Open Distributed Processing

Typing rules OCL specification of *QoS-capable* ODP Computational Interfaces

Oussama Reda[*] — Bouabid El Ouahidi[**] — Daniel Bourget [**]

[*] Département d'informatique
Faculté des Sciences Rabat
B.P. 10 14 Rabat
Maroc {ouahidi, reda_oussama}@fsr.ac.ma
[**] Département d'informatique
Telecom Bretagne Technopole de l'Iroise
CS 83818 29238 Brest
{Bouabid.Ouahidi,Daniel.Bourget}@telecom-bretagne.eu

RÉSUMÉ. Dans ce travail, nous analysons le concept ODP de signatures d'interfaces de traitement et leurs règles de typages, le but étant de redéfinir les signatures de manière concise et compacte. Pour cela, nous modélisons les signatures par des concepts UML équivalents. Ensuite, nous spécifieons des contraintes imposées sur ces signatures d'interfaces de traitement ODP liées aux règles de typages et sous-typages. Nous allons montrer également comment nous pouvons littéralement redéfinir ces règles afin de mieux les formaliser en utilisant OCL 2.0. Par la suite nous introduisons trois nouveaux concepts qui sont: *functional* computational interfaces, ***QoS-definable interactions*** and ***QoS-capable interfaces***. Finalement, nous définissons les règles typologiques relatives aux ***QoS-capable interfaces*** et les spécifions en OCL 2.0

ABSTRACT. In this work we model the interaction signature concepts in a consistent and compact manner as well as their related type checking rules. First, we begin by literally analyzing those concepts in order to bring unambiguous definitions out of them. Following this analysis we shall formalize those concepts by mapping them into UML language constructs. Secondly, we specify constraints imposed on computational interfaces interaction signatures related to the computational language typing and subtyping rules. We shall show how we can we literally redefine those rules in order to steadily formalize them. After rewriting those rules in a compact way, we make use of OCL 2.0 which provides the means to exploit those new definitions. Then we introduce the concept of *Functional* computational interface and a set of related concepts which unify signal and operation interfaces notions. Based on the new additional concepts introduced, we introduce two new important concepts, namely; ***QoS-definable interactions*** and ***QoS-capable interfaces***. We then provide a UML metamodel of interfaces and interaction signatures. The final metamodel being a first step towards a ***QoS-capable computational metamodel***. Finally, as an application of our modeling choices we define ODP QoS-capable computational interfaces ***type checking rules*** and then specify them using OCL 2.0

MOTS-CLÉS : ODP, point de vue traitement, UML, OCL, Meta-modélisation, signature d'interface, QoS, règles de typages

KEYWORDS : ODP, Computational Viewpoint, UML, OCL, Meta Modeling, Interaction Interface signature, QoS, Type Checking Rules

1. Introduction

The expansion of distributed processing field has led to the ODP standardization initiative [1],[2],[3] which consists of a framework by which distributed systems can be modelled using five viewpoints. The computational viewpoint is concerned with the description of the system as a set of objects that interact at interfaces. A computational specification describes the functional decomposition of an ODP system in distribution transparent terms and is constrained by the rules of the computational language. These comprise, among others, typing rules. Researches in [4], [5],[6] has been particularly interested in applying UML [12]as a formal notation for the specification of the computational viewpoint. Works [4], [5] have mainly addressed the specification of the functional decomposition of an ODP system using UML. Authors in [16] have focused on how to consistently present concepts of the ODP computational viewpoint and clarified some ambiguities found while aiming to express them formally. The solutions proposed were given on a semantic level. Authors in [9],[10] [11] have also noted those issues, then, provided solutions and presented them on a syntactic level without the need to relegate them on a semantic one, as well as specifying constraints related to computational interface signatures typing and subtyping rules. However, the OCL specifications of those rules are not easy to write, and thus, are complicated to read and understand. This comes from the fact that OCL 1.1 doesn't provide any means in order to write easily comprehensible OCL expressions. OCL 2.0 has known significant enhancements, especially with the provision of expressions that allow the definition and reuse of variables/Operations over multiple OCL expressions. This fits well the specification of OCL constraints on typing/subtyping rules associated to interaction signatures, since those rules are redundant and have the same literal description pattern within their definitions. The attempt of this work is to model concepts of the ODP computational viewpoint and our main focus is the formalization of concepts of the interaction signature part as well as specification of their associated typing and subtyping rules. In this respect, we use the UML language to discuss and present our proposals. We also use the OCL 2.0 language [13]to specify clear and understandable constraints associated to computational signature interfaces typing rules. More importantly, we consider QoS (Quality of service) aspects of distributed applications by enhancing the computational metamodel with QoS features in a whole new fashion.
The remainder of the paper is organized as follows. In section 2 we present the interaction signatures concepts as they are defined in the computational viewpoint. These definitions will serve us to discuss the remainder of the paper. Section 3 shows interaction signatures can be defined and modelled in a precise and concise way. Section 4 enhances the metamodel elaborated in section 3. We first lead an algebraic analysis of interaction signatures concepts, then, based on the results of this analysis we elaborate a new interface signatures UML metamodel. We mainly show interactions are of two kinds : discrete (parameterized) interactions and flows. Discrete interactions are composed by primitive and compound interactions, primitives which are incoming or outgoing interactions. On the other hand, we show interface signatures can principally be classified in two main classes, namely ; Procedural interface signatures and stream interface signatures. Based on those new concepts we have elaborated a new metamodel of interaction and interface signatures. In section 5 we introduce QoS-definable interaction signatures which serve us to define QoS-capable interface signatures. We also define incoming and outgoing interaction signatures based on the definition of QoS-definable interaction signatures. The purpose of section 6 is to define type checking rules associated with ODP computational interfaces based on the

