Electronic Notes in Theoretical Computer Science 72 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 15 pages

Process semantics for UML component specifications to assess inheritance

E.E.Roubtsova [1,2]

Faculty of Technology Management, TU Eindhoven, Den Dolech 2, Box 513, 5600MB Eindhoven, The Netherlands

R.Kuiper [3]

Faculty of Mathematics and Computing Science, TU Eindhoven, Den Dolech 2, Box 513, 5600MB Eindhoven,TU Eindhoven, The Netherlands

Abstract

We define a component specification as a process. The starting point is the specification of a component in a UML profile. The process of the component is a derivable feature from the component specification. We define the inheritance of component specifications as inheritance of processes. Process semantics of the UML profile allows to check inheritance of specifications using a process algebra with renaming functions, we have presented.

1 Introduction

Inheritance of component specifications is a difficult practical problem, because different definitions of component specification focus on different component features and the inheritance mechanism for specific features demands specific models.

The main feature of a component is the behavioural pattern corresponding to this component. If a component inherits a parent-component, the parent behavioural pattern should be inherited. We use the Unified Modeling Language (UML) profile for component design [4,5,6] and capture the behavioural

[1] The work is supported by PROGRESS grant EES.5141 and ITEA grant TSIN2009 "EMPRESS". We thank dr. H.B.M. Jonkers (Philips Research Laboratories Eindhoven, the Netherlands), prof. W. van der Aalst (TU Eindhoven, the Netherlands), prof. F. Corradini (University L'Aquila, Italy) for the inspiration of this work.
[2] Email: E.Roubtsova@tue.nl
[3] Email: R.Kuiper@tue.nl

©2002 Published by Elsevier Science B. V.

pattern in terms of roles and interfaces provided by roles. The pattern is represented by an interface-role diagram being a class diagram in the UML and by a set of sequence diagrams. However, a set of diagrams does not represent a behavioural pattern as an entity to inherit from.

We formalize the behavioural pattern as a process algebra term, a process for short. The process is derived from the UML specification of a component. So, the UML specification of a component is transformed to a process. The actions of the process are interfaces provided and required by roles from a closed group of roles. A closed group of roles with interfaces provided and required by these roles is called an interface-suite and represents one component [6].

We consider the approaches where composition is done by inheritance. If a system is composed from components by inheritance, the system specification inherits component specifications. Inheritance is defined in the UML at the level of class diagrams. The inheritance of behavioural views is not defined in the UML. We therefore define the inheritance of behavioural views as the inheritance of processes. We show by example how helpful this approach is to check inheritance of a component specification.

The paper is organized as follows. Section 2 defines an interface-suite as a process. We show how such a process is specified in a UML profile for component specification. In section 3, we formalize the inheritance of component specifications as inheritance of processes and we show by example how to check the inheritance. Section 4 contains conclusions.

2 An interface-suite as a behavioural pattern

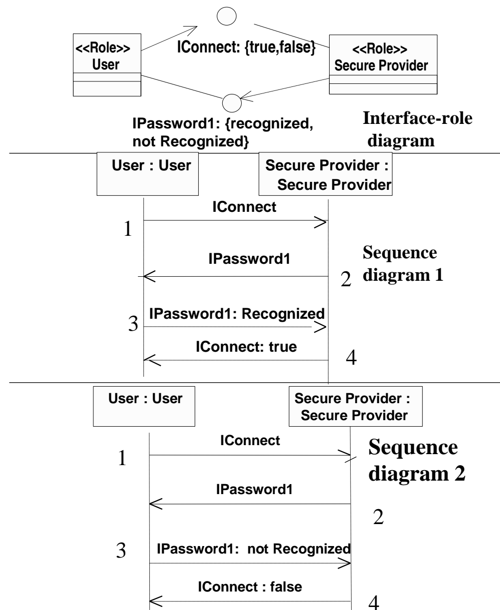

Fig. 1. Component *Internet Provider*

2.1 An interface-suite example

Consider component *Internet Provider*. It is specified by an interface-suite *Internet Provider* which contains two roles: role *User* and role *Secure Provider* (Fig. 1). The *User* asks the *Secure Provider* to give an internet connection via interface *IConnect*. The *Secure provider* checks the password of the *User* via interface *IPassword1*. If the password has been recognized, the *Secure Provider* connects the *User*. If the password has not been recognized, the *Secure provider* does not connect the *User*.

2.2 A UML Profile for Interface-suites

We specify an interface-suite IS in a UML profile which contains an *interface-role diagram* IR and a *set of sequence diagrams* $s_1..s_n$ (Fig.1): $IS = (IR, s_1 \dots s_n)$.

2.2.1 An interface-role diagram IR

An interface-role diagram is a UML class diagram where roles are represented by classes with stereotype $\ll$ *Role* $\gg$. An interface-role diagram is a graph $IR = (R, I, PI, RI, RR)$ with two kinds of nodes and three kinds of relations:

- R is a finite set of roles depicted by boxes. Each role $r \in R$ has a set of players Pl_r. If the number of players $|Pl_r|$ is more than one, the number is drawn near the role.
- I is a finite set of interfaces depicted by circles. Each interface $i \in I$ has a set of results of interface Res_i. Results are shown as sets of values near the interface.
- $PI = \{(r, i)|\ r \in R, i \in I\}$ defines interfaces provided by roles. Each role provides a finite set of interfaces, $|PI \cap R \times I'| \geq 0, I' \subseteq I$. The relation is depicted by a solid line between a role and an interface (Fig. 1).
- $RI = \{(r', (r, i))|\ r', r \in R, i \in I, (r, i) \in PI\}$ defines interfaces required by roles. Each role requires a finite set of provided interfaces $|RI(r, PI')| \geq 0, PI' \subseteq PI$. The required relation is drawn by a dashed arrow connecting a role and a provided interface. The arrow is directed to the interface (Fig. 1).
- $RR = \{(r, r')|r, r' \in R\}$ is the relation of inheritance on the set of roles. The relation is shown by a solid line with the triangle end $r' \!-\!\!\triangleright r$ directed from role-child r' to role-parent r.

