UML4ODP: OCL 2.0 Constraints Specification & UML Modeling of Interfaces in the Computational Metamodel

OUSSAMA REDA, BOUABID EL OUAHIDI
Mohammed-V University, Faculty of Sciences
Dept of Computer Sciences
Ibn Battouta P.O Box 10 14, Rabat
MOROCCO
ouahidi@fsr.ac.ma, reda_oussama@yahoo.fr

DANIEL BOURGET
ENST Bretagne
Dept of Computer Sciences
Technopôle Iroise - CS 83818, 29238 Brest
FRANCE
Daniel.Bourget@enst-bretagne.fr

Abstract:- The purpose of this work is analysis of computational language concepts and introduction of novel pertinent ones in order to provide a new computational metamodel of interaction signatures in UML4ODP FDIS. we mainly introduce the concept of *Functional* computational interface which unify signal and operation interfaces notions. The unification of *signal* and *operation* interactions concepts is presented by introducing the *Parameterized* interaction concept. We show that parameterized interactions are of two main kinds; namely, *primitives* and *compounds*. we also introduce the notion of *incoming* and *outgoing* primitives. As an application of our modeling choices we redefine interaction, refinement and type checking rules in a concise manner, and then specify them using the useful specification functionalities of OCL 2.0, showing how novel definitions as well as their specification are easy to read, write and understand.

Key–Words: RM-ODP, UML4ODP, Computational language, Meta-modeling, Computational interface, Interaction Signature, type checking, Interaction refinements.

1 Introduction

The expansion of distributed processing field has led to the ODP standardization initiative which consists of a framework [1][2][3][4]by which distributed systems can be modeled using five viewpoints. The computational viewpoint is concerned with the description of the system as a set of objects that interact at interfaces constrained by rules such that, amongst others, typing rules, naming rules and refinement rules. No particular formal description and specification techniques for the specification of ODP systems has been prescribed by RM-ODP to be used. Over the past several years, there has been a considerable amount of research [6][5][11][7][8][9][10][13][14][**?**][16][12][15][17] [18][19][20] in the field of applying the UML Language as a formal notation with the ODP viewpoints, particularly for the computational language. The outcome of these works, amongst others, was the adoption of the UML4ODP *FDIS* (*Final Draft International Standard)* [23] which provides the necessary needed framework for ODP systems specification using UML 2.0 [24]. Works [13][14][16] within the computational viewpoint have mainly addressed the specification of the functional decomposition of an ODP system using UML. [**?**][15][17][18][19] [20] have shown the UML4ODP computational metamodel contains inconsistencies concerning the semantic relationship between interaction signatures concepts and action templates, then proposed in reliable solutions. In the same perspective we analyze computational interfaces and interaction signatures concepts so as to resolve residual inconsistencies in the UML4ODP computational metamodel.

On the other hand, the second main focus of this work is application of our modeling choices to the redefinition of refinements of interactions, typing rules as well as naming rules. We shall see how to refine any kind of interactions into *primitives* which are elementary interactions and provide OCL constraints relating to those refinements. we also see how can we redefine operational interfaces typing rules and general interaction rules concisely then, specify them in OCL 2.0[25]. In doing so, we are addressing fundamental QoS (Quality of Service) as well as distribution transparencies issues.

The remainder of the paper is organized as follows. In Section 2, we present concepts of computational interfaces and interactions signatures provided by RM-ODP. We address in section 3 the problem concerning the relationship between *Interaction Signatures* and *Action Templates*. In section 4 we lead a conceptual algebraic analysis of *computational interfaces* and *interaction* signatures notions in order to steadily define these concepts. We prove interactions are either *parameterized* or flows, then classify *parameterized* interactions in *primitives* and *compounds* ones. We also define the concept of *functional* computational interface. The result of this analysis is the definition and introduction of new concepts to the computational language. Section 5 deals with the introduction of complementary concepts (*incoming* and *outgoing* interactions) to those given in section 4 in order to provide the final model of computational interfaces and interaction signatures. In section 6 we discuss interaction refinements rules and provide their specification in OCL 2.0. We also discuss typing rules of operational computational interfaces and general interaction rules, then specify those rules in OCL 2.0 based on this discussion. A conclusion and perspectives end the paper.

