Transformations and Consistent Semantics for ODP Viewpoints*

C. Bernardeschi^a, J. Dustzadeh^b, A. Fantechi^c, E. Najm^b, A. Nimour^b, F. Olsen^{b,d}

*(a) Dipartimento di Ingegneria della Informazione, Università di Pisa
Via Diotisalvi 2, 56100 Pisa, Italy*

*(b) Département Réseaux, Ecole Nationale Supérieure des
Télécommunications
46, Rue Barrault, F-75013, Paris, France*

*(c) Dipartimento di Sistemi e Informatica, Università di Firenze
via S. Marta 3, 50139 Firenze, Italy*

*(d) PAA/TSA/TLR, France Telecom — CNET
38/40 rue Général Leclerc, 92794 ISSY MOULINEAUX CEDEX 9,
FRANCE*

Abstract

We discuss correctness preserving transformations of specifications in the passage from the Information viewpoint to the Computational viewpoint of the ODP ISO reference model. In our transformation exercise we use two relatively simple languages for each of these viewpoints; a class-oriented language for the Information viewpoint with the separation of actions from classes as the main distinguishing feature. For the Computational viewpoint the language chosen is a variant of the Actor language.

Keywords: ODP, Information Viewpoint, Computational Viewpoint, Correctness Preserving transformations, Actors, Objects, Object-Oriented Methods, Process Calculii.

*Supported by the Galileo project.

1 INTRODUCTION

The ISO reference model for Open Distributed Processing (ODP) introduces the concept of *viewpoint* to describe the system from a particular set of concerns (Farooqui et al. 1995). Five different viewpoints of the system have been introduced: “Enterprise” (requirement capture and early design of distributed systems), “Information” (conceptual design and information modeling), “Computational” (Software design and development), “Engineering” (system design and development) and “Technology” (technology identification, procurement and installation). Together, these different viewpoints allow a complete description of the system.

Formal methods provide a mathematical way to specify and analyze the behavior of concurrent and distributed systems. In the development of a system design trajectory supported by formal methods, transformations are usually defined in order to aid the development of correct specifications.

This is preliminary work in the direction of defining correctness preserving transformations from the Information viewpoint to the Computational viewpoint of the ODP reference model. In the Information viewpoint, a system is abstractly specified by giving the actions the system may perform; in the Computational viewpoint, a more concrete specification of the behavior of the system is given. In particular, the ODP computational model is an object-based framework. Its object-oriented nature emphasizes abstraction and encapsulation, while no assumption is made on the internal structure of objects or on linguistic support for objects. Since no constraint is placed on intra object activities, any type of intra object concurrency is allowed. Inter object concurrency is instead specified in the ODP computational model (Najm et al. 1995).

Due to their different nature, the passage from the Information viewpoint to the Computational viewpoint is a critical point in the system development methodology and it has been recognized as a possible area of intervention of transformations of specifications.

Information objects are mapped to computational object templates, and the abstract, collective behaviour of objects, as described by Information Actions, is mapped to message passing between autonomous interacting computational objects.

Structure of the paper. The core of the paper is section 4 where we present our proposed transformations from the Information viewpoint to the Computational viewpoint. Before that however, we describe the source language for the transformations in section 2 and the target language in section 3. The paper ends with an outlook for future work, comparisons with existing work and some concluding remarks.

2 INFORMATION VIEWPOINT: THE SOURCE LANGUAGE

2.1 Informal description

The language chosen for the Information viewpoint is a simple class-oriented language, where a system is defined as a collection of objects with attributes, but without methods, and a collection of actions that can be performed on the objects. The actions, contrary to methods, are perceived by the user as atomic, and operate on the object states as if acting on a single global state space.