2.2.2 Defining the set of actions from an interface-role diagram

Let set $RR = \emptyset$ for a moment.

Notice, that usually **not** all roles can use all interfaces. An action $a = r^1.r^2.i$ is specified by an interface role diagram $IR = (R, I, PI, RI, RR)$, if $i \in I, \quad r^1, r^2 \in R, \quad (r^2, i) \in PI$ and $(r^1, (r^2, i)) \in RI$. So, *the set of required interfaces defines the set of actions at the interface-role diagram.* If we take into account that the use of each interface can return different results res from the set Res and that a role has a finite set of instances named players Pl, $pl \in Pl$, then the set of actions is defined completely.

An action of an interface-suite can be represented by the following complex name $a = r^1.pl^1.r^2.pl^2.i : res$, where res is empty if a represents an interface

call [8], $res \in Res_i$ if a represents a return. The set of required interfaces RI, $ri_k \in RI$, defines the set of actions in the process corresponding to the interface-suite.

In our case study, we have only one player of each role. So there are two create actions: $createPlayers = \{createUser, createSecureProvider\}$. The name of an action has the following structure: $a = r^1.r^2.i : res$. We abbreviate the action names to letters a, b, c, d, e, f:
$a = User.SecureProvider.IConnect$; $b = User.SecureProvider.IConnect : true$;
$c = User.SequreProvider.IConnect : false$; $d = SecureProvider.User.IPassword1$;
$e = SecureProvider.User.IPassword1 : Recognized$;
$f = SequreProvider.User.IPassword1 : not\ Recognized.$

2.3 Sequence diagram

A sequence diagram for an interface-suite is a tuple $s = (R \times Pl, T_s, N_s)$,

- $R \times Pl$ is a set of players of roles. A player of a role is represented by a box with a line drawn down from the box [7];
- $T_s = \{(v, w, l) \mid v, w \in R \times Pl,\ l \in L = I \times Res\}$ is a labelled relation.
  Notice, that $T_s \subseteq R \times Pl \times R \times Pl \times I \times Res$ corresponds to the interface-role diagram. The relation T_s is represented by a labelled arrow between lines drawn down from boxes v and w (Fig. 1).
- However, an action $a = v.pl_v.w.pl_w.i : res$ defined by the interface-role diagram can have several occurrences and can be represented by *several* arrows at an sequence diagram. To distinguish arrows labelled by the same name and to define the order of actions there is an ordering line drawn down from each box. All these lines together represent one ordering line (time dimension) [7], which gives numbers in the sequence to all actions; $N_s = \{((v, w, l), n) \mid (v, w.l) \in T_s,\ n = 1, 2, ..., n\}$.

As follows from the definition, a sequence diagram corresponds to a sequence

$$s = (a_1, ..., a_j, ..., a_n),\ \text{where } j \in N,\ a_j \in R \times Pl \times R \times Pl \times I \times Res.$$

2.4 Process semantics of the UML profile for interface-suites

We will construct from component specifications processes of type

$$IS = (p, A, T, p*, p_F)\ [2].$$

- p is the initial state of the process. In this paper, the states are abstract. States are named by letters with numbers: $p, p_1, p_2, ..., p^F$.
- A is a finite set of actions.
- T is a set of transitions. A transition $t \in T$ defines a pair of states (p', p''), such that p'' is reachable from p' as a result of the action a, denoted $p' \stackrel{a}{\Longrightarrow} p''$. If we define an abstract set of all possible states $\mathcal{P}$ of the interface suite, then $T \subseteq \mathcal{P} \times A \times \mathcal{P}$.
- $p*$ is the finite set of states reachable from the initial state p. $p* \subseteq \mathcal{P}$. The reachability relation on the set of states $\stackrel{*}{\Longrightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ is the smallest relation reflexive and transitive for any $p, p', p'' \in \mathcal{P}$, $a \in A$, $p \stackrel{*}{\Longrightarrow} p$, $\quad (p \stackrel{*}{\Longrightarrow} p' \wedge p' \stackrel{a}{\Longrightarrow} p'') \rightarrow p \stackrel{*}{\Longrightarrow} p''$.
- p_F is the final state of a process, $p_F \in p*$. If $p'' \neq p_F$ then exists a nonempty subset of states $p''* \subseteq \mathcal{P}$ reachable from p''.

2.4.1 Constructing the process corresponding to a set of sequence diagrams Each sequence diagram is a path of the constructed process IS from the initial state to the final state. The set of sequence diagrams for an interface-suite represents a process IS.

In the initial state of any process, a role *Factory* is created by action *start*, then all players of roles are created by role *Factory*. Usually, these initial actions *start* and *createPlayers* are not shown at sequence diagrams.