2 Computational Interface Signatures concepts definitions

In this section, we present interaction signatures concepts as they are defined in the computational language. These definitions will serve us to discuss the ideas of the rest of the paper. the definitions are given as follows:

A Computational interface template is an interface template for either a signal interface, a stream interface or an operation interface. Each interface has a signature:

- a signal interface signature comprises a finite set of *Action Templates*, one for each signal type in the interface. Each action template comprises the name for the signal, the number, names and types of its parameters and an indication of causality (initiating or responding, but not both) with respect to the object which instantiates the template.

- An operation interface signature comprises a set of announcement and interrogation signatures as appropriate, one for each operation type in the interface, together with an indication of causality (client or server, but not both) for the interface as a whole, with respect to the object which instantiates the template.

  Each announcement signature is an action template containing both the name of the invocation and the number, names and types of its parameters.

  Each interrogation signature comprises an action template with the following elements : the name of the invocation; the number, names and types of its parameters, a finite, non-empty set of *Action Templates*, one for each possible termination type of the invocation, each containing both the name of the termination and the number, names and types of its parameters.

- A stream interface comprises a finite set of action templates, one for each flow type in the stream interface. Each action template for a flow contains the name of the flow, the information type of the flow, and an indication of causality for the flow (i.e., producer or consumer but not both) with respect to the object which instantiates the template.

  These concepts are the necessary and sufficient ones for our proposals.

3 Interaction Signatures: What is the matter?

The matter with interaction signatures concepts is the difficulty of expressing *operation signatures* in terms of *Action Templates* since it is not obvious whether *operation signatures* are kinds of *Action Templates* or are constituents of *Action Templates*. Another issue concerning interaction signatures is the way we can describe all of them in terms of *Action Templates* in one blow. The problem with all this difficulty in modeling is that the definitions of the concepts are not precise and leaves room to plenty of interpretations. To eliminate this ambiguity one have to analyze the definitions on a conceptual level in order to bring out the exact semantic relationships between those concepts.
In fact we show interactions are of two kinds: *parameterized* interactions and flows. *Parameterized* interactions are composed by *primitive* and *compound* interactions, *primitives* are being *incoming* and *outgoing* interactions.
On the other hand, interface signatures are defined in terms of three kind of computational interfaces. However when we analyze interface signatures concepts we show that in fact there are only two relevant categories they are to be classified in. We shall see how interface signatures can principally be classified in two main classes, namely; *Functional* interface signatures and stream interface signatures.

4 *Functional* Interface Signature & *Parameterized* Interaction Signatures

We begin by introducing the notation needed to demonstrate our propositions.

Notation:

- The symbol $\bigcap$ denote the intersection of algebraic sets (it has the same meaning as it is in classical set theory).
- Di, Pi and Ci are respectively the contracture of Definition i, Proposition i and corollary i, where i is an integer related to the order of their appearance in the text.
- A$\setminus$B denotes the set of elements which are in A and are not in B.
- *bby* is the contracture of *by and only by*.

Let SAinv, SAann, SAint, SAter, SAflo, denote the sets of attributes that respectively describe signatures of *Invocations*, *Announcements*, *Interrogations*, *Terminations* and finally *Flows*.

Definition 1:
An *Action Template* is defined *bby* the name of the action and its causality.

Proposition 1: All Interaction Signatures are Action Templates.

Proof :
We have :

SAinv = SAann = SAint = SAter = {name, numbers of parameters, names of parameters, types of parameters, causality} and separately SAflo={name, causality, information type}.

The led set of these sets denoted SA which is their intersection SA = SAinv ∩ SAann ∩ SAint ∩ SAter={name, causality} is the set composed *by and only by* both the name and causality of interaction signatures. Moreover, the Action Template concept is involved in the core description of all interaction Signatures concepts, and since the UML semantic of intersection is a `generalization`, it follows that all Interaction Signatures are *Action Templates*.

