

Constraints of Behavioural Inheritance

E. E. Roubtsova, S. A. Roubtsov*

Technical University Eindhoven, Den Dolech 2, P.O. Box 513,
5600MB Eindhoven, The Netherlands.

E.Roubtsova@tue.nl, S.Roubtsov@tue.nl

Abstract. We present an approach to component inheritance and reuse which closes the gap between architectural design and process-oriented approaches. To apply inheritance checks in design and verification of a system, one should consider an inheritance relation as a *property* of the system and specify it as an inheritance constraint. To specify the inheritance constraints we offer a logic of behavioural inheritance. In a UML profile with the process tree semantics we show how to use this logic for architectural design and for verification with respect to the specified inheritance constraint.

Keywords: Constraint of behavioural inheritance, logic of behavioural inheritance, process tree semantics, UML profile, behaviour specification reuse.

1 Introduction

Inheritance of components is one of the accepted instruments for reuse of components in architectural design [1, 2]. However, in architectural approaches, like CATALYSIS [2] or ISpec [3]), and in Architecture Description Languages (ADLs), like Rapide, C2 [1] or Koala [4], the notion of component inheritance is a predefined part of the underlying metamodel. The support of the system evolution in those approaches is restricted by structural subtyping [1] of components. However, the structural subtyping relation allows defining an infinite set of behaviour inheritance relations on parent and child components. The behaviour of a parent can be repeated in a child before or after some new behaviour fragments, it can be repeated for a specific part of the child behaviour or it can be divided into parts by some new behaviour fragments. Thus, a component-inheritor specification can satisfy one of the behavioural inheritance relations and not satisfy another. In practice, this usually becomes clear only after producing and testing the behaviour specification of a component-inheritor, because the current approaches to architectural design do not direct and help designers to think about the necessary behavioural inheritance relation in advance. Consequently, this causes semantic mistakes in architectural design.

There are process-oriented architectural approaches, like SADL [1], which represent ordering constraints among sub-processes of a process. Those approaches come closer to the problem of component behavioural inheritance. But

* The research of S.A.Roubtsov was partly supported by PROGRESS (*STW EES5141*) and EMPRESS (*ITEA 01003*) projects.

the component behavioural inheritance [5] is defined for the process approaches as a finite set of potential inheritance relations on processes representing components. The relations are classified on the basis of the back transformation of a component-inheritor specification to a component-parent specification. So, if it is possible to transform a component-inheritor specification to a component-parent specification, then a designer can prove that the inheritance of some type is correct. However, the process-oriented approaches do not give us any clue of where to use one or another type of behavioural inheritance relations and how to specify such relations, i.e. the notion of behavioural inheritance given in the process theory [5] has little connection with the tasks of architectural design.

In this paper we present an approach to component inheritance and reuse which closes the gap between architectural design and process-oriented approaches. We suggest that, to apply inheritance checks in design and verification of a system, one should consider an inheritance relation as a *property* of the system. Moreover, we define such a property in terms of architectural design. Any particular type of behavioural inheritance cannot be correct or incorrect in itself. It is for a designer to decide which type of possible behavioural inheritance relations fits the case in question and, then, to prove that such a type holds in the design specification.

In verification methods, specification of properties is always based on an abstraction chosen for the system specification [6, 7]. Our system is a component exchanging messages with the environment and other components. We consider a *behavioural pattern* containing sequences, alternatives and cycles of such messages (e.g., operation calls and returns) as a unit of system specification.

When a new component inherits a parent component, we should give a specification of how exactly the parent's behavioural pattern should be reused. Designers may have different ideas on how to reuse a particular behavioural pattern. One case demands establishing some conditions on reuse of the pattern, another case - fulfilling the pattern for all alternatives. So, there is no sole behavioural inheritance specification, but an infinite set of them. The chosen inheritance specification should become a property of the inheritor's design specification and this property must be kept.

Consequently, we consider behaviour inheritance relations as *constraints*. The standard constraint language OCL [8] is not suitable for the specification of behavioural inheritance relations because it does not manipulate processes as abstract elements. Because of that we offer a *logic of behavioural inheritance* to define inheritance constraints. Constraint languages can be extended on the basis of this logic.