In appendix A we present the algorithm for constructing the process-term IS corresponding to a set of sequence diagrams. The algorithm is based on comparing the elements of complex action-names (roles, players etc.). If sets of players from two sequence diagrams are disjoint, then sequences belong to parallel processes. If sets of players from two sequences are overlap, then actions with equal names from begin of sequences form a sequential process, the first unequal actions raise a branching process.
The resulting process term IS is of type

$$IS = start \cdot createPlayers \cdot (T) \cdot final, \text{where} \quad T = Z^1 \| ... \| Z^K, k = 1...K;$$
$$Z = \epsilon \quad \text{or} \quad Z = X \cdot (Y^1 + ... + Y^L), l = 1...L;$$
$$X = \epsilon \quad or \quad X = x_1 \cdot ...x_h \cdot ...x_n, \ h = 1...n;$$
$$Y^l = \epsilon \quad or \quad Y^l = y_1^l \cdot rest_{Y^l}, \ l = 1...L.$$

If we apply the algorithm from appendix A for our case study (Fig.1), then two sequence diagrams from (Fig.1)

$$\begin{aligned} SequenceDiagram1 = & (User.SecureProvider.IConnect)_1, \\ & (SecureProvider.User.IPassword1)_2, \\ & (SecureProvider.User.IPassword1 : Recognized)_3, \\ & (User.SecureProvider.IConnect : true)_4. \\ SequenceDiagram2 = & (User.SecureProvider.IConnect)_1, \\ & (SecureProvider.User.IPassword1)_2, \\ & (SecureProvider.User.IPassword1 : not\ Recognized)_3, \\ & (User.SecureProvider.IConnect : false)_4. \end{aligned}$$

define the following process $IS = start \cdot createPlayers \cdot \ IP \ \cdot final$, where

$$\boldsymbol{IP} = [User.SecureProvider.IConnnect] \cdot \ [SecureProvider.User.IPassword1] \cdot \Big([SecureProvider.User.IPassword1 : Recognized] \cdot \ [User.SecureProveider.IConnect : true] + [SecureProvider.User.IPassword1 : not\ Recognized] \cdot \ [User.SecureProvider.IConnect : false\ ] \Big) = \boldsymbol{a \cdot d \cdot (e \cdot b + f \cdot c)}.$$

3 Inheritance of interface-suites

Definition 3.1 *Let two interface-suites be given*

$$IS_1 = (IR_1, s_1^1 ... s_n^1), \ IS_2 = (IR_2, s_1^2 ... s_m^2).$$

IS_2 *inherits* IS_1*:* $IS_2 \multimap IS_1$, *if and only if*

- *interface-role diagram* IR_2 *inherits interface-role diagram* IR_1*:* $IR_2 \multimap IR_1$*;*
- *set of sequences* $(s_1^2 ... s_m^2)$ *inherits set of sequences* $(s_1^1 ... s_n^1)$*:* $(s_1^2 ... s_m^2) \multimap (s_1^1 ... s_n^1)$.

3.1 Inheritance at the interface-role diagram level

We define the inheritance of interface-suites at the interface-diagram level via inheritance of roles as used in [5,6]. Roles are specific UML classes. If role r_2 inherits r_1, $r_2 \multimap r_1$, it is drawn at the interface-role diagram by a solid arrow with the triangle-end from r_2 to r_1 (Fig. 2).

Definition 3.2 *Let interface-role diagrams C and S be given:*

$$C = (R_C, I_C, PI_C, RI_C, RR_C),\ S = (R_S, I_S, PI_S, RI_S, RR_S).$$

Interface-role diagram S inherits interface-role diagram C: $S \multimap C$, if and only if there is an interface-role diagram $G = (R_G, I_G, PI_G, RI_G, RR_G)$, such that

(i) • $R_C \cap R_G = \emptyset, I_C \cap I_G = \emptyset$,
 • $R_S = R_C \cup R_G,\ I_S = I_C \cup I_G$,

(ii) $RR_S = RR_C \cup RR_G \cup RR_*$, *where*
$RR_* = \{(r_C, r_G) | r_C \in R_C, r_G \in R_G, \& r_G \multimap r_C\},\ RR_* \neq \emptyset$.
So, the relation defines $R_* \subseteq R_G$, $r_* \in R_*$ if exists $r_C \in R_C$ such that $r_* \multimap r_C$, i.e. $(r_C, r_*) \in RR_*$.

(iii) $PI_S = PI_C \cup PI_G \cup PI_*$,
$PI_* = \{(r_*, i) |\ \ r_* \in R_*,\ \ i \in I_C, \exists r \in R_C$,
such that $r_* \multimap r$, *and* $(r, i) \in PI_C)\}$.

(iv) $RI_S = RI_C \cup RI_G \cup RI_*$, *where*
$RI_* = \{(x_*, (r_*, i))\ \ |\ \ r_*, x_* \in R_*,\ i \in I_C$,
there exist roles $r, x \in R_C$, *such that* $r_* \multimap r,\ x_* \multimap x$
and $(r, i) \in PI_C$ *and* $(x, (r, i)) \in RI_C\}$.

Fig.2 shows interface-role diagrams C from Fig.1, interface-role diagram G drawn in black lines and interface-role diagram S from definition 3.2. Interface-role diagram G has role *Employee* which provides interface *IPassword2*. This interface is required by role *Administrator*.

(i) Role sets of C and G are disjoint, interface sets of C and G are disjoint. There are roles of G which inherit roles of C. Role *Employee* inherits role *User*, *Administrator* inherits *Secure Provider*. Interface-role diagram S inherits interface-role diagram C.

(ii) The consequence of the role inheritance is the corresponding inheritance of interfaces. If $r_G \multimap r_C$ then r_G inherits all provided interfaces of r_C and may provide more. Role *Employee* inherits role *User*, so *Employee* provides interface *IPassword1* provided by *User*. Moreover, *Employee* provides interface *IPassword2*.