2.2 Syntax

The syntax of the source language is given in syntax diagram 1 using a BNF notation, with some freedom for simplicity reasons. The keywords are given in bold.

```

ClassDef ::= class_id = Class
Class    ::= class
           att1:Type1;
           ...
           attn:Typen;
           endclass
ActionDef ::= action_id(var1:Type1, ..., varn:Typen):Type = Action
Action    ::= action Decl Body endaction
Decl      ::= var:Type; | Decl1 Decl2
Body      ::= case Exp1: S1; return Exp'1;
           ...
           case Expn: Sn; return Exp'n;
S          ::= var := Exp; | var.att := Exp;
           if Exp then S1 else S2 endif;
           S1 S2
Exp        ::= c | var | var.att | nil
           | op(Exp1, ..., Expn) | action_id(Exp1, ..., Expn)
           | new class_id(Exp1, ..., Expn)
Type       ::= ground_type | class_id

```

Syntax 1: The Source Language.

The terminals have the following meaning:

At this point it is important to make some comparisons between this language and other languages proposed for the Information viewpoint.

abbreviation	meaning
class_id	Class identifier
action_id	Action identifier
var	variable identifier
att	Attribute name
op	Operation on ground types
c	Constant

Of the formal languages, the most common of these is Z, see for ex. (Derrick et al. 1996) which discusses the use of Z in relation to transformations and viewpoint consistency. Our language is more concrete than Z — it is more close to an implementation language. We can imagine using it as the final refinement at the Information viewpoint. We could start with an abstract Z specification and end up with our language as the last description before going to the Computational viewpoint.

Of the object-modelling notations, we compare with OMT and Fusion. In comparison with OMT, our language is very close to OMT's static object model. The difference is first that actions (methods) are not encapsulated in classes at this level; this is what Fusion does. The second difference is that we do not provide explicit associations — instead associations are implicit via attributes referencing other classes.

2.3 Example

Let us consider the following simple example of an Information viewpoint specification of part of a bank account management system, made of a single object and a single action:

```
ACCOUNT = class owner: string;
           balance: integer;
        endclass

transfer(provider:account; recipient:account; amount:positive) =
(* the action transfers a sum equal to amount from the provider account *)
(* to the recipient account *)
action

case provider.balance < amount:      return NACK;

case provider.balance >= amount:
    provider.balance := provider.balance - amount;
    recipient.balance := recipient.balance + amount;
return ACK;
end action
```

2.4 Semantics

We give here the dynamic semantics of our source language. We suppose at this level that the programs are syntactically correct and type-checked.

The dynamic semantics is given using the following judgments:

- $E \models (Decl \Rightarrow E')$ "In the environment E the declaration $Decl$ gives the environment E' " ;
- $E \models (S \Rightarrow E')$ "In the environment E the execution of the statement S gives the environment E' " ;
- $E \models (Exp \rightarrow v) \Rightarrow E'$ "In the environment E the evaluation of the expression Exp returns the value v and gives the new environment E' ."

The values we consider here are predefined ground values and record references representing behavior-less objects. Each record is of the form:

$$< att_1 = v_1, \dots, att_n = v_n >$$

where the v_i are also values. We consider also a nil value, noted $\perp$, that is polymorphic.

An environment is an ordered collection of pairs "variable/value" noted: $var = v$. A variable name may appear in more than one pair in an environment. The extension of the environment E with the pair $var = v$ gives a new environment denoted: $E, var = v$. We define a lookup function that gives for a given variable the most recent binding:

$$lookup((E, var = v), var') = \begin{cases} v & \text{if } var \equiv var' \\ lookup(E, var') & \text{otherwise} \end{cases}$$

Similarly, for the notation $E, var = v[var' \leftarrow v]$ we define the function:

$$E, var = v[var' \leftarrow v] = \begin{cases} E, var = v' & \text{if } var \equiv var' \\ E[var' \leftarrow v] & \text{otherwise} \end{cases}$$

The dynamic semantics is then given by the following SOS rules:

Declarations:

$$E \models (var : Type) \Rightarrow E, var = \perp$$

A simple declaration extends its evaluation environment with the new declared variable bound to $\perp$.

$$\frac{E_1 \models Decl_1 \Rightarrow E_2 \ ; \ E_2 \models Decl_2 \Rightarrow E}{E_1 \models Decl_1 Decl_2 \Rightarrow E}$$

Two consecutive declarations extend twice the initial environment.

