeCriteria and PreferenceCriteria}))

4 Conclusion
We address in this paper issues related to the ODP information specification language and ODP. We justify that OCL is an appropriate language for the ODP information specifications. We use OCL for the information specification of the ODP trading function. Using the meta-modeling approach, we are defining a meta-model syntax and semantics for the information language. Also we investigate how to use OCL as a language for specifying properties to be tested in ODP enterprise and information viewpoint specifications.

References