(iii) The inheritance of required interfaces is different. Role r_G can require new interfaces which are different from interfaces required by r_C. So, there is the option to define new required interfaces RI_G for role r_G, provided by roles of G. However, if role r_G requires the same interfaces as role r_C requires from role r'_C, then some role r'_G should exist in the interface-role diagram G to provide those interfaces. This role r'_G is supplied through inheritance from the role r'_C. For example, role *Administrator*$\in G$ requires both interfaces provided by role *Employee*$\in G$: *Employee.IPassword1*$\in RI_*$ and *Employee.IPassword2*$\in RI_G$, but interface *Employee.IPassword1*$\in RI_*$ is not drawn in interface-role diagram G, it is just a duplication of interface *User.IPassword1*$\in RI_C$.

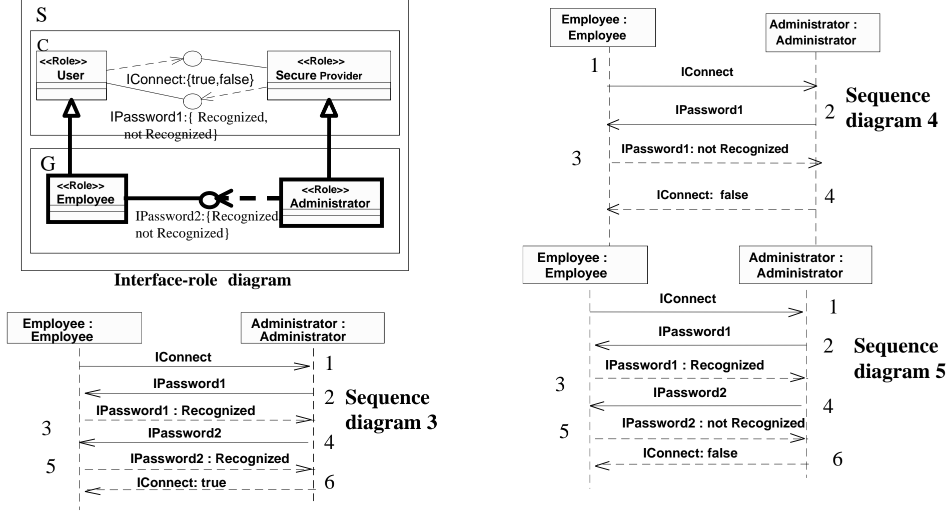

Fig. 2. Component *Local Access*

The main feature of our definition is the following. The roles of the interface-role diagram G can not require interfaces of parent roles from the interface-role diagram C and roles from C can not require interfaces of roles from G. This feature is the basis for compatibility of interface-suites specified by C and by S. C specifies the old version of the product. S specifies the new one. The old version of the specified product should always be available in the new version so that old interfaces can be used by old roles. Therefore the specification of the old version is saved in the specification of the new product.

The interface-role diagram specifies three sets of actions:

- $Inh = createPlayers_C \cup RI_C \times Pl_C \times Res_{RI_C}$,
- $New = createPlayers_{New} \cup RI_G \times Pl_{New} \times Res_{RI_G}$,
- $W = createPlayers_* \cup RI_* \times Pl_* \times Res_{RI_*}$.

The inheritance of interface-suites defines the duplicating $1-1$ function ρ^W_{Inh} which duplicates actions from set Inh of the parent interface-suite to actions from subset W of the interface-suite-inheritor.

3.1.1 Example of inheritance at the interface-role diagram level

We construct a component which is an inheritor from the *Internet provider.* It is a *Local Access* component that provides access to secure information in a company. Role *Administrator* of this new component inherits the *Secure provider* and role *Employee* inherits the *User* (Fig. 2). The behavioural pattern of the *Local Access* component is the following: *An Employee asks about an access to the secure information. The Administrator checks the password of*

the Employee two times. Only if the password is recognized two times, the Administrator connects the Employee. In all other cases, the Administrator does not connect the Employee.

The *Local Access* component inherits actions from *Internet Provider* as we can see at the interface-role diagram.

- The set of inherited actions which belong to the interface-suite C has been listed in subsection 2.2.2: $Inh = \{createPlayers, a, b, c, d, e, f\}$.
- The set of new actions is specified the interface-suite G :

  $New = \{createPlayers_{New}, x, y, z\}$
  $createPlayers_{New}$ are actions creating players of new roles which do not inherit parent roles; $createPlayers_{New} = \emptyset$ for this case study;
  $x = Administrator.Employee.IPassword2$;
  $y = Administrator.Employee.IPassword2:$ *Recognized*;
  $z = Administrator.Employee.IPassword2:$ *not Recognized.*
- The set of duplicated actions represents the actions which have been copied from C and duplicated in correspondence with definition 3.2. :
  $W = \{createPLayers', a', b', c', d', e', f'\}$.

$createPlayers$ **is renamed to** $createPlayers'$ to create players of roles which inherit parents roles: *createEmployee* and *createAdministrator*.
$a = User.SecureProvider.IConnect$ **is renamed to** $a' = Employee.Administrator.IConnect$;
$b = User.SecureProvider.IConnect: true$ **to** $b' = Employee.Administrator.IConnect: true$;
$c = User.SequreProvider.IConnect: false$ **to** $c' = Employee.Administrator.IConnect: false$;
$d = SecureProvider.User.IPassword1$ **to** $d' = Administrator.Employee.IPassword1$;
$e = SecureProvider.User.IPassword1:$ *Recognized* **is renamed to**
$e' = Administrator.Employee.IPassword1:$ *Recognized*;
$f = SecureProvider.User.IPassword1:$ *not Recognized* **to**
$f' = Administrator.Employee.IPassword1:$ *not Recognized.*

3.2 Inheritance of the set of sequence diagrams as inheritance of processes

Inheritance of the set of sequence diagrams is not defined in UML. But, not every set of sequence diagrams, constructed from those inherited and renamed actions, can be viewed as properly inherited. It is not clear, for example, from the first look, if the set of sequence diagrams defined for the *Local Access* component (Fig. 2) inherits the set of sequences specified for the *Internet Provider* component in Fig.1. (We deliberately introduce a flaw to exemplify our approach.)