analysis given in section 2. The main objectif of the current section is to show how typing rules on ODP interfaces can be defined in a coherent manner than it is in RM-ODP. All the OCL constraints presented in this section apply on UML constructs in the metamodel of figure 1. This section is also intended to be as a first step to defining QoS-capable interfaces typing rules. Section 7 provides the OCL specification of typing rules associated with computational interfaces. The metamodel used for the specification is the one given in section2. Finally, section 8 enhances those OCL constraints by applying them in the context of the QoS-aware concepts defined in section 5. We shall see how the OCL specification of QoS-aware typing rules is a more concise and precise specification of typing rules associated with ODP computational interfaces. A conclusion ends the paper.

2. Definitions of Interaction signatures Concepts

In this section we present the interaction signatures concepts as they are defined in the computational viewpoint.

A Computational interface template is an interface template for 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 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 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 definitions will serve us to discuss the remainder of the paper.

3. Unifying invocation signatures and announcement signatures

This section introduces the following section. It mainly shows interaction signatures can be defined and modelled in a precise and concise way.

When trying to formalize those concepts (introduced in the previous section), we have met with the issue concerning interaction signature concepts and how they are currently used and defined. In other works such as [16] discussions have focused on whether the action template concept lays on a syntactic level or a semantic one. Here, we do not take sight of these considerations as the solution we propose lies on a syntactic level. We analyze how all these concepts are linked to each other, and Bring a consistent description out from their definitions. Announcement signatures definition is clear and easy to understand when taken apart and separately from the other definitions. However, it becomes ambiguous when we shall join it to the definition of Interrogation signatures. This is due to the fact that the Invocation and announcement concepts are indistinguishable. interaction signatures other than Interrogations and announcement signatures are unambiguous. The new literal definition of Interrogation signatures is as follows : *Each interrogation signature comprises at least two action templates which are an invocation and its corresponding termination. An invocation can possibly have more than one associated termination. Invocations and terminations are action templates and they are statically described by their name and their number of parameters. Each parameter is characterized by its name and its type* .

Based on their definitions, announcement signatures and Interrogations signatures are two different concepts ; at least, conceptually distinct. An announcement signature is an action template ; so, it is formalized as in Figure 1. Now, Interrogation signatures do comprise action templates. Invocations and Terminations are also both kind of action templates ; and, since Invocations and Announcements describe the same concept from a practical point of view, it is preferable to merge them in one term. Thus, Invocations are now absorbed by Announcements, and, consequently, the Announcement term present both Invocation and Announcement concepts. The natural way to formalize them is as in Figure 1. Interrogation signatures comprise one and only one Invocation and to each Invocation there is a corresponding finite non-empty set of Terminations. Terminations are packed in an Interrogation signature. After we just come to put an Invocation into an Interrogation signature, their associated Terminations have to be joined to it, getting them packed into an Interrogation signatures (See Figure 1).