An inheritance constraint describes how the process of a component-parent can appear in the process of a component-child. We consider the component-child to be a correct inheritor of the component-parent with respect to the specified behavioural inheritance constraint if the inheritance constraint holds for the process of the component-child. A predicate of the logic of behavioural inheritance represents the place of the parents's process tree in the child's process tree. A process tree is an abstract variant of a computation tree. Formulas of our

logic describe properties of this computation tree. So, our logic is a computation tree logic with a process interpretation.

The logic of behavioural inheritance allows designers to specify what kind of behavioural inheritance they would like to achieve. Moreover, the logic defines types of constraints as logical units and allows us to put a corresponding technique to each type of constraint to prove that this constraint holds. So, the logic provides methodological support for reuse of component behaviour in architectural design.

The paper is organized as follows. In Section 2 we define a component behavioural pattern as a process and a corresponding process tree. An example of the process tree semantics is given for a component specification profile in the UML. In this profile we also demonstrate specification of components using inheritance. Section 3 describes our logic of behavioural inheritance and explains how to use this logic for architectural design and for proving correctness of component behavioural inheritance. Section 4 concludes the paper.

2 A Behavioural Pattern as a Process Tree

In our approach, a **component specification** is a process p of **type**

$$P = (A, SP, T) [5], p \in P, \text{ where}$$

- A is a finite set of actions.
- $SP = \{sp, sp_1, sp_2, \dots, sp_F\}$ is a finite set of abstract states from the unique initial state sp to the unique final state sp_F .
- T is a set of transitions. Transition $t \in T$ defines a triple (sp', sp'', a) , such that state sp'' is reachable from state sp' as a result of action $a \in A$: $sp' \xrightarrow{a} sp''$.

We construct a *process tree* for a component behaviour specification.

A **process tree** is a process graph [9] $G_p = (N, E)$ which has a unique path from the node *root* to every other node. Each process tree corresponds to its process p so that:

- Each node $n \in N$ of the process tree corresponds to an abstract state from set SP . The *root* corresponds to the initial state sp .
- Each node, except final nodes, is labelled by the process name which represents the process starting from the state corresponding to this node. Final nodes, labelled by \surd , correspond to the final state sp_F .
- Each edge $e = (n', n'', a) \in E$ of the process tree corresponds to a transition from set T . An edge is labelled by an action $a \in A$. Edges to final nodes carry the termination label \downarrow .

Thus, a **path** in a process tree is a sequence of arcs

$$((n_1, n_2, a_1), (n_2, n_3, a_2), \dots, (n_{m-1}, \surd, \downarrow)).$$

There is a unique sequence of actions that corresponds to each path:

$a_1, a_2, \dots, a_{m-1} \downarrow$. A path which starts from the root is called a *root path*. The

node labels, the final nodes labelled by \surd and the edges labelled by \downarrow can be omitted [9] to simplify process tree graphical representation.

If a component behaviour specification contains cycles, then we represent each cycle by two paths: one path for the cycle's body and the other path for the cycle's exit. Repeated cycle's bodies are replaced by dots: "...".

Computation trees similar to our process tree are widely accepted as internal models for specification languages. Computation tree semantics has been defined for automata specifications [6, 10] and for UML profiles containing statecharts [7]. In the next subsection, we define a process tree semantics for our UML profile for component specification and reuse.

2.1 Process Tree Semantics for Our UML Profile

Our UML profile is one in the family of UML-like languages [11]. We specify a process of a component in a role language. This role language is represented in an identified subset of the UML metamodel [12].

The elements of a process are specified in terms of roles communicating via interfaces. A role is a UML class with stereotype $\ll Role \gg$. In general, a role can have several players (instances), but we do not refer to players in this paper. An interface comprises a semantically close group of operations of a role. An interface is always provided by a particular role which implements operations of this interface. The interface can be required by other roles or, maybe, by the role itself. These *provide* and *require* relations specify actions of our process graph. To refer to a particular action we use the conventional "dot"-notation. In this notation an operation call, for example, is denoted as *role_{require}.role_{provider}.interface.operation(parameter : type)* and its return (callback) as

role_{require}.role_{provider}.interface.operation(parameter : type) : result.

The notation above provides uniqueness of operation names within the entire component specification. In all examples of this paper each interface has exactly one operation with different results. So, it is possible to use the shortened notation for action names:

role_{require}.role_{provider}.interface(parameter : type)

role_{require}.role_{provider}.interface(parameter : type) : result.