However, following [2] we can easily define a notion of interface-suite inheritance as inheritance of processes corresponding to sets of sequence diagrams.

Definition 3.3 *For any processes p, q being closed terms in a process algebra PA_{IS}, process q is an inheritor of process p under* interface-suite inheritance relation $q -\triangleright p$ *if and only if*

- *there are disjoint sets of actions Inh, New, W, there is a set $H \subseteq W$.*
- *process $\rho^{W}_{Inh}(p)$ is derived from process q in the process algebra PA_{IS} using*

functions $\tau_{Inh}, \tau_{New}, \delta_H$

$$PA_{IS} \vdash \tau_{New \setminus H}(\delta_H(\tau_{Inh}(q))) = \rho^W_{Inh}(p).$$

Process algebra for interface-suites PA_{IS}

$P, -$*set of processes*$, A -$ *set of actions*$, A \subseteq P, \quad \delta : P, \delta -$ *deadlock action*

$+ -$ *alternative composition*$, \cdot -$ *sequential composition*$, \| -$ *parallel composition*,

τ- silent action; $\lfloor\!\lfloor$ left process must perform the first action, $P \times P \rightarrow P$,

$H, I \subseteq A$, H, I are disjoint; $\delta_H, \tau_I; P \rightarrow P$,

A_1	$x + y = y + x$	M_1	$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$
A_2	$(x + y) + z = x + (y + z)$	M_2	$a \lfloor\!\lfloor x = a \cdot x$
A_3	$x + x = x$	M_3	$a \cdot x \lfloor\!\lfloor y = a \cdot (x \| y)$
A_4	$(x + y) \cdot z = x \cdot z + y \cdot z$	M_4	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$
A_5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		
A_6	$x + \delta = x$	B_1	$x \cdot \tau = x$
A_7	$\delta \cdot x = \delta$	B_2	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$
D_1	$a \notin H \Rightarrow \delta_H(a) = a$	T_1	$a \notin I \Rightarrow \tau_I(a) = a$
D_2	$a \in H \Rightarrow \delta_H(a) = \delta$	T_2	$a \in I \Rightarrow \tau_I(a) = \tau$
D_3	$\delta_H(x + y) = \delta_H(x) + \delta_H(y)$	T_3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
D_4	$\delta_H(x \cdot y) = \delta_H(x) \cdot \delta_H(y)$	T_4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$

Axioms $A_1 - A_7$ formalize alternative and sequential composition of processes, $M_1 - M_4$ - behaviour of concurrent processes, constant $a \in A \cup \{\delta\}$. Axioms B_1, B_2 allow to remove a silent action τ which does not enforce a choice. Axioms $D_1 - D_4$, $T_1 - T_4$, $R_1 - R_6$ introduce renaming operators. The blocking operator δ_H renames occurrence of actions from $H \subseteq A$ in a process term to δ constant. The hiding operator τ_I renames action in $I \subseteq A$ in a process term to silent action τ [2].

The inheritance relation is a preorder, i.e. a reflexive and transitive relation that induces an equivalence relation: *Two processes are equivalent under the inheritance relation if and only if their equality is derivable from the axioms of PA_{IS}.* So, the transformation of the sets of sequence diagrams to the corresponding processes allows to apply axioms of process algebra and check equivalence of processes under the interface-suite inheritance relation. The transformation allows also to say that *a UML component specification S inherits a UML component specification C if the process constructing from S inherits from the process constructing from C.*

3.2.1 Example of inheritance of the set of sequence diagrams. Have we specified a correct inherited process?

The process LA corresponding to the set of sequence diagrams for the *Local Access* component (Fig. 2) is the following

$LA = start \cdot createPlayers \cdot createPlayers' \cdot (IP \| X) \cdot final,$ where

- Process IP is represented by sequence diagrams 1 and 2 and the corre-

sponding process term has been derived in section 2.4.1.

- $X = a' \cdot d' \cdot (f' \cdot c' + e' \cdot x \cdot (y \cdot b' + z \cdot c'))$. Sequence diagrams 3-5 represent the process X. Process X can be constructed applying the algorithm from appendix A.

Let us try to check inheritance of interface-suites *Internet Provider* and *Local Access*.

(i) We should derive process $IS' = \rho^{W}_{Inh}(IS)$ from process LA.
$IS' = \rho^{W}_{Inh}(IS) = start \cdot createPlayers' \cdot a' \cdot d' \cdot (e' \cdot b' + f' \cdot c') \cdot final.$

(ii) First we abstract from the inherited actions from set *Inh*:
$X_1 := \tau_{Inh}(LA) = \tau_{Inh}(start \cdot createPlayers \cdot createPlayers' \cdot (IP \| X) \cdot final)) =$
(Axioms T_1, T_2) $start \cdot \tau \cdot createPlayers' \cdot (\tau \| X) \cdot final =$
(Axioms B_1, M_1) $start \cdot createPlayers' \cdot (\tau \lfloor\!\lfloor X + X \lfloor\!\lfloor \tau) \cdot final =$
(Axiom M_2) $start \cdot createPlayers' \cdot (\tau \cdot X + X) \cdot final =$
(Axiom B_2) $start \cdot createPlayers' \cdot X \cdot final.$

(iii) Then, we abstract from actions of set
$New = \{x, y, z\}$ defined by interface suite G:
$X_3 := \tau_{New}(X_2) = \tau_{New}(start \cdot createPlayers' \cdot (a' \cdot d' \cdot (e' \cdot x \cdot (y \cdot b' + z \cdot c') + f' \cdot c')) \cdot final =$
(Axioms T_1, T_2) $start \cdot createPlayers' \cdot (a' \cdot d' \cdot (e' \cdot \tau \cdot (\tau \cdot b' + \tau \cdot c') + f' \cdot c')) \cdot final =$
(Axioms B_1) $start \cdot createPlayers' \cdot (a' \cdot d' \cdot (e' \cdot (\tau \cdot b' + \tau \cdot c') + f' \cdot c')) \cdot final.$

(iv) There is no such an $H \subseteq W$ which we can block to derive the desirable process IS'.