Proposition 2: Interaction Signatures but flows are parameterized(i.e contain finite set of parameters as well as their name and numbers).

Proof :
The sets SAinv$\setminus$SA, SAann$\setminus$SA, SAint$\setminus$SA, SAter$\setminus$SA have the same elements since SAinv$\setminus$SA= SAann$\setminus$SA = SAint$\setminus$SA = SAter$\setminus$SA ={numbers of parameters, names of parameters, causality}. Consequently, All Interaction Signatures but *Flow Signatures* are parameterized (i.e described by finite sets of parameters as well as their names and numbers).

Now, when we separately take the set SAflo$\setminus$SA={information type} we deduce that flow signatures are of different nature than the other Interaction Signatures.

Flow Signatures are *Action Templates* with an (information type) attribute which is not significant to the other interactions. Conversely, all Interaction Signatures have parameters, their name and their numbers as attributes which do not contribute to the description of flows.

Definition 2:
A *Parameterized* interaction signature is an *Action Template* with a finite set of parameters as well as their numbers.

Corollary 1:
From P1, P2 and the definition of interface signatures given in the previous section we have :

1. Interaction signatures are of two kinds: ***Parameterized* interactions signatures** and flow interactions signatures.
2. Operation Interfaces signatures and Signal Interfaces signatures are composed *bby* ***Parameterized* interaction signatures**.
3. A stream interface signature is composed *bby* a set of flow interactions signatures.

Definition 3: A ***Functional*** Interface Signature is an interface signature composed *bby* ***Parameterized* interactions signatures**.

Corollary 2: From D3 and the definition of interface signatures given in the previous section we have :
Interface Signatures are of two kinds, namely; ***Functional*** Interface Signature and Stream Interface Signatures.

5 Computational Interface Signatures UML Metamodel

In this section we model computational interfaces and interaction signatures by means of constructs of the UML language.
Interactions in the computational language are of three kinds (signals, operations and flows). We have shown

interactions are of two main kinds: *Parameterized* interactions and flows. Signals in the computational language are defined as being atomic interactions that constitutes the building blocks of the other kinds of interactions. Similarly, *parameterized* interactions are classified in two main categories: *Primitive parameterized* interactions (homologous of signals) and *compound parameterized* interactions (homologous of operations). This classification is necessary to guarantee that the metamodel given (see figure 1) serves as a basis to define end-to-end QoS in open distributed systems, and the operation of multi-party binding and bindings between different kinds of interfaces (e.g. stream to operation interface bindings).

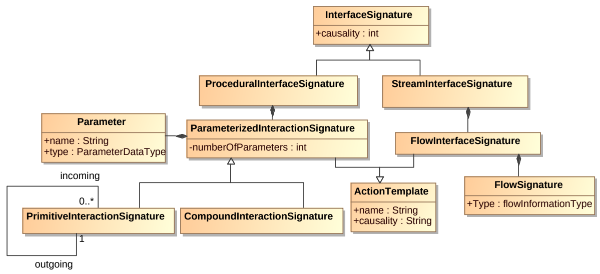

Figure 1: Functional interface signature & Parameterized interaction signatures

In the computational language operations in a computational interface consist of invocations and announcements which are *outgoing* interactions. To each invocation in the interface corresponds a finite non empty set of terminations which are *incoming* interactions. In [18] we have shown that invocations and announcements do play the same role conceptually and practically. Thus, We define *Compound parameterized* interaction signatures as being composed by two kinds of interactions: *outgoing* interactions and *incoming* interactions. Indeed, invocations and announcements are identical *outgoing* interactions. Moreover, invocations and terminations which are (terminations) incoming interactions are associated to each other by a one to many correspondance, and since announcements can be replied to or not during the interaction, we conclude there is a correspondance between *outgoing* and *incoming* interactions. That is, for every *outgoing* interaction corresponds a finite set (possibly empty) of *incoming* interactions (see figure 1).
Signals in the computational language are the least degree of representation of interactions between computational objects. Since signals do provide the constructing bricks of all other interactions, it is tempting to make use of them in order to refine interactions in their terms. To do so, the computational language imposes rules on these mappings so as to provide for reliable refinements when required. *Primitive parameterized* interactions do play the role of signals. While operations are represented in terms of signals, *compound parameterized* interactions can be decomposed in terms of *primitive parameterized* interactions which are now elementary *parameterized interactions*.
Another application of our conceptual modeling choices is specification of computational interfaces non recursive type checking system. We present the definitions of computational typing rules[18][22]and redefine them based on the novel concepts presented before. In effect, We provide novel definitions based only on *incoming* and *outgoing* interactions. Finally, we redefine then specify in OCL interaction naming rules as well as interaction parameters naming rules in a concise manner based on the concepts introduced before.