The actions and the component behavioural pattern built from such actions are specified by an **interface-role diagram** IR and a **set of sequence diagrams** $S_1 \dots S_k$.

An interface-role diagram (Fig. 1a, 2) 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. Each role $r \in R$ is depicted by a box.
- I is a finite set of interfaces depicted by circles. In this paper, each interface $i \in I$ has one operation identified by the interface name. Each operation has a set of results Res_i .

- $PI \subseteq \{(r, i) \mid r \in R, i \in I\}$ is a *provide relation* on roles and interfaces. Each role provides a finite set of interfaces. An element of the relation is depicted by a solid line between a role and an interface.
- $RI \subseteq \{(r', (r, i)) \mid r', r \in R, i \in I, (r, i) \in PI\}$ is a *require relation* on roles and interfaces. Each role requires a finite set of provided interfaces. An element of the relation is drawn by a dash arrow connecting a role and a provided interface. The arrow is directed to the interface.
- $RR \subseteq \{(r, r') \mid r, r' \in R\}$ is a *relation of inheritance* on the set of roles. An element of the relation is shown by a solid line with the triangle end $r' \rightarrow r$ directed from role-child r' to role-parent r .

A sequence diagram (Fig. 1b) is a UML sequence diagram [8]

$$S = (B, A_s, \aleph \rightarrow A_s),$$

- $B = \{b_i\}$ is a set of boxes with dash lines drawn down from each box and representing the time dimension. In our profile, box $b_i \in B$ represents a player (an instance) of a role from the interface-role diagram. We have assumed that each role has only one player, so a box represents a role.
- A_s , is a set of labelled arcs.

An arc $(b_i, b_j, l) \in A_s$ is depicted as an arrow that connects the dash line running from box b_i to the dash line running from box b_j . An arc has a label l which represents an operation, for example,

$l = \text{interface.operation(parameter)}$ for an operation call or

$l = \text{interface.operation(parameter) : result}$ for an operation return

(or $l = \text{interface(parameter)}$ and

$l = \text{interface(parameter) : result}$, if each interface has only one operation.)

- $\aleph \rightarrow A_s$, $\aleph = \{1, 2, 3, \dots\}$ is a function defined on a subset of natural numbers that orders arcs.

Process tree. From each specification in the described above profile we construct a process tree.

S-tree. A sequence diagram corresponds to a process tree $G = (N, E)$ which contains one path. We name such a tree *s-tree*: $e_1, \dots, e_k = (n_1, n_2, a_1), \dots, (n_k, n_{k+1}, a_k)$, where $e_x = (n_x, n_{x+1}, a_x)$, $x = 1, \dots, k$, $a_x = x \rightarrow (b_i, b_j, l)$, $(b_i, b_j, l) \in A_s$.

Operation Fusion: Let a *process tree* and an *s-tree* be given.

- If a root path of the *process tree* and a root subsequence of the *s-tree* have the same sequence of labels of arcs, then this path and this subsequence are fused, i.e. joined in one path.
- The first arc of the *s-tree*, the label of which differs from the label of the current arc in the root path of the *process tree*, starts a new branch from the last fused node of the *process tree*.

Process tree construction.

1. *S-tree constructed from a sequence diagram is a process tree.*
2. *The result of the fusion of a process tree with an s-tree is a process tree.*
3. *There are no other process trees.*

The detailed description of the algorithms can be found in [13].

For a component specification in our profile, the set E of arcs of the process tree $G_p = (N, E)$ is exactly defined from the set of arcs of all the sequence diagrams: $E = A_{s_1} \cup \dots \cup A_{s_k}$. In turn, the set of arcs of all the sequence diagrams $A_{s_1} \cup \dots \cup A_{s_k}$ is a multiset on the require relation set RI from the interface-role diagram of this component (some operations can be called several times). The process tree of a component can be easily transformed back to its sequence diagrams: each root path of the process tree is mapped onto an s-tree corresponding to a sequence diagram. In the next subsection, we give an example of a component specification in our profile.