So, the derived process X_3 is not equivalent to the parent process IS' under the interface-suite inheritance relation. This means, we have a wrong design decision. Indeed, the result of interface call $IConnect$ in component *Internet provider* depends on the result of one interface call $IPassword1$, but the result of interface call $IConnect$ in component *Local Access* depends on results of two interface calls $IPassword1$ and $IPassword2$. So, we can not inherit interface $IConnect$, but should redefine it. Constructing of component *Local Access* via inheritance from component *Internet provider* is not reasonable.

On a different case, component *Access Administration*, we show an example of correct inheritance from component *Internet provider*. Besides the internet connection, component *Access Administration* has role *Statistics* which registers begins and ends of successful connections and denied connections (Fig. 3). The set of new actions is specified by interface-suite G:
$New = createPlayers_{New} \cup RI_g \times Pl_g \times Res_g = \{createPlayers_{New}, p, pp, q, qq, r, rr\},$

where an action from $createPalyers_{New}$ creates a player of role *Statistics*;
$p = Administrator.Statistics.IDeny$; $pp = Administrator.Statistics.IDeny: void$;
$q = Administrator.Statistics.IBegin$; $qq = Administrator.Statistics.IBegin: void$;
$r = Employee.Statistics.IEnd$; $rr = Employee.Statistics.IEnd: void$;

The complete process of component *Access Administration* is
$AA = start \cdot createPlayers \cdot createPlayers' \cdot createPlayers_{New} \cdot (IP \| X) \cdot final.$

Sequence diagrams $1, 2$ from Fig. 1 represent the process IP. Sequence diagrams $3 - 6$ from Fig. 3 are used to construct process X:

$$X = a' \cdot d' \cdot (e' \cdot q \cdot (b \cdot qq \cdot r \cdot rr + qq \cdot b \cdot r \cdot rr) + f' \cdot p \cdot (pp \cdot c' + c' \cdot pp)).$$

Let us try to check inheritance of interface-suites *Access Administration* and *Interface Provider*.