Action body:

$$\begin{array}{c}
E \models (Exp_j \rightarrow \text{FALSE}) \Rightarrow E, \forall j < i \\
E \models (Exp_i \rightarrow \text{TRUE}) \Rightarrow E \\
E \models S_i \Rightarrow E_i \\
E_i \models (Exp'_i \rightarrow v) \Rightarrow E'_i \\
\hline
E \models \left(\left(\begin{array}{c} \text{case } Exp_1 : S_1 ; \text{ return } Exp'_1 ; \\ \vdots \\ \text{case } Exp_n : S_n ; \text{ return } Exp'_n ; \end{array} \right) \rightarrow v \right) \Rightarrow E'_i
\end{array}$$

The conditions (Exp_i) of the case statement are evaluated. The branch corresponding to the first Exp_i evaluated to TRUE is executed. Note that the evaluation of the Exp_i has no side effects on the environment.

Assignments:

$$\frac{E \models (Exp \rightarrow v) \Rightarrow E'}{E \models (var := Exp) \Rightarrow E[var \leftarrow v]}$$

To assign an expression (Exp) to a variable (var), we first evaluate Exp and then change the binding of var to this value.

$$\frac{E \models (Exp \rightarrow v) \Rightarrow E'}{E \models (var.att := Exp) \Rightarrow E[var.att \leftarrow v]}$$

Similarly, to assign an expression to an object attribute, we evaluate this expression and then we update the attribute of the corresponding record.

IF statements:

$$\frac{E \models (Exp \rightarrow \text{TRUE}) \Rightarrow E' ; E' \models S_1 \Rightarrow E''}{E \models \text{if } Exp \text{ then } S_1 \text{ else } S_2 \text{ endif} \Rightarrow E''}$$

$$\frac{E \models (Exp \rightarrow \text{FALSE}) \Rightarrow E' ; E' \models S_2 \Rightarrow E''}{E \models \text{if } Exp \text{ then } S_1 \text{ else } S_2 \text{ endif} \Rightarrow E''}$$

If the condition expression is evaluated to TRUE (respectively, to FALSE) we execute the "then branch" (respectively, the "else branch").

Statement chaining:

$$\frac{E \models S_1 \Rightarrow E_1 ; E_1 \models S_2 \Rightarrow E_2}{E \models S_1 S_2 \Rightarrow E_2}$$

The second statement is executed in the environment obtained after the execution of the first statement.

Constants:

$$E \models (c \rightarrow c) \Rightarrow E$$

A ground constant evaluates to itself.

Variables values:

$$E \models (var \rightarrow lookup(E, var)) \Rightarrow E$$

We have to search the environment for the last binding of *var*.

Function calls:

$$\frac{E \models (Exp_1 \rightarrow v_1) \Rightarrow E; \dots; E \models (Exp_n \rightarrow v_n) \Rightarrow E}{E \models (op(Exp_1, \dots, Exp_n) \rightarrow eval(op(v_1, \dots, v_n))) \Rightarrow E}$$

We first evaluate the arguments of the function and then call the predefined evaluation function on ground expressions: *eval*. All the arguments are evaluated in the same environment.

Action calls:

$$\frac{\begin{array}{l} E \models (Exp_1 \rightarrow v_1) \Rightarrow E; \dots; E \models (Exp_n \rightarrow v_n) \Rightarrow E \\ A(var_1 : Type_1, \dots, var_n : Type_n) \overset{\Delta}{=} ActionB \\ E, var_1 = v_1, \dots, var_n = v_n \models (ActionB \rightarrow v) \Rightarrow E' \end{array}}{E \models (A(Exp_1, \dots, Exp_n) \rightarrow v) \Rightarrow E'}$$

As for ground functions calls, we first evaluate the arguments and then evaluate the body of the action in an environment where the formal parameters have been bound to the actual ones.