6 Application

In this section, we show how our unification choices in modeling computational interface signatures and interaction signatures concepts help us to specify compact OCL constraints applied on interaction rules, computational interface refinements and type checking rules.

6.1 Interaction rules

6.1.1 Parameterized interaction rules & OCL constraints specification

Each interaction of a computational object occurs at one of its computational interfaces. The computational language imposes constraints on the behavior permitted at a computational interface. Particularly, constraints are imposed on the kind of interaction occurring at well defined interfaces[3].
The interaction part of the computational language supports three models of interaction, each of which has an associated kind of computational interface:

- Signals and signal interfaces.
- Flows and stream interfaces.
- Operations and operation interfaces.

<u>Flow interaction rules:</u>

A computational object offering a signal interface of a given signal interface type[3]:

- Generates signals that have initiating causality in the interface's signature.
- Receives to signals that have responding causality in the interface's signature.
  Since stream interface signatures and flow interaction signature have not been redefined in the current work, flow interaction rules OCL specification are not presented here.

Signal interaction rules:

A computational object offering a signal interface of a given signal interface type[3]:

- Initiates signals that have initiating causality in the interface's signature.
- Responds to signals that have responding causality in the interface's signature.

Operation interaction rules:

A client object using an operation interface invokes the operations named in the interface's signature. A server object offering an operation interface expects any of the operations named in the interface's signature. In the case of an interrogation, the server responds to the invocation by initiating any one of the terminations named for the operation in the server interface signature. The client expects any of the terminations named for the operation in the client interface signature. The duration of the operation is arbitrary unless required otherwise by environnement contracts applicable and interfaces involved[3].

Since signals and operations are *parameterized interactions*, operation and signal interaction rules can be reduced to only rules applicable to *parameterized interaction signatures* occurring in the context of *functional* interfaces. *"client"* and *"initiate"* causalities are reduced to *"acting"* causality while *"server"* and *"respond"* causalities are reduced to *"reacting"* causality. Having stating that, the new interaction rules of parameterized interactions are as follows:

A computational object offering a ***functional*** *interface of a given signal interface type*:

- ***Enact*** PARAMETERIZED *interactions that have* ***"acting"*** *causality in the interface's signature.*

  The OCL specification of this rule is given as follows:

  Context *FunctionalInterfaceSignature* **inv :**

  Let *PIS : ParameterizedInteractionSignature = ParameterizedInteractionSignature.oclAsType(PrimitiveInteractionSignature)*

  FunctionalInterfacesignature→forAll(PIS : P |

  P.oclAsType(PIS.outgoing)
  implies
  P.causality="acting"

- ***React*** *to* PARAMETERIZED *interactions that have* ***"reacting"*** *causality in the interface's signature.*

  The OCL specification of this rule is given as follows:

  Context *FunctionalInterfaceSignature* **inv :**

  Let *PIS : ParameterizedInteractionSignature = ParameterizedInteractionSignature.oclAsType(PrimitiveInteractionSignature)*

  FunctionalInterfacesignature→forAll(PIS : P |

  P.oclAsType(PIS.incoming)
  implies
  P.causality="reacting"

6.1.2 Interaction naming rules & OCL contraints specification:

Each kind of name defined in the computational language has an associated context, as follows:

- A signal name in a signal interface signature is an identifier in the context of that signature.
- A flow name name in a stream interface signature is an identifier in the context of that signature.
- A invocation name in a operation interface signature is an identifier in the context of that signature.