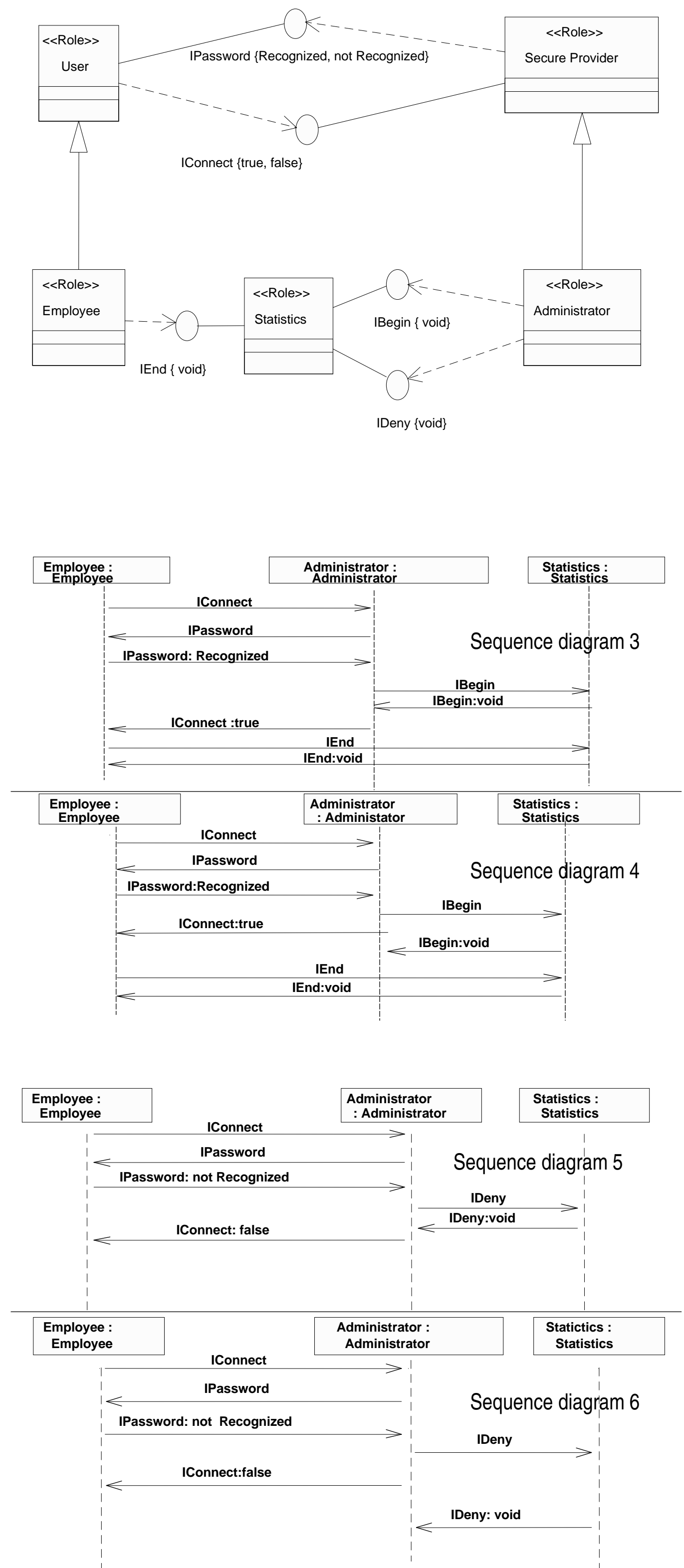

Fig. 3. Component *Access Administration*

(i) We should derive the process $IS' = \rho^W_{Inh}(IS)$ from process AA.
$IS' = \rho^W_{Inh}(IS) = start \cdot createPlayers' \cdot a' \cdot d' \cdot (e' \cdot b' + f' \cdot c') \cdot final.$

(ii) We do not repeat two standard steps to abstract from set of set Inh. The result of those steps is process:
$X_0 = start \cdot createPlayers' \cdot createPlayers_{New} \cdot X \cdot final.$

(iii) Abstracting from actions of set New we have the following process:
$X_1 := \tau_{New}(X_0) =$
$\tau_{New}(start \cdot createPlayers' \cdot createPlayers_{New} \cdot (a' \cdot d' \cdot (e' \cdot q \cdot (b' \cdot qq \cdot r \cdot rr + qq \cdot b' \cdot r \cdot rr) + f' \cdot p \cdot (pp \cdot c' + c' \cdot pp)) \cdot final) =_{(Axioms\ T_1,\, T_2)} start \cdot createPlayers' \cdot \tau \cdot (a' \cdot d' \cdot (e' \cdot \tau \cdot (b' \cdot \tau \cdot \tau \cdot \tau + \tau \cdot b' \cdot \tau \cdot \tau) + f' \cdot \tau \cdot (\tau \cdot c' + c' \cdot \tau)) \cdot final =$
$_{(Axiom\ B_1)}\ start \cdot createPlayers' \cdot (a' \cdot d' \cdot (e' \cdot (b' + \tau \cdot b') + f' \cdot (\tau \cdot c' + c')) \cdot final =$
$_{(Axiom\ B_2)}\ start \cdot createPlayers' \cdot (a' \cdot d' \cdot (e' \cdot b' + f' \cdot c')) \cdot final.$

Derived process X_1 is equivalent to process IS' under the interface-suite inheritance relation. The inheritance is correct.

4 Conclusion

The UML is the standard for system design, however, the methodology for component system design in the UML is still under development. The problem is that the notion of component includes different features and the semantics of the UML specification has to have a semantic match with all the notations used for checking of component features. Moreover, the methodology should guarantee saving component features when the system is composed from components.

In this paper we have defined a process semantics of a UML profile for component specification, which uses interface-role diagrams and sequence diagrams. We have developed an algorithm for transforming a specification in this profile to the corresponding process. The process represents the main component feature, namely, the behavioural pattern. On the basis of this pattern we define, compare and compose components via inheritance. The inheritance relation on component specifications in our UML profile is an inheritance of processes derived from the diagram sets representing components. If a pair of component specifications is an element of the inheritance relation, then the specifications represent the old and the new versions of a program product. The inheritance guarantees that the old specification is saved in the new one and the old product version will work in the new product.

In general, process semantics is a useful necessary step in the UML component system design at early stages. First, this semantics composes UML diagrams to a consistent specification. Second, process algebra notations are input notations for some tools such as [3], which can check inheritance of behavioural patterns as an equivalence. Third, process semantics can be extended to the automata semantics used in many model checkers at later stages of component system design. The pair of related process algebraic and automata models allows to verify most of vital properties of component systems.

References

[1] Baeten J.C.M.,W.P. Weijland , "Process Algebra," Cambridge University Press, 1990.

[2] Basten T., W. M. P.van der Aalst, *Inheritance of behaviour*, The Journal of Logic and Algebraic Programming **46** (2001), pp. 47–145.

[3] CafeOBJ, "`http://www.ldl.jaist.ac.jp/cafeobj/`," (2001).

[4] Cheesman J., J. Daniels, "UML Components. A simple Process for Specifying Component-Based Software," Addison-Wesley, 2001.

[5] D'Souza D.F., A.C.Wills, "Objects, Components and Frameworks with UML. The CATALYSIS Approach," Addison-Wesley , 1999.

[6] Jonkers H.B.M., *Interface-Centric Architecture Descriptions*, In proceedings of WICSA, The Working IEEE/IFIP Conference on Software Architecture (2001), pp. 113–124.

[7] OMG, "Unified Modeling Language Specification v.1.3, ad/99-06-10 `http://www.rational.com/uml/resources/documentation/index.jsp`," (1999).

[8] Roubtsova E.E, L.C.M.van Gool, R.Kuiper, H.B.M.Jonkers, *A Specification Model For Interface Suites*, UML'01, LNCS 2185 (2001), pp. 457–471.

A Algorithm for constructing the process IS.

The algorithm is described in the well known process algebra notation [1]. P, IS, Q, Q_1, Q_2, $X, B, Z^1, ...Z^K, Y^1, ...Y^L$ $rest_Y$ are processes of type P defined in 2.4. The sequential composition of processes is represented by point $Q_1 \cdot Q_2$, the alternative composition - by plus $Q_1 + Q_2$ and the parallel composition - by the parallel symbol $Q_1 \| Q_2$. The axioms about the ϵ process are as usual [1].