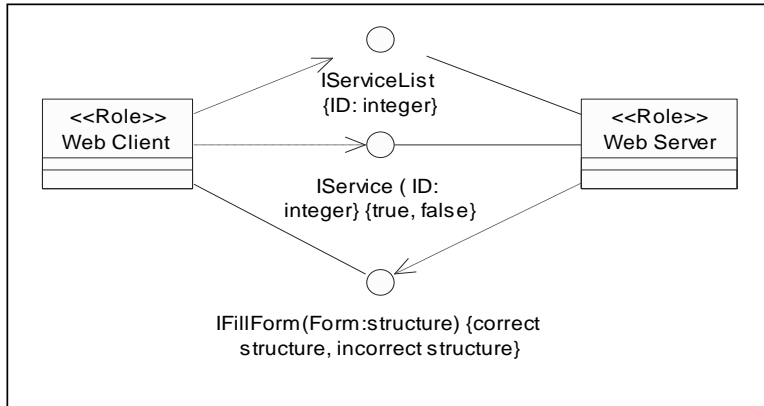
2.2 Specification of Component *Web Service* in Our UML Profile

Let us consider an abstract component *Web Service*. Like most services on the Web, this service sends back some data in response to a user's request. (Even if you buy something, Internet itself never sends you goods, it only promises you to send goods later.) The component provides an opportunity to choose one of Web services from a list. Usually, before responding, the server asks the client for some additional information. For example, to get access to search engines the client should identify a kind of information to be retrieved; to buy things in an e-shop the customer should choose them and provide data that guarantees the purchase, and so on. In all cases the process is essentially the same; the differences (what kind of response is required, what additional information is needed and how to ask it) can be hidden in the server's software. This allows us to consider such an adjustable service as a reusable component in the Web service interaction model. Fig. 1 shows the specification of component *Web Service*, which we intend to reuse.

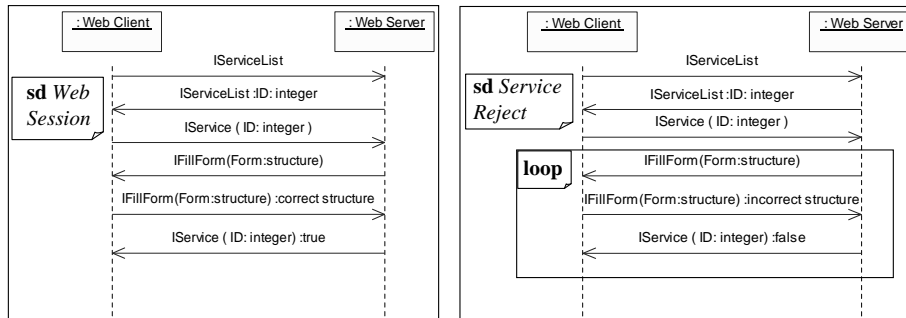
The interface-role diagram of component *Web Service* is shown in Fig. 1a. Role *Web Server* provides two interfaces: $IServiceList : \{ID : integer\}$, which returns identifier ID of a chosen service and $IService(ID:integer) : \{true, false\}$, which has two return values: *true* that means the successful result of a service request and *false* that means the unsuccessful result.

Role *Web Client* requires interfaces of *Web Server* and provides interface $IFillForm(Form:structure) : \{correct\ structure, incorrect\ structure\}$. Two types of the return value indicate two possible results of interaction via this interface: the *correct structure*, if the form is filled in correctly, and the *incorrect structure*, if some fields of the form are filled in incorrectly.

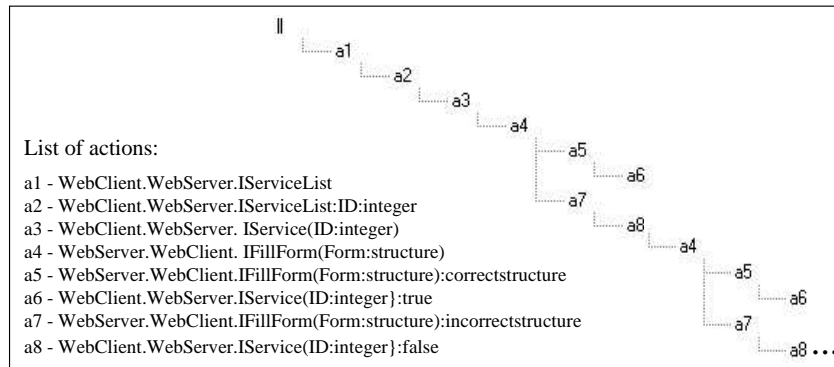
Two sequence diagrams present the behavioural model of component *Web Service* (Fig. 1b). The first sequence diagram models the successful behavioural pattern. *Web Client* chooses the service defined by parameter ID from the list and requests this service. In response *Web Server* sends back the form defined by the parameter $Form$ to be filled in. After the correct data structure has been filled in and sent to *Web Server*, it fulfils the service (return value *true*) and the session ends. The second sequence diagram corresponds to the case when the client's data for some reason is inappropriate. In such a case *Web Server* responds by the *false* return value and requests the data again.