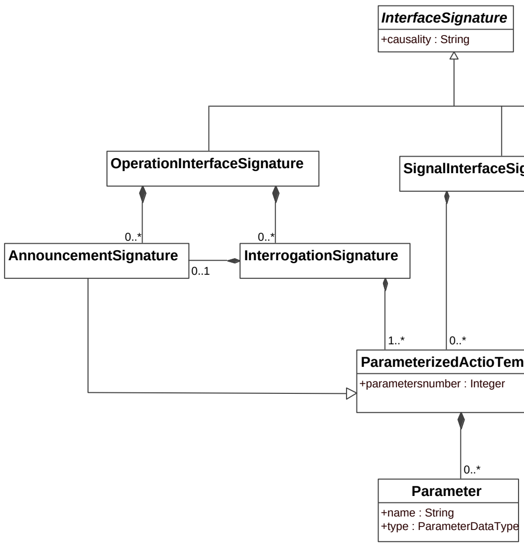

Figure 1. *Interface signatures Metamodel*

In this model (See Figure 1), Interrogation signatures comprise both Invocations and termination signatures. Each Interrogation signature contains one and only one Invocation and contains also a set of its corresponding Terminations which is not empty. This modelling choice has been influenced by two main factors. First, Invocation and Interrogation signatures are different from each others. Even if Interrogation signatures comprise one and only one Invocation, it is more convenient to distinguish them from each other, mainly from a conceptual point of view. Secondly, since an Invocation has a finite non-empty set of corresponding Terminations ; it is more convenient to aggregate them in Interrogation signatures rather than Invocations. This is mainly due to the fact that Interrogation signatures consist of both Invocations and their associated Terminations and thus, they conceptually belong to Interrogation signatures and should not be comprised by Invocations. On the other hand, both Invocations and Terminations are respectively absorbed by Announcements and action templates (Parameterized Action Template), and, by doing it this way, we must bear in mind that two UML terms representing Invocations and Terminations were moved away, and, thus, unloading the model from superfluous terms, especially for practical considerations. In this section, we have analysed the interaction signature concepts. We have detected inconsistencies in their verbal description. We have mainly shown interaction signatures need to be redefined in order to to correct their conceptual description. Finally, based on this new conceptual definition we have proposed an interface signatures metamodel.

In the next section we enhance the metamodel elaborated here. We first lead an algebraic analysis of interaction signatures concepts, then, based on the results of this analysis we elaborate a new interface signatures UML metamodel.

4. Introduction of *Procedural* Interface Signature & *Discrete* Interaction Signatures

The problem with all the difficulty in modeling interaction signatures is that the definitions of the concepts are not precise and leaves room for plenty of interpretations. To eliminate this ambiguity one have to analyze the definitions on a conceptual level in order to bring out the exact conceptual relationships semantics between those concepts.

In this section we show interactions are of two kinds : *discrete* (parameterized) interactions and flows. *discrete* interactions are composed by *primitive* and *compound* interactions, *primitives* which are *incoming* or *outgoing* interactions.
On the other hand, interface signatures are defined in the computational language in terms of three kind of computational interfaces. However when we analyze interface signatures concepts we show that in fact there are only two significant categories they are to be classified in. We shall see how interface signatures can principally be classified in two main classes, namely ; *Procedural* interface signatures and stream interface 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 mean 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$\backslash$B denotes the set of elements which are in A and are not in B.

– *bby* means *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\SA, SAann\SA, SAint\SA, SAter\SA have the same elements since SAinv\SA= SAann\SA = SAint\SA = SAter\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\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 *Discrete* interaction signature is an *Action Template* with a finite set of parameters as well as their numbers (See Figure 2).

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 : ***Discrete* interactions signatures** and flow interactions signatures.

2) Operation Interfaces signatures and Signal Interfaces signatures are composed *bby* ***Discrete* interaction signatures**.

3) A stream interface signature is composed *bby* a set of flow interactions signatures.

Definition 3 : A ***Procedural*** Interface Signature is an interface signature composed *bby* ***Parameterized* interactions signatures**(See Figure 2).

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

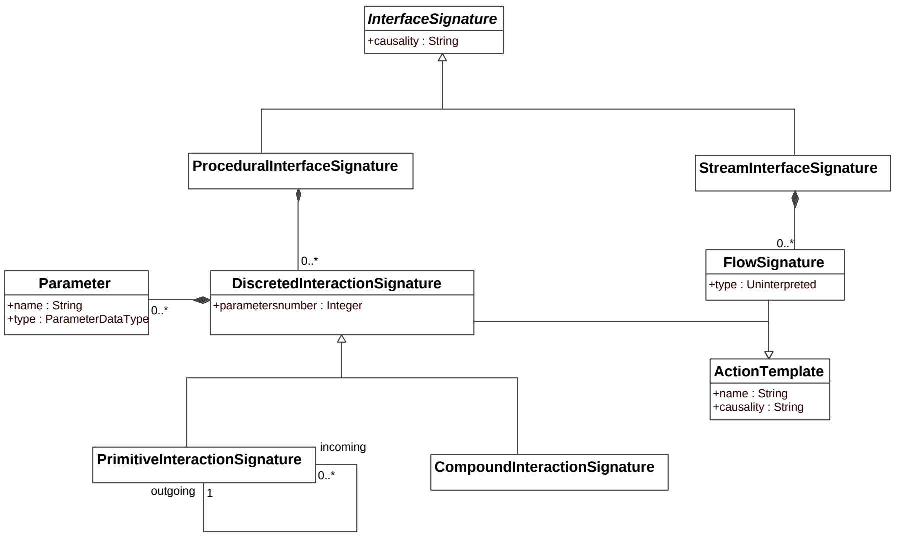

Figure 2. *Procedural interface signatures metamodel*

In this section we have introduced a precise definition of *Action Templates*. We have also defined two new concepts. First we have defined *Discrete* interaction signatures which we have used to define *Procedural* interface signatures. Based on those new concepts, we have elaborated a new metamodel of interaction and interface signatures.
In the following section we use the results of the present section in order to introduce new *QoS-aware* concepts to the computational viewpoint.