- A termination name in a signal interface signature is an identifier in the context of the operation template in which it appears.

Since signal invocation and terminations are primitive interactions, interaction naming rules can be redefined in just two rules, as what follows:

- *A primitive interaction name in a functional interface signature is an identifier in the context of that signature.*

  The OCL specification of this rule is given as follows:

  Context *FunctionalInterfaceSignature* **inv :**

  Let *PIS : ParameterizedInteractionSignature = ParameterizedInteractionSignature.oclAsType(ParameterizedInteractionSignature)*

  PIS →forAll(p,q |
  p.name=q.name
  implies
  p=q)

- A flow name in a stream interface signature is an identifier in the context of that signature.

  The OCL specification of this rule is given as follows:

  Context *StreamInterfaceSignature* **inv :**

  Let *FIS : FlowInteractionSignature = FlowInteractionSignature.oclAsType(FlowInteractionSignature)*

  FIS →forAll(p,q |
  p.name=q.name
  implies
  p=q)

6.1.3 Interaction parameters naming rules & OCL contraints specification:

- The name of a parameter in a signal template in a signal interface signature is an identifier in the context of that template.
- The name of a parameter in an invocation template in an operation interface signature is an identifier in the context of that template.
- The name of a parameter in a termination template in an operation interface signature is an identifier in the context of that template.

Since flows have no parameters[21][18][22] and that signals, invocations and terminations are primitive interactions, interaction parameters naming rules are reduced to only one rule, it is defined as what follows:

- *The name of a parameter in a primitive interaction template in a* functional *interface signature is an identifier in the context of that template.*

  The OCL specification of this rule is given as follows:

  Context *ParmeterizedInteractionSignature* **inv :**

  Let *PIS : ParameterizedInteractionSignature = ParameterizedInteractionSignature.oclAsType(ParameterizedInteractionSignature)*

  PIS →forAll(parameter: p,q |
  p.name=q.name
  implies
  p=q)

6.2 Refinement rules for Interaction Signatures

An operation or a flow can be resolved in terms of a composition of several individual signals. For instance, we can interpret an interrogation in terms of a sequence of four signals: invocation emission (by the client object), invocation receipt (by the server object), termination emission (by the server), termination receipt (by the client). In opposition, since the computational model do not provide the precise semantics of flows, their mapping on signals is not defined. In fact, a definition of flows using signals depends upon the details of the interactions abstracted in the specification of the stream interface concerned and therefore is beyond the scope of the ODP Reference Model [3].

In [19] we specified those constraints based on their definitions provided in RM-ODP [3] which are given as follows :

- In a signal interface corresponding to a client operation interface there is a signal -invocation submit- corresponding to each invocation with the same parameters. in the case of an interface containing interrogations, a signal - termination deliver - corresponding to each possible termination with the same parameters as that termination.

- In the signal interface corresponding to a server operation interface there is a signal -invocation deliver- corresponding to each invocation with the same parameters. in the case of an interface containing interrogations there is a signal - termination submit- corresponding to each possible termination with the same parameters as that termination.

In the definitions above the correspondance rules do neither depend on the causality of computational interfaces nor the causality of interactions. What only matters is the existence of a corresponding refining interaction with the same parameters. Thus we can redefine those rules in one unified rule applied to *parameterized* interactions establishing a correspondance between *primitive* and *compound parameterized* interactions. This rule is given as follows :

Compound into primitive refinement rule:

For each Compound parameterized interaction there is a corresponding Primitive Parameterized interaction with the same parameters.