Given: a set of sequences diagrams $J_1, ..., J_r, ..., J_R$.
$J = (a_1, .., a_j, .., a_m) \mid j = 1...m,\ a_j = (r^1.pl^1.r^2.pl^2.i.res)_j = r^1_j.pl^1_j.r^2_j.pl^2_j.i_j.res_j$.

The resulting process IS is of type

$\boldsymbol{IS = start \cdot createPlayers \cdot (T) \cdot final,}$
where $\boldsymbol{T = Z^1 \| ... \| Z^K, k = 1...K;}$
$\boldsymbol{Z = \epsilon}$ **or** $\boldsymbol{Z = X \cdot (Y^1 + ... + Y^L), l = 1...L;}$
$\boldsymbol{X = \epsilon}$ ***or*** $\boldsymbol{X = x_1 \cdot ...x_h \cdot ...x_n,\ h = 1...n;}$
$\boldsymbol{Y^l = \epsilon}$ ***or*** $\boldsymbol{Y^l = y^l_1 \cdot rest_{Y^l},\ l = 1...L.}$

BEGIN

(i) Construct process IS from *start* and *createPlayers* actions for all players from all sequences diagrams $J_1, ..., J_r, ..., J_R$:
 $IS := start \cdot createPlayers;$

(ii) Construct process of type T from all processes of type Z.
 (a) Let initial process be empty $T = \epsilon$, let the number of parallel components be $K = 0$.
 (b) For all diagrams $J_1, ..., J_r, ..., J_R$
 begin

If the sets of players from T from J_r are disjoint then
begin
$Z_{K+1} := Process(\epsilon, J_r)$ *(the function is defined on the next page)*
– for all $k := 1, .., K \quad T := Z_1 \| ... \| Z_K \| Z_{K+1}$;
$K := K + 1$;
end;
If there is a k
such that sets of players from J_r and from Z_k are not disjoint
then for all such a k begin
$Z_k := Process(Z_k, J_r)$;
$T := Z_1 \| \dots Z_k \| \dots \| Z_K$;
end;
end;

(iii) Finish constructing process IS
$IS := IS \cdot (T) \cdot final$;

END.

FUNCTION $PROCESS(Z, J)$: *Result*
The function constructs a process *Result* of type $Z = X \cdot (Y^1 + ... + Y^L), l = 1...L$;
$X = \epsilon \quad or \quad X = x_1 \cdot ...x_h \cdot ...x_n,\ h = 1...n$;
$Y^l = \epsilon \quad or \quad Y^l = y_1^l \cdot rest_{Y^l},\ l = 1...L.$
from a process Z of type Z and
a sequence diagram $J = (a_1, .., a_j, .., a_m) \mid j = 1...m, .$
The function is repeated recursively.

BEGIN $\{PROCESS(Z, J)\}$
Result $:= \epsilon$;

(i) If $Z = \epsilon$ then if $m > 0$ then *Result* $:= a_1 \cdot ... \cdot a_m$;

(ii) If $Z \neq \epsilon$ then begin
- $h := 0,\ j := 0$;
- if $h = n$ { means $X = \epsilon$} and $j < m$ then begin
  $Y^{L+1} := a_j \cdot ... \cdot a_m$;
  $h := h + 1,\ j := j + 1$;
  Compare a_j and the first element of process Y^l for all $l = 1..L$.
  · if $y_1^l \neq a_j$ for all $l = 1..L$ then
  Result $:= Y^1 + ... + Y^L + Y^{L+1}$;
  · if there exists an l such that $y_1^l = a_j$ then
  begin
  for all such l $Y^l := PROCESS(Y^l, Y^{L+1})$;
  Result $:= Y^1 + ... + Y^l + Y^L$;
  end;
  end $\{X = \epsilon\}$
- if $h < n$ {means $X = x_1 \cdot ...x_h, ...x_n$} then begin
  $B := \epsilon$;
  A: h:=h+1; j:=j+1;

(a) if $h \leq n$ and $j \leq m$ *(process X and sequence J are continued)* then begin
if $x_h = a_j$ then begin $B := B \cdot x_h$;
goto (A), end ;
If $x_h \neq a_j$ then begin $Y^{L+1} := a_j \cdot ... \cdot a_m$;
if there exists an l such that $a_j = y_1^l$ then
begin

for all such l $Y^l := PROCESS(Y^l, Y^{L+1})$;
$Result := B \cdot (Y^1 + ... + Y^l + .. + Y^L)$;
end;
if for all $l = 1..L$ $a_j \neq y_1^l$ then
$Result := B \cdot (Y^1 + ... + Y^L + Y^{L+1})$;
end $(x_h \neq a_j)$;
end (a);
(b) if $h > n$ and $j > m$ then $Result := B$;
(c) if $h \leq n$ and $j > m$
(process X is continued, sequence J is at the final point) then
begin $Q := x_h \cdot ... \cdot x_n$;
$Result := B \cdot (\epsilon + Q \cdot (Y^1 + ... + Y^L))$;
end (b);
(d) if $h > n$ and $j \leq m$ *(process X is at the final point, sequence J is continued)* then
begin
$Y^{L+1} := a_j \cdot ... \cdot a_m$,
if there exists an l such that $a_j = y_1^l$ then
begin
for all $l = 1..L$ $Y^l := PROCESS(Y^l, Y^{L+1})$;
$Result := B \cdot (Y^1 + ... + Y^L)$;
end;
if for all l $a_j \neq y_1^l$ then
$Result := B \cdot (Y^1 + ... + Y^L + Y^{L+1})$;
end (d);
end $(X = x_1 \cdot ...x_h, ...x_n)$;
end $(Z \neq \epsilon)$;

END $\{PROCESS(Z, J)\}$.