5. Definition and UML modeling of QoS-Capable interfaces and QoS-labeled interactions

Interactions in the computational language are of three kinds (signals, operations and flows). We have shown interactions are of two main kinds : *Discrete* and *Continuous* interactions. Signals in the computational language are defined as being atomic interactions that constitutes the building blocks of the other kinds of interactions. Indeed, 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.
Similarly, *discrete* interactions are classified in two main categories : *Primitive discrete* interactions (homologous of signals) and *compound discrete* interactions (homologous of operations). This classification is necessary 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).

In [17], authors specified those constraints based on their definitions provided in RM-ODP [3], then redefined[14],[15], those rules in one unified rule applied on *Procedural* interface signatures establishing a correspondance between *primitive* and *compound discrete* interactions. As a consequence, it is shown [14],[15], unification choices in modeling computational interfaces signatures and interaction signatures concepts help specifying compact OCL constraints applied on computational interfaces refinements which establishes a correspondance between *primitive* and *compound* interactions of *Procedural* computational interfaces, thus providing for the definition of end to end QoS characteristics, as well as allowing for different kind of computational interfaces to be bound (e.g *Procedural* to stream interfaces bindings).
In this section, we drop compound interactions in favor of primitive interactions in order to provide a QoS-*built-in* features computational metamodel. Indeed, instead of refining compound interactions into primitives, we model primitives as being the only interactions composing Procedural interfaces. In fact, every Procedural interface containing compound interactions can also be considered as being composed by primitives, since a compound interaction is an alliance of primitives. Having said that, an interaction between two or more procedural interfaces only consist of primitives being involved in the interaction process. Since primitives are now the only constituent interactions of procedural interfaces, end-to-end QoS characteristics become an intrinsic feature of Procedural interfaces, thus, there is no need for interaction refinement rules to be specified since interaction refinements for QoS ends are now statically (plugged in) within the computational metamodel. In doing so, refinement rules OCL specification have gone from the specification of two refinement rules in [17] to reducing it to only one in [14],[15], which is (refinement rule) now superfluous in the actual metamodel.
At this point, we are ready to provide the final interaction signatures computational metamodel, but before this we give the two following definitions which summarize what is explicited above.
Definition 4 :

A ***QoS-labeled*** interaction signbature is a *Primitive discrete* interaction signature (See Figure 3).

Definition 5 :

A ***QoS-embdded*** interface signature is a *Procedural* interface signature composed *bby* ***QoS-labeled*** interactions (See Figure 3) .

Operations in the computational language 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 the previous section we have shown invocations and announcements do play the same role conceptually and practically. In [14],[15], we have also shown *Primitive discrete* interaction signatures can be composed by two kinds of interactions : *outgoing* interactions and *incoming* interactions. That is, for every *outgoing* interaction corresponds a finite set (possibly empty) of *incoming* interactions and conversely, an *incoming* interaction corresponds to one and only one *outgoing* interaction(see figure **??**). However, neither outgoing nor incoming interactions have been defined in [14],[15].

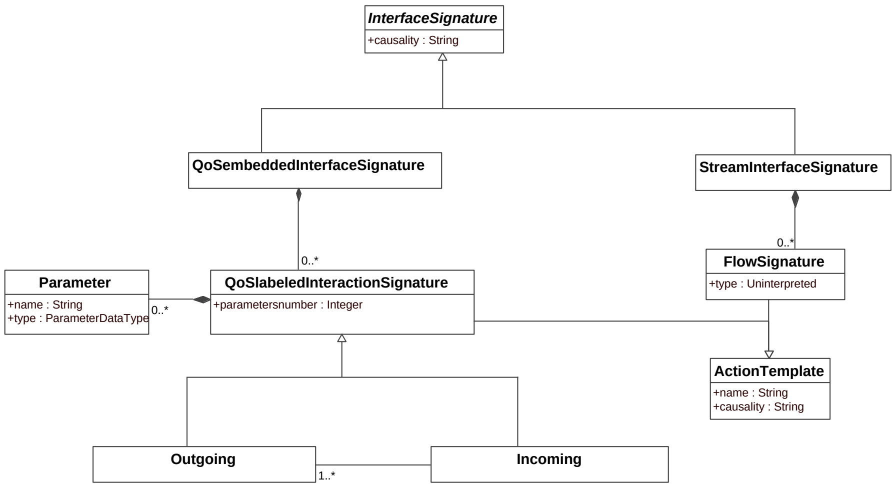

Figure 3. *Qos-embedded interface & QoS-labeled interaction signatures metamodel*

Before we define outgoing and incoming interactions we have to redefine causalities at both interface and interaction level. Causalities in the computational language are of three kinds corresponding to the types of both interfaces and the interactions they support. That is, *"client"* and *"server"* causalities, *"initiator"* and *"responder"* as well as *"producer"* and *"consumer"*, respectively for operational interfaces and operations, signal interfaces and signals and finally, stream interfaces and flows.