We can break down this rule to bring OCL sub-expression out of it which establishes a correspondance between parameterized interactions. That is, two parameterized interactions related by a refinement relationship must have the same parameters. The OCL sub-expression is given in what follows:

Context *ParameterizedInteractionSignature* **inv:**

def: *hasSameParameters(PIS:*
ParamterizedInteractionSignature): Boolean =
self.Parameter → forAll(Px : Parameter |
ParamterizedInteractionSignature →Exists(
Py: Parameter |
Px.name = Py.name
and
Px.type = Py.type))

The final OCL constraint to refine *Compound* interactions into *primitives* interactions is given in what follows:

Context *ParameterizedInteractionSignature* **inv:**

Let *CIS : ParameterizedInteractionSignature =*
ParameterizedInteractionSignature.oclAsType(
CompoundInteractionSignature)

Let *PIS: ParameterizedInteractionSignature =*
ParameterizedInteractionSignature.oclAsType(
PrimitiveInteractionSignature)

CIS → forAll(c |
PIS → Exists(P |
CIS.hasSameParameters(PIS)))

This constraint establishes a correspondance between *primitive* and *compound* interactions of *functional* computational interfaces, thus providing for end to end QoS characteristics to be defined, as well as allowing for different kind of computational interfaces to be bound (e.g *functional* to stream interfaces bindings).

6.3 Type checking rules specification in OCL 2.0

In this subsection we specify semantics of interaction signatures related to subtyping rules. Type checking rules are not precisely defined in the computational language[21][18][22]. In [18][22] we have redefined those literal rules in order to consistently specify them in OCL. Typing rules corresponding to operation interface signatures are defined[18][22] as follows:

Operation interface X is a signature subtype of interface Y if the conditions below are met:

- For every interrogation in Y, there is an interrogation signature X with the same name, with the same numbers and names of parameters and that each parameter in the interrogation signature in Y is a subtype of the corresponding parameter in the interrogation signature in X.

- For every termination in an interrogation signature in Y, there is a corresponding termination in interrogation signature X with the same name, with the same numbers and names of parameters and that each parameter in the termination of the interrogation signature in X is a subtype of the interrogation signature in Y.

- For every announcement in Y, there is an announcement signature X with the same name, with the same numbers and names of parameters and that each parameter in the interrogation signature in Y is a subtype of the corresponding parameter in the interrogation signature in X.

Since interrogations (invocations in the context of type checking rules definitions) and announcements are *outgoing* interactions and on the other hand, terminations are *incoming* interactions we can redefine those rules which will hold in only two rules:
Functional Interface X is a signature subtype of interface Y if the conditions below are met:

- *For every outgoing interaction signature in Y there is a corresponding outgoing interaction signature in X with the same name, with the same number and names of parameters, and that each parameter type in Y is a subtype of the corresponding parameter type in X.*
- *For every incoming interaction signature in X there is a corresponding incoming interaction signature in Y with the same name, with the same number and names of parameters, and that each parameter type in X is a subtype of the corresponding parameter type in Y.*

In [21] it is shown that ODP does not distinguish between the clients and servers when establishing type relationships for operational computational interfaces. This leads to incorrect type checking rules specification. In our model this is implicitly stated by the fact that we only have two kinds of primitive interactions; namely, incoming and outgoing interactions.

At this point, we specify typing rules in OCL 2.0. We can break down those two rules to bring OCL sub-expression out of them which establishes a correspondance between incoming interactions and outgoing interactions for two computational interfaces related by a type/subtype relationship. That is, two incoming or two outgoing interactions related by a type/subtype relationship must have the same name, the same number and names of parameters, and that corresponding parameter types verify type/subtype relationship following the rules provided above. The OCL sub-expressions are given in what follows:

Context *ParameterizedInteractionSignature* **inv:**

def: *hasSameName(PAT: ParameterizedInteractionSignature): Boolean =*
self.name= PAT.name)

def: *hasSameParametersNumber(PAT: ParameterizedInteractionSignature): Boolean =*
(self.parameternumbers= PAT. parameternumbers)

def: *hasSameParametersNames(PAT: ParameterizedInteractionSignature): Boolean =*
self.Parameter → forAll(Px : Parameter |
ParameterizedInteractionSignature → Exists(
Py: Parameter |
Px.name = PAT.Py. name))

def: *isSubTypeOf (PAT: ParameterizedInteractionSignature): Boolean =*
self.Parameter → forAll(Px : Parameter |
ParameterizedInteractionSignature → Exists(
Py: Parameter |
PAT.Py.type.oclIsKindOf(Px.type)))