Attribute selection:

$$E \models (var.att \rightarrow lookup(E, var).att) \Rightarrow E$$

We look for the value of *var* in the environment and then select the appropriate attribute.

Object creation:

$$\frac{\begin{array}{l} E \models (Exp_1 \rightarrow v_1) \Rightarrow E; \dots; E \models (Exp_n \rightarrow v_n) \Rightarrow E \\ Class \overset{\Delta}{=} \text{class } att_1 : Type_1, \dots, att_n : Type_n \text{ endclass} \end{array}}{E \models (\text{new } Class(Exp_1, \dots, Exp_n) \rightarrow \langle att_1 = v_1, \dots, att_n = v_n \rangle) \Rightarrow E}$$

The expressions Exp_i are evaluated in the same environment then a new record is created according to the definition of *Class* and initialized with the values of $Exp_1, \dots, Exp_n$. The returned value is this record.

3 COMPUTATIONAL VIEWPOINT: THE TARGET LANGUAGE

The target language for the computational viewpoint is an object language related to the *Actor* paradigm (see (Agha 1986) for an introduction to Actors) and to mobile process algebras (see (Pierce et al. 1995) and (Milner et al. 1992) for information on mobile process algebras and concurrent objects). The syntax is given in syntax diagram 2 and some examples in section 4 (where we discuss the transformations).

```

ActorDef ::= actor _name(var1, ..., varn) = Behavior
Behavior ::= [m1(ṽar1) -> Behavior1,
              ...,
              mk(ṽark) -> Behaviork]
              | Actor.m(Val1, ..., Valn); Behavior
              | if val then Behavior1 else Behavior2
              | become actor _name(Val1, ..., Valn)
              | create actor _name(Val1, ..., Valn); Behavior
              | let var = val in (Behavior)
Actor      ::= self | var
Val        ::= g_Val | var | nil | self
              | new actor _name(Val1, ..., Valn)
G_Val      ::= c | op(G_Val1, ..., G_Valn)

```

Syntax 2: Target Language

It is not our purpose to completely and formally define the semantics of our language here (see (Agha et al. 1993) or (Agha 1986) for semantics of Actors). In short, the addressing part is based on Actors and the computational part on process algebras. Active objects (actors) have unique identity (mail address). Objects communicate using asynchronous message passing and each object has its own, private queue of incoming messages. These messages are treated in the order of arrival (i.e. the queue is FIFO).

Referring to the syntax diagram we explain the main distinguishing features of the language:

- *message reception*

```

[ m1(ṽar1) -> Behavior1,
  ...,
  mk(ṽark) -> Behaviork ]

```

Wait for one of the messages $m_1, \dots, m_k$, and react by executing the corresponding *Behavior*.

- *message send*

```
Actor.m(val1, ..., valn); Behavior
```

Send message m to *Actor* with arguments $val_1, \dots, val_n$ and become *Behavior*.

- *replacement behaviour*

```
become actor _name(Val1, ..., Valn)
```

The current actor behaviour acts as specified in *actor_name*.

- *Actor creation*

```
create actor _name(Val1, ..., Valn); Behavior
```


Create a new actor that behaves as specified in *actor_name* and then act according to *Behavior*.

4 TRANSFORMATION

4.1 Style of the Transformation

A class is transformed into an actor whose acquaintances (parameters) are the attributes of the class. For each attribute correspond *get* and *set* methods. An action is transformed into an actor with only one method. This method is used to create a thread responsible for executing the action. The main difference added by the transformation is that access to or modification of the object state, which is direct in the source language, is possible only through the *get* and *set* methods in the target language.