Thus, causalities follow the causal-effect principle. Consequently, we can reduce all kind of causalities in the computational language to only two kinds. In fact, *"client"* , *"initiator"* and *"producer"* causalities are reduced to ***"actor"*** causality as well as *"server"*, *"responder"* and *"consumer"* causalities are abstracted to ***"reactor"*** causality. Having said that, computational interfaces as well as interactions defined in are reduced to acting and reacting entities.

Since outgoing interactions go from a causal interface out to a receipient interface and conversely incoming interactions come from a causal interface into a receipient one, we define outgoing and incoming interfaces as follows :

Definition 6 :

An ***outgoing*** interaction signature is a *QoS-labeled* interaction signature going from an *acting QoS-embedded* interface out to a *reacting QoS-embedded* interface (See Figure 3).

Definition 7 :

An ***incoming*** interaction signature is a *QoS-labeled* interaction signature coming from a *reacting QoS-embedded* interface into an *acting QoS-embedded* interface (See Figure 3).

In this section we have defined four new concepts. We have introduced *QoS-labeled* interaction signatures which served us to define *QoS-embedded* interface signatures. Then we defined incoming and outgoing interaction signatures based on the definition of *QoS-labeled* interaction signatures. We have also elaborated a new metamodel specifying all the new concepts introduced here. Having done that, the final metamodel is a consistent QoS-aware formalization of ODP computational interfaces & interaction signatures.

6. OCL Typing Rules Re-Definition

The purpose of this section is to define type checking rules associated with ODP computational interfaces based on the analysis given in section 2. The main objectif of the current section is to show how typing rules on ODP interfaces can be defined in a coherent manner than it is in RM-ODP. All the OCL constraints presented in this section apply on UML constructs in the metamodel of figure 1. This section is also intended to be as a first step to defining *QoS-embedded* interfaces typing rules, and thus, showing how we can specify more precisely and easily OCL constraints related to ODP computational interfaces *QoS-embedded* typing rules.

In this section we specify semantics of interaction signatures related to subtyping rules. We rewrite those literal rules and present them under a new form. First, we give the rules as they are presented in the ODP computational language, and, then provide a clearer and compact description of those rules. We shall just concentrate on Interrogation signatures as the other rules are already compact and easy to understand. Typing rules in the computational language corresponding to Interrogation signatures are defined 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 in X (the corresponding signature in X) which defines an Interrogation with the same name.

– For each Interrogation signature in Y, the corresponding Interrogation signature in X has the same number and names of parameters.

– For each Interrogation signature in Y, every parameter type is a subtype of the corresponding parameter type of the corresponding Interrogation signature in X.

– The set of termination names of an Interrogation signature in Y contains the set of termination names of the corresponding Interrogation signature in X.

– For each Interrogation signature in Y, a given termination in the corresponding Interrogation signature in X has the same number and names of result parameters in the termination of the same name in the Interrogation signature in Y.

– For each Interrogation signature in Y, every result type associated with a given termination in the corresponding Interrogation signature in X is a subtype of the result type (with the same name) in the termination with the same name in Y.

– For every announcement in Y, there is an announcement signature in X (the corresponding signature in X) which defines an announcement with the same name.

– For each announcement signature in Y, the corresponding announcement signature in X has the same number and names of parameters.

– For each announcement signature in Y, every parameter type is a subtype of the corresponding parameter type in the corresponding announcement signature in X.

As we look at those literal constraints provided by the ODP computational language, we realize they can be aggregated in a more compact description and especially for Operation interface signature typing rules. When we shall give an equivalent description of the ones prescribed by the computational language, we can see how easy their specification becomes.

Having said that, the new form these definitions are rewritten in is 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.

For signal interface types that are not defined recursively, the rules [3] are summarized below. Signal interface signature type X is a subtype of signal interface signature type Y if the conditions below are met :

– For every initiating signal signature in Y there is a corresponding initiating signal signature in X 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.

– For every responding signal signature in X there is a corresponding responding signal signature in Y 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.