Based on the sub-expressions above we can specify Functional typing rules interfaces as follows:

Context *FunctionalInterfaceSignature* **inv :**

Let *PIS : ParameterizedInteractionSignature =*
ParameterizedInteractionSignature.oclAsType(
PrimitiveInteractionSignature)

FunctionalInterfacesignature→forAll(X,Y |

(PIS.outgoing →forAll(PY |
PIS.outgoing →exists(PX |
Y.PY.hasSameName(X.PX)
and
Y.PY.hasSameParametersNumbers(X.PX)
and
PY.hasSameParametersNames(X.PX)
and
Y.PY.isSubTypeOf(X.PX)))))

and

(PIS.incoming →forAll(PY |
PIS.incoming →exists(PX |
Y.PY.hasSameName(X.PX)
and
Y.PY.hasSameParametersNumbers(X.PX)
and
PY.hasSameParametersNames(X.PX)
and
X.PX.isSubTypeOf(Y.PY)))))

implies

X.oclIsKindOf(Y)

Since stream interface signatures and flow interaction signature have not been redefined in the current work, stream interface signatures typing rules OCL specification are not presented here, since they are provided in [18][22].

7 Conclusion

We analyze in this work computational interfaces and interaction signatures in order to consistently model them within the UML4ODP computational metamodel. Computational interfaces in the computational language are of three kinds (signals, operations and streams). We show that computational interfaces are classified in two main classes instead of three: Functional and stream interfaces. We also demonstrate that interactions are of two kinds, namely; Parameterized and flowing interactions. Then, we show that only two kinds of parameterized interactions have to be taken into account, Primitive and Compound interactions, primitives are being incoming or outgoing interactions. Based on these, we provide a UML metamodel of interfaces and interactions signatures. Finally, we show how our modeling choices prove to be pertinent to specify OCL constraints on refinements of interactions to define end-to-end QoS and bindings between computational interfaces. We also provide OCL specification of type checking rules which are essential to support distribution transparencies as well as general rules in a concise manner.

8 perspectives

Interactions between given computational interfaces is only possible if a binding (i.e. some communication path) has been established between them. Binding in the Reference Model is defined with reference to binding actions. Use of such actions is called explicit binding. There are two kinds of binding actions: primitive binding actions and compound binding actions. Primitive binding actions enable binding of an interface of the object which initiates the action to another interface (of another object, or itself). Compound binding actions enable a set of interfaces to be bound, using a binding object to support the binding [1][3].
On the other hand, a primitive binding action binds two computational object directly. A compound binding action uses primitive binding actions linking two or more computational objects via a binding object.
The UML4ODP computational metamodel lacks the specification of binding refinements. Thus, there is a need to provide for such a refinement to be realized. We are looking forward to provide for such refinements, especially by establishing a correspondance between PRIMITIVE interactions and PRIMITIVE BINDING actions from one side and COMPOUND interactions and COMPOUND BINDING actions on the other side.
Finally, since all kinds of interactions may be mapped into primitive interactions (signals), many rules relating to interactions can be reduced to rules applied on *primitive* interactions (signals). We are investigating how to define all the rules relating to interactions in terms of rules corresponding to *primitives*.

References:

[1] ISO/IEC, *Basic Reference Model of Open Distributed Processing-Part1: Overview and Guide to Use* ISO/IEC CD 10746-1, 1994.

[2] ISO/IEC, *RM-ODP-Part2: Descriptive Model* ISO/IEC DIS 10746-2, 1994.

[3] ISO/IEC, *RM-ODP-Part3: Perspective Model* ISO/IEC DIS 10746-3, 1994.

[4] ISO/IEC, *RM-ODP-Part4: Architectural semantics* ISO/IEC DIS 10746-4, 1994.