4.2 A transformation example

To illustrate the transformation we come back to the example given in section 2.3. The transformation would produce the following pair of actors, one corresponding to the **ACCOUNT** class and one corresponding to the **transfer** action.

```

Act_Account(owner, balance) =
  [get_owner(ret) ->
    ret.value(owner);
    become Account(owner, balance),
  get_balance(ret) ->
    ret.value(balance);
    become Account(owner, balance),
  set_owner(new_owner) ->
    become Account(new_owner, balance),
  set_balance(new_balance) ->
    become Account(owner, new_balance)
  ]
Act_Transfer() =
  [transfer(provider, recipient, amount, return) ->
    create Thread_Transfer(provider, recipient, amount, return);
    become Act_Transfer()
  ]
Thread_Transfer(provider, recipient, amount, return) =
  provider.get_balance(self);
  [value(p_balance) ->
    if p_balance < amount then return.value(NACK)
    else recipient.get_balance(self);
  ]

```

```

    [value(r_balance) ->
      provider.set_balance(p_balance-amount);
      recipient.set_balance(r_balance+amount);
      return.value(ACK)
    ]
  ]

```

4.3 Definition and formalization of the transformation

We describe here a syntactic transformation from our information viewpoint language to the computational language. This transformation is a function, noted $\mathcal{T}$, that takes a well formed program text of the source language and returns a program text of the target language that has the same functionality. Function $\mathcal{T}$ is defined by induction on the structure of the source language.

Class definition: each class definition of the source language is translated into an actor definition in the target language.

A class C is transformed into an actor Act_C . The attributes of C become parameters of Act_C and for each attributes, att_i , there is a method named “get_” att_i ” that returns the value of att_i to the return address and a method named “set_” att_i ” that update the value of att_i with the specified value.

Action definition: each action becomes a stateless actor with a single method.

An action A is transformed into an actor Act_C whose behavior is to repeat-

edly wait for reception of the message A and then creates a new actor (bound to “Thread_A”) which is responsible for executing the action.

Action behavior: the translation is straightforward. Note that the body of an action is evaluated in a context where the declared variables have been introduced by a “let” statement.

$$\boxed{\mathcal{T}[\text{Decl Body}]} \longrightarrow \boxed{\text{let } \mathcal{T}[\text{Decl}] \text{ in } \mathcal{T}[\text{Body}]}$$

Declarations: Consecutive declarations are transformed into nested “let” statements.

$$\boxed{\mathcal{T}[\text{Decl}_1 \text{ Decl}_2]} \longrightarrow \boxed{\mathcal{T}[\text{Decl}_1] \text{ in } (\text{let } \mathcal{T}[\text{Decl}_2])}$$

A simple declaration binds the declared variable to nil.

$$\boxed{\mathcal{T}[\text{var:Type}]} \longrightarrow \boxed{\text{var} = \text{nil}}$$

Action Bodies: the “case” statement is transformed into a nesting of “if” statements.

$$\boxed{\begin{array}{l} \mathcal{T}[\text{case Exp}_1: S_1; \\ \quad \text{return Exp}'_1; \\ \dots \\ \text{case Exp}_n: S_n; \\ \quad \text{return Exp}'_n; \end{array}] \longrightarrow \boxed{\begin{array}{l} \mathcal{T}[\text{Exp}_1] \\ [\text{value}(\text{cond}) \rightarrow \\ \text{if cond} \\ \quad \text{then } \mathcal{T}[S_1] [\text{return Exp}'_1] \\ \quad \text{else} \\ \quad \mathcal{T}[\text{case Exp}_2: S_2; \text{return Exp}'_2; \\ \quad \dots \\ \quad \text{case Exp}_n: S_n; \text{return Exp}'_n;] \\] \end{array}}$$

We first transform “Exp₁”. We wait for the value of “Exp₁” (the method “value”). This value is bound to the cond. The “case” is transformed into an if statement such that if cond is “true”, we execute the transformation of S₁, otherwise we execute the transformation of the rest of the case.