Now, that we have reorganized the verbal description of these rules in a compact form, we realize they do share the same description pattern. Indeed, interaction signatures which are related by a Type/Subtype relation must have the same names, the same names and numbers of parameters, the latter having to satisfy a Type/Subtype relation. We can break these rules in order to bring out OCL sub-expressions which can be used in the context of all kinds of interaction signatures. We shall exploit these similarities between these definitions and come up with general formal expressions which can be used to specify OCL constraints on all interaction signatures. OCL 2.0 provides the means to realize this.

Identification of OCL sub-expressions

The different type checking rules related to computational interfaces contain similarities in their literal definitions. That is, all interaction signatures (but flow signatures) of all computational interfaces which are related by a Type/Subtype relation must have the same names, the same names and numbers of parameters and that parameters have to satisfy a Type/Subtype relation. Since all interaction signatures (but flow signatures) derive from the Parameterized Action Template term, we explore this fact in order to specify those similarities mentioned above in the context of the Parameterized Action Template classifier. In what follows we give the identified OCL sub-expressions to be used in Typing/Subtyping relation specification constraints :

Context ParameterizedActionTemplate **inv :**

def : hasSameName(PAT : ParameterizedActionTemplate) : Boolean = (self.name= PAT.name)

def : hasSameParametersNumber(PAT : ParameterizedActionTemplate) : Boolean = (self.parameternumbers= PAT. parameternumbers)

def : hasSameParametersNames(PAT : ParameterizedActionTemplate) : Boolean = self.Parameter $\rightarrow$ forAll(Px : Parameter | ParameterizedActionTemplate $\rightarrow$ Exists(Py : Parameter | Px.name= PAT.Py. name))

def : isSubTypeOf (PAT : ParameterizedActionTemplate) : Boolean = self.Parameter $\rightarrow$ forAll(Px : Parameter | ParameterizedActionTemplate $\rightarrow$ Exists(Py : Parameter | PAT.Py.oclIsKindOf(Px)))

def : isOfCausality(c : String) :Boolean= self.causality=c

At this point, given the OCL expressions above, we are ready to specify OCL constraints related to typing rules associated with computational interfaces. The metamodel used for the specification is the one figure 1.

7. OCL Typing Rules Specification

7.1. Signal Interface signatures Subtyping Rules OCL Specification

The RM-ODP definition of signal interface subtyping rules was given in the previous section.This constraint is described using OCL as follows :

Context SignalInterfacesignature **inv :**
SignalInterfacesignature.allInstances $\rightarrow$
forAll(X,Y | ParameterizedActionTemplate.allInstances $\rightarrow$
forAll(PY | ParameterizedActionTemplate.allInstances $\rightarrow$
exists(PX |
PX.isOfCausality('initiate')
and
PY.isOfCausality('initiate')
and
Y.PY.hasSameName(X.PX)
and
Y.PY.hasSameParametersNumbers(X.PX)
and

Y.PY.hasSameParametersNames(X.PX)
and
X.PX.isSubTypeOf(Y.PY))))

and

ParameterizedActionTemplate.allInstances→forAll(PX |
ParameterizedActionTemplate.allInstances→
exists(PY |
PX.isOfCausality(’respond’)
and
PY.isOfCausality(’respond’)
and
Y.PY.hasSameName(X.PX)
and
Y.PY.hasSameParametersNumbers(X.PX)
and
Y.PY.hasSameParametersNames(X.PX)
and
Y.PY.isSubTypeOf(X.PX)))

implies

X.oclIsKindOf(Y)

7.2. Operation Interface signatures Subtyping Rules OCL Specification

The rules for Operation interface types that are not defined recursively were given in the previous section. This constraint is described using OCL as follows :

Context OperationInterfacesignature **inv :**
OperationInterfacesignature.allInstances → forAll(X,Y |
(Interrogationsignature.allInstances → forAll (Iy |
Interrogationsignature.allInstances → exists(Ix |
Announcementsignaturee.allInstances → forAll (Ay |
Announcementsignature.allInstances → exists(Ax |
Y.Iy.Ay.hasSameName(X.Ix.Ax) and
Y.Iy.Ay.hasSameParametersNumbers(X.Ix.Ax)
and
Y.Iy.Ay.hasSameParametersNames(X.Ix.Ax)
and
X.Ix.Ax.isSubTypeOf(Y.Iy.Ay))))))

and

(Interrogationsignature.allInstances → forAll(Iy |
Interrogationsignature.allInstances→exists(Ix |
ParameterizedActionTemplate.allInstances→forAll (Ty |
ParameterizedActionTemplate.allInstances → exists(Tx |
Y.Iy.Ty.hasSameName(X.Ix.Tx)
and
Y.Iy.Ty.hasSameParametersNumbers(X.Ix.Tx)
and
Y.Iy.Ty.hasSameParametersNames(X.Ix.Tx)
and
Y.Iy.Ty.isSubTypeOf(X.Ix.Tx))))))

and

(Announcementsignature.allInstances→forAll(Ay |
Announcementsignature.allInstances→exists(Ax |
Y.Iy.Ay.hasSameName(X.Ix.Ax)
and
Y.Iy.Ay.hasSameParametersNumbers(X.Ix.Ax)
and
Y.Iy.Ay.hasSameParametersNames(X.Ix.Ax)
and
X.Ix.Ax.isSubTypeOf(Y.Iy.Ay))))

implies

X.oclIsKindOf(Y)