[5] P.F. Linington et al., *The specification and testing of conformance in ODP systems*, http://citeseer.nj.nec.com/170353.html, 1999.

[6] M. W. A. Steen et al.,*Formalising ODP Enterprise Policies*, IEEE Com. Soc. Press, EDOC'99, 1999.

[7] X. Blanc et al., *Using the UML Language to Express the ODP Enterprise Concepts* In Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, pp. 50-59, September 1999. IEEE Computer Society Press.

[8] J. Aagedal et al., *ODP Enterprise Language: UML Perspective* In Proceedings of the 3rd 29 International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, pp. 60-71, September 30 1999. IEEE Computer Society Press.

[9] M. W. Steen and al. *ODP Enterprise Viewpoint Specification* In Computer Standards & Interfaces, 22(3):165-189, September 2000. Elsevier.

[10] OMG, *Relationship of the Unified Modelling language to the Reference Model of Open Distributed Processing* OMG document ormsc/2001-01-01, Version 1.4, January 2001.

[11] Interoperability Technology Association for Information Processing, Japan, *Guide for Using RM-ODP and UML Profile for EDOC* Document available at http://www.net.intap.or.jp/e/odp/intap-guide.pdf.

[12] , ID. Hashimoto et al., *UML 2 Models for ODP Engineering/Technology Viewpoints* In Proc of the Second International Workshop on ODP for Enterprise Computing (WODPEC 2005), pages 32-Enschede, The Netherlands, September 2005.

[13] B. Bordbar et al., *Using UML to specify QoS constraints in ODP*, Computer Networks Journal pp. 279-304, 2002

[14] D.H.Akehurst et al., *Addressing Computational Viewpoint Design*, Seventh IEEE International EDOC, IEEE Computer Society, 2003

[15] O. Reda et al., *Interaction Signatures and Action Templates in the ODP Computational Viewpoint* In proceedings of SEPADS'07, 6th WSEAS International Conference on SOFTWARE ENGINEERING, PARALLEL and DISTRIBUTED SYSTEMS, Greece, Februry 16-19, 2007.

[16] R. Romeo et al., *Modelling the ODP Computational Viewpoint with UML 2.0* IEEE International Enterprise Distributed Object Computing Conference, 2005.

[17] O. Reda et al.*Resolving the ODP Computational Viewpoint Interaction Signatures in Terms of Action Templates using UML* ICTIS'O7 : Information and Communication Technologies InternatIonal Symposium , April 3-5, Fes, Maroc, 2007

[18] O. Reda et al. *Specification of OCL Constraints on ODP Computational Interfaces* 7th WSEAS International Conference on Applied Informatics And Communications (AIC'07), August 24-26, Athens, Greece, 2007

[19] O. Reda et al. *Towards a Refinement of the Open Distributed Systems Interactions Signatures* WSEAS transactions on communications, April 2007, vol. 6, pp. 601-607

[20] O. Reda et al. *Modeling of Interactions and Refinements of Bindings in the UML4ODP Computational Meta-Model*, International Journal on Information and Communication Technologies (IJICT 2008).

[21] Richard O. Sinnott and Kenneth J. Turner, *Type Checking in Open Distributed Systems: A Complete Model and its Z Specification*, Rolia, Jacob Slonim and John Botsford, editors, Open Distributed Processing and Distributed Platforms, pages 85-96, Chapman and Hall, May 1997.

[22] O. Reda et al, *OCL 2.0 Constraints Specification On Computional Interfaces of ODP Applications*, accepted in Proc. CARI'08 (Africain Conference on Research in Computer Science and Applied Mathematics) Rabat, October 2008.

[23] ITU-T Recommendation X.906 — ISO/IEC 19793, *Information technology Open distributed processing Use of UML for ODP system specifications*, SC 7/WG19 and ITU-T, 2007.

[24] OMG, *UML 2.0 Superstructure Specification*, OMG document formal/05-07-04, 2005 .

[25] OMG, OCL 2.0 Specification, version 2.0 OMG document ptc/05-06-06, 2005.