Return statement: a “return” statement simply evaluates its argument expression and then sends it to the return address “ret” which is an argument of the initial method.

$$\boxed{\mathcal{T}[\text{return Exp}]} \longrightarrow \boxed{\mathcal{T}[\text{Exp}] [\text{value}(v) \rightarrow \text{ret.value}(v)]}$$

Statements: the statements of the source language can change the state of the configuration (object attributes or local variables). So we need an extra argument for our transformation function which is the continuation of the

transformed statement. So that this continuation is executed in an environment where the state update has been taken into account.

$$\boxed{\mathcal{T}[\text{var} := \text{Exp}][k]} \longrightarrow \boxed{\mathcal{T}[\text{Exp}] \text{ [value(v) } \rightarrow \text{ let var=v in (k)]}}$$

We first evaluate “Exp” and then execute “k” (the continuation of the assignment) in an context where “var” is bound to its new value.

$$\boxed{\mathcal{T}[\text{var.att} := \text{Exp}][k]} \longrightarrow \boxed{\mathcal{T}[\text{Exp}] \text{ [value(v) } \rightarrow \text{ var.set_att(v); become k]}}$$

For the assignment of a object attribute, we send to this object the message “set_att” with the new value as argument and then behave as specified by “k”.

$$\boxed{\mathcal{T}[\text{if Exp then } S_1 \text{ else } S_2][k]} \longrightarrow \boxed{\mathcal{T}[\text{Exp}] \text{ [value(v) } \rightarrow \text{ if v then } \mathcal{T}[S_1][k] \text{ else } \mathcal{T}[S_2][k]]}}$$

An if statement is straightforwardly transformed into an if statement in the target language.

$$\boxed{\mathcal{T}[S_1 S_2][k]} \longrightarrow \boxed{\mathcal{T}[S_1][\mathcal{T}[S_2][k]]}$$

The transformation of the second statement will be executed in a context possibly changed by the first statement.

Expressions: the expressions are transformed such that they send their values by sending the message “value” with their value as argument.

$$\boxed{\mathcal{T}[c]} \longrightarrow \boxed{\text{self.value(c);}}$$

Constants are evaluated to themselves.

$$\boxed{\mathcal{T}[\text{var}]} \longrightarrow \boxed{\text{self.value(var);}}$$

As for constants, the transformation of an access to a variable (var) consists in invoking the method “value” of “self” with “var” as argument.

$$\boxed{\mathcal{T}[\text{var.att}]} \longrightarrow \boxed{\text{var.get_att(self);}}$$

For attribute selection, we invoke the *get_att* method of actor *var* with *self* as return address.

$$\boxed{T[\text{op}(\text{Value}_1, \dots, \text{Value}_n)]} \longrightarrow \boxed{\text{self.value}(\text{op}(\text{Value}_1, \dots, \text{Value}_n))}$$

Ground expressions are evaluated to themselves.

$$\boxed{T[A(\text{Exp}_1, \dots, \text{Exp}_n)]} \longrightarrow \boxed{\begin{array}{l} T[\text{Exp}_1] \\ [\text{value}(v_1) \rightarrow \\ \dots \\ T[\text{Exp}_n] \\ [\text{value}(v_n) \rightarrow \\ \quad \text{Act_A.A}(v_1, \dots, v_n, \text{self}) \\ \dots \\] \\] \end{array}}$$

Expressions Exp_1 to Exp_n are evaluated sequentially, then method A of actor Act_A is invoked with self as return address.

$$\boxed{T[\text{new C}(\text{Exp}_1, \dots, \text{Exp}_n)]} \longrightarrow \boxed{\begin{array}{l} T[\text{Exp}_1] \\ [\text{value}(v_1) \rightarrow \\ \dots \\ T[\text{Exp}_n] \\ [\text{value}(v_n) \rightarrow \\ \quad \text{self.value}(\text{new Act_C}(v_1, \dots, v_n)) \\ \dots \\] \\] \end{array}}$$

Expressions Exp_1 to Exp_n are evaluated sequentially, then method value of the current actor (self) with newly created actor behaving according to Act_C as parameter.

4.4 Proving the correctness of the transformation

Transformation T is one of the two steps that should be performed so that the semantics of the source information description is preserved in the target computational programs. Indeed, the target actors produced by T behave correctly with regard to their source information specification only when the executions of action translated actors are serialisable*. Thus, another transformation is needed in association with T , and which generates the appropriate mechanism to control the concurrency of executions of actors (this is the role which is played by transaction object services in CORBA).