7.3. Stream Interface signatures Subtyping Rules OCL Specification

Stream signature subtyping rules depend upon the details of the interactions abstracted in the definition of the stream interfaces concerned. In particular these details will clarify whether or not the subtyping rules will permit incomplete correspondences between the set of flows in the two interfaces. For stream interface types that are defined recursively, the constraints are summarized below. Stream interface X is a signature subtype of stream interface Y if the conditions below are met for all flows which have identical names : If the causality is producer, the information type in X is a subtype of the information type in Y. If the causality is consumer, the information type in Y is a subtype of the information type in X. This constraint is described using OCL as follows :

Context StreamInterfacesignature **inv :**

StreamInterfacesignature.allInstances→forAll(X,Y |
(Flowsignature.allInstances *to* forAll(Fxp,Fyp |

```
Fxp. isOfCausality('produce')
and
Fyp.isOfCausality('produce')
and
X.Fxp.hasSameName(Y.Fyp)
implies
X.Fxp.type.oclIsKindOf(Y.Fyp.type)))

   and

   (Flowsignature.allInstances to forAll(Fxp,Fyp |
Fxp.isOfCausality('consume')
and
Fyp.isOfCausality('consume')
and
X.Fxp.hasSameName(Y.Fyp)
implies
Y.Fyp.type.oclIsKindOf(X.Fxp.type))))

   implies

   X.oclIsKindOf(Y)
```

In this section we have specified OCL constraints related to typing rules associated with ODP interfaces. Since a direct OCL specification of those rules is a complicated task, we have decomposed those rules into elementary rules easily specified in OCL. Then, we have constructed OCL constraints corresponding to type checking rules by combining the elementary OCL sub-expression identified. However, the final OCL constraints provided can be presented in a more concise way. The purpose of the next section is to enhance those OCL constraints by applying them in the context of the QoS-aware concepts defined before. We shall see how the OCL specification of QoS-aware typing rules is a more concise and precise specification of typing rules associated with ODP computational interfaces.

8. Definition and OCL specification of Type checking rules on ODP QoS-embedded computational interfaces

In the continuity of the previous section this section represents an enhancement of typing rules definitions and specification given in the previous section. Thus, we specify semantics of interaction signatures related to QoS-embedded computational interfaces subtyping rules. The objectif of this section is to show how can we specify OCL constraints associated to ODP typing rules in a more precise and compact way. It is intended to show that our conceptual choices provides us with clear and precise concepts which are easy to specify.

Interrogations (invocations in the context of type checking rules definitions) and announcements are *outgoing* interactions. On the other hand, terminations are *incoming* interactions we can redefine those rules which will hold in only two rules :

Procedural 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.

RM-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 *QoS_labeledInteractionSignature* **inv :**

def : *hasSameName(QoS_LIS : QoS-labeledInteractionSignature) : Boolean =*
self.name= QoS_LIS.name)

def : *hasSameParametersNumber(QoS_LIS : QoS-labeledInteractionSignature) : Boolean =*
(self.parameternumbers= QoS_LIS.parameternumbers)

def : *hasSameParametersNames(QoS_LIS : QoS-labeledInteractionSignature) : Boolean =*
self.Parameter to forAll(Px : Parameter |
QoS-labeledInteractionSignature to Exists(
Py : Parameter |
Px.name = QoS_LIS.Py. name))

def : *isSubTypeOf (QoS_LIS :QoS-labeledInteractionSignature) : Boolean =*
self.Parameter to forAll(Px : Parameter |
QoS-labeledInteractionSignature to Exists(

Py : Parameter |
QoS_LIS.Py.type.oclIsKindOf(Px.type)))

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

Context *QoS-embeddedInterfaceSignature* **inv :**

Let *QoS_out : QoS_labeledInteractionSignature =*
QoS_labeledInteractionSignature.oclAsType(
outgoing)

Let *QoS_in : QoS-labeledInteractionSignature =*
QoS_labeledInteractionSignature.oclAsType(
incoming)

QoS-embeddedInterfacesignaturetoforAll(X,Y |

(QoS_out.outgoing →forAll(PY |
QoS_out.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

(QoS_in.incoming toforAll(PY |
QoS_in.incoming toexists(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, their corresponding typing rules OCL specification is not given here, since they are provided in [11][**?**].