*Conditions looser than serialisable could be investigated, but this is out the scope of the present paper.

We discuss hereafter the semantics preservation properties of $\mathcal{T}$ only for the serialisable executions of actors; the concurrency control aspect is being researched and left for a second, forthcoming contribution. Let $\mathcal{C}$ be a collection of classes and actions of the source language and $\mathcal{A}$ the collection of actors resulting from the transformation of elements of $\mathcal{C}$. We denote by $\mathbb{S}_{\mathcal{C}}$ the set of instances of classes of $\mathcal{C}$. Thus, for each instance c of $\mathbb{S}_{\mathcal{C}}$, $\exists C \in \mathcal{C}$ such that $c : C$. Similarly, we denote by $\mathbb{T}_{\mathcal{A}}$ the set of instances of actors of $\mathcal{A}$. Transformation $\mathcal{T}$ can be extended in order to be defined on instances of classes, i.e. $\hat{\mathcal{T}}$ maps instances of classes on instances of actors. The extension is given by:

for $\langle att_1 = v_1, \dots, att_n = v_n \rangle : C$,

$$\hat{\mathcal{T}}(\langle att_1 = v_1, \dots, att_n = v_n \rangle) = become \mathcal{T}(C)(v_1 \dots v_i)$$

Let $\bar{\mathbb{S}}_{\mathcal{C}}$ denote the power-set of $\mathbb{S}_{\mathcal{C}}$. Elements of $\bar{\mathbb{S}}_{\mathcal{C}}$ denote configurations of class instances. Similarly, elements of $\bar{\mathbb{T}}_{\mathcal{A}}$ denote configurations of Actor instances. For each action $A \in \mathcal{C}$, the semantics of the source language defines a partial* function $\llbracket A \rrbracket_{\mathcal{C}}$ which maps class instances on class instances: $\llbracket A \rrbracket_{\mathcal{C}} : \mathbb{S}_{\mathcal{C}} \rightarrow \mathbb{S}_{\mathcal{C}}$. In a similar way, the semantics of the target language allows to associate to each actor A_c , a partial* function $\llbracket A_c \rrbracket_{\mathcal{A}} : \mathbb{T}_{\mathcal{A}} \rightarrow \mathbb{T}_{\mathcal{A}}$.

The semantics preservation of our transformation can now be stated: $\mathcal{T}$ is an information to computational semantics preserving transformation iff the diagram 1 is commutative.

Figure 1 Transformation diagram.

The full proof of this property is too long to be given in this paper. It is achieved in two steps. In a first step, we define a labelled transition based semantics of the source language which we prove equivalent to the one given in section 2.4. In this setting, an information state of the source language is made of a configuration of class instances and of the current state of execution of the action under consideration. In the second step, we construct a simulation relation between a source description (a configuration made of class instances and one action) and its image in the transformation, whereby the information states are mapped on to computational states and elementary execution steps of the source language are mapped onto sequences of executions (possibly involving message exchanges between actors) in the target language.

*The partiality of this function reflects non terminating actions.

5 CONCLUSIONS

This paper has presented a correctness preserving transformation from a class-oriented Information viewpoint language to an Actor-like language for the Computational viewpoint. The language for the Information viewpoint has been given a formal semantics, whereas the language for the Computational viewpoint has been described informally only. We then defined the transformation in the form of a set of rules that perform syntactic transformations on the Information viewpoint grammar.

The main distinguishing feature of the Information viewpoint language is that actions (operations) are separated from class definitions. This will provide a lot of flexibility for composing objects at the Computational viewpoint. More work is needed to define rules for how objects can be composed.