9. Conclusion

In the present work, we specified OCL constraints associated to type checking rules for distributed applications. In the first main part of this work, we analyzed the interaction signatures concepts. We then raised inconsistencies in their verbal description ; and finally provided an UML model of those concepts. We have already came across those inconsistencies in other works [**?**],[10] [11], and have provided reliable solutions to those issues mainly from a conceptual point of view. Here, we proposed new solutions based on a different analytical approach. The result is a consistent formal model of the interaction signature concepts. Then, we specified in OCL, semantics of interaction signatures relating to subtyping rules. We showed that those literal rules provided by the ODP computational language can be aggregated in a more compact definition. We then reorganized them and gave an equivalent description in a clearer manner. After we have done that, we enhanced the computational metamodel elaborated before. That was achieved by the introduction and UML formalization of the *Functional* computational interface, ***QoS-definable interactions*** and ***QoS-capable interfaces*** concepts. Finally we defined type checking rules related to ***QoS-definable interactions*** and ***QoS-capable interfaces*** and specified them using OCL 2.0.

.

10. Annexe

In Cari'08 proceedings we have proposed a new model of ODP interaction signatures. In fact we have corrected, the conceptual relationship between Action Templates and interaction signatures. The new conceptual relationship between Action Templates and interaction signatures allowed us to define consistent type checking rules on interaction signatures. Based on the new rules we have specified typing relationships between computational interfaces.

The present extension of the work summarized above provide a new conceptual relationship between Action Templates and interaction signatures. Indeed, we provide algebraic proofs that justify our proposals. The results of this Algebraic analysis is introduction of new kind of interactions (Discrete interactions) and consequently a new type of ODP computational interfaces (Procedural interfaces). A second contribution we have introduced in the present work is enhancing ODP discrete interactions procedural interfaces. Indeed we introduce additional concepts and specify them using UML2.0 and OCL 2.0. Finally we define type checking rules on the new proposed concepts. Based on these we provide trhe final metamodel which is considered as a QoS-aware computational metamodel.

11. Bibliographie

[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 CD 10746-2*, 1994.

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

[4] R. ROMEO ET AL., , « Modelling the ODP Computational Viewpoint with UML 2.0 », *IEEE International Enterprise Distributed Object Computing Conference*, 2005.

[5] D.H.AKEHURST ET AL, , « Addressing Computational Viewpoint Design », *Seventh IEEE International EDOC, IEEE Computer Society*, 2003.

[6] BEHZAD BORDBAR ET AL, , « Using UML to specify QoS constraints in ODP », *Computer Networks Journal pp.279-304*, 2002.

[7] R. ROMERO ET AL, , « Action templates and causalities in the ODP computational viewpoint », *WODPEC'04 pp. 23-27*, 2004.

[8] J. RUMBAUGH AND AL, , « OMG Document ptc/03-10-14 », *Addison Wesly*, 2003.

[9] B.EL OUAHIDI ET AL, , « Interaction signatures and Action Templates in The ODP Computatinal Viewpoint », *Proceedings of the 6th WSEAS International SEPADS'07, Corfu Greece, Feb 16-19*,2007.

[10] B.EL OUAHIDI ET AL, , « Towards Refinement of The ODP Computational Viewpoint Interaction signatures », *WSEAS Transactions On Telecommunications Journal, pp 601-606*, May 2007.

[11] B.EL OUAHIDI ET AL, , « Interaction signatures and Action Templates in The ODP Computatinal Viewpoint », *Proceedings of the 6th WSEAS International SEPADS'07, Corfu Greece, Feb 16-19*,2007.

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

[13] OMG, , « UML 2.0 OCL Final Specification », *OMG Document ptc/03-10-14*,2003.

[14] B.EL OUAHIDI ET AL, , « On UML Modeling of Computational Interfaces & Interactions in the UML4ODP Computational Language », *In Proceedings of the 12th WSEAS International multiconference, Advances in computers, CSCC'08, Crete Island*, July 23-25, Greece, 2008.

[15] B.EL OUAHIDI ET AL, , « UML4ODP : OCL 2.0 Constraints Specification & UML Modeling of Interfaces in the Computational Metamodel », *Accepted in WSEAS Transactions on Computers international Journal*, , February 20, 2009.

[16] R. ROMEO ET AL, , « Action Templates and Causalities in the ODP Computational Viewpoint », *1St International Workshop on ODP in the Enterprise Computing (WODPEC), Monterey, California USA pp. 23-27* , 2004.

[17] O. REDA ET AL, , « Towards a Refinement of the Open Distributed Systems Interactions Signatures », *1WSEAS transactions on communications, vol. 6, pp. 601-607* , Apr 2007.