The general issue of consistency in open distributed systems is a particularly challenging problem. So far, little work has been carried out in this area as pointed out by Bowman et.al. in (Bowman et al. 1996) and (Derrick et al. 1996). (Bowman et al. 1996) discusses this issue in the context of ODP and gives a general definition of consistency which covers several of the consistency definitions listed for ex. in the ISO ODP/RM (Farooqui et al. 1995). They emphasise the importance of unification of descriptions from several viewpoints into a final *implementation specification*. In (Derrick et al. 1996) the same authors propose a formal translation from LOTOS to Z. The emphasis is on the definition of a common semantic model to which specifications in the two languages can be translated and then compared for observational equivalence.

Apart from the work of Bowman et.al. we should also mention the work of the LOTOS community. The development of correctness preserving transformations carried out for the LOTOS specification language is a striking example (Cunha et al. 1995). In the development cycle of LOTOS, constraint oriented specifications are used in a first stage of the development, to pass then to resource oriented specifications for the late stage of the development. Transformations which allow this passage preserving the correctness of the specifications have been successfully defined and used.

In the near future we will extend this work to deal with transaction management as an integral part of the transformation. The motivation for this is to provide guarantees for the Information viewpoint's atomicity assumption. Furthermore, we consider to strengthen the links between the viewpoints by using reflection techniques. This might be particularly interesting in order to take into account design decisions made at the Computational viewpoint. The simple language we have defined for the Information viewpoint can only express functional requirements. There is a lot of other information that affects the final design. It is probably not feasible to provide formal languages for all of this information, but still, having more than one language is clearly imaginable. In that case it would be interesting to define transformations from each of the Information viewpoint languages and to have some way of combining

these transformations into a coherent whole at the Computational viewpoint, with as much automatic support as possible.

REFERENCES

- G. A. Agha, (1986), *Actors: A Model of Concurrent Computational in Distributed Systems*, MIT Press, Cambridge, Mass.
- G. A. Agha, I. A. Mason, S. F. Smith and C. L. Talcott, (1993), *A Foundation for Actor Computation*, J. Functional Programming 1 (1).
- C. Bernardeschi, A. Fantechi, F. Paternò, (1995), *Application of Correctness Preserving Transformations for Deriving Architectural Descriptions of Interactive Systems from User Interface Specifications*, Proc. SEKE'95, Rockville, Maryland.
- H. Bowman, E. A. Boiten, J. Derrick, and M. W. A. Steen, (1996), *Viewpoint Consistency in ODP, a General Interpretation*. Proc. First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96, Paris, France. Chapman and Hall.
- T. Bolognesi (Ed.), (1992). Catalogue of LOTOS Correctness Preserving Transformation. Lo/WP1/T1.2/N0045/V03, Final Deliverable, LOTOSphere ESPRIT project 2304.
- P. Cunha, A. Fantechi, B. Mekhanet, E. Najm and J. Queiroz, (1995), *Correctness Preserving Transformations for the Late Phases of Development*. Chapter 9 of T. Bolognesi, J. van de Lagemaat and C. Vissers eds., LOTOSphere: Software Development with LOTOS, Kluwer Academic Publishers, 181–199.
- J. Derrick, E. A. Boiten, H. Bowman, and M. W. A. Steen, (1996). *Supporting ODP — Translating LOTOS to Z*. Proc. First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96, Paris, France. Chapman and Hall.
- K. Farooqui, L. Logrippo, J. de Meer, (1995), *The ISO reference model for Open Distributed Processing: an introduction*, Computer Networks and ISDN Systems 27, 1215–1229.
- H-M, Jarvinen and R. Kurki-Suonio, (1990), *The DisCo Language*, Report 8, Tampere University of Technology, Software Systems Laboratory, March.
- Robin Milner, Joachim Parrow, David Walker, (1992), *A Calculus of Mobile Processes (Parts I and II)*, Information and Computation, 100:1-77.
- E. Najm and J-B Stefani, (1995), *A formal semantics of the ODP computational model*, Computer Networks and ISDN Systems 27, 1305–1329.
- B. C. Pierce and D. N. Turner, (1995), *Concurrent objects in a process calculus*, Proc. Theory and Practice of Parallel Programming, Sendai, Japan, 1994, Lecture Notes in Computer Science 907, 187–215.