a)



b)



c)

Fig. 1. The specification of component *Web Service*: a) interface-role diagram; b) sequence diagrams; c) process tree

This is a usual behavioural pattern for Web services: you can escape repetition of data requests by cancelling the connection or navigating to another Web page. Of course, more robust variants exist, e.g., a client may also be allowed to cancel the session within a requested form, the number of attempts may be restricted, and so on. However, in our component we have decided to rely on the common Web ideology. So, here we have a cycle, which is depicted by operator **loop** from the UML2.0 notation [14] newly adapted by OMG group.

The process tree representing behaviour of component *Web Service* is shown in Fig. 1c. In this figure and later we show action names only; the node labels, the final nodes labelled by \surd and the edges labelled by \downarrow are omitted to simplify the picture.

From action $a_4 = \text{WebServer.WebClient.IFillForm}(\text{Form} : \text{structure})$ (the request from *Web Server* to *Web Client* to fill in the *Form*) the process tree branches out: one branch ends after the service has been completed and the other runs a possibly infinite cycle. Let the name of the process of component *Web Service* be $p = \text{Web Service}$. We shall use this process as a unit of behaviour to inherit from.

2.3 Component Specification by Inheritance

Inheritance of components in interface-role diagrams is specified by the inheritance relation on roles RR .

Definition 1. *Let two interface-role diagrams be given:*

$$IR_p = (R_p, I_p, PI_p, RI_p, RR_p), \quad IR_q = (R_q, I_q, PI_q, RI_q, RR_q).$$

Interface-role diagram IR_q inherits interface-role diagram IR_p , if and only if there is an interface-role diagram $IR_{new} = (R_{new}, I_{new}, PI_{new}, RI_{new}, RR_{new})$, (Fig. 2) such that

1. $R_q = R_p \cup R_{new}$. Role sets R_p and R_{new} are disjoint, i.e. $R_p \cap R_{new} = \emptyset$.
2. $I_q = I_p \cup I_{new}$. Interface sets I_p, I_{new} are disjoint, i.e. $I_p \cap I_{new} = \emptyset$.
3. *Only new roles can inherit roles of the parent interface-role diagram. Parent roles cannot inherit new roles.*
 $RR_q = RR_p \cup RR_d$, where
 $RR_d = \{(r_p, r_d) \mid r_p \in R_p \wedge r_d \in R_d \wedge R_d \subseteq R_{new}, \wedge r_d \dashv r_p\}$, $RR_d \neq \emptyset$.
So, the relation RR_d defines subset of roles $R_d \subseteq R_{new}$ which have parents in the set R_p .
4. *Elements of the provide relation from roles-parents are duplicated in the interface-role diagram IR_q by roles-inheritors because of the inheritance relation RR_d .*
 $PI_q = PI_p \cup PI_d \cup PI_{new}$,
 $PI_d = \{(r_d, i) \mid r_d \in R_d \wedge i \in I_p \wedge (\exists r \in R_p \mid r_d \dashv r \wedge (r, i) \in PI_p)\}$.
Sets PI_p and $(PI_d \cup PI_{new})$ are disjoint, i.e. $PI_p \cap (PI_d \cup PI_{new}) = \emptyset$.
5. *Elements $(x, (r, i))$ of the require relation RI_p are duplicated in the interface-role diagram IR_q if both role r that provides interface i and role x that requires interface i are inherited.*
 $RI_q = RI_p \cup RI_{new} \cup RI_d$, where
 $RI_d = \{(x_d, (r_d, i)) \mid r_d, x_d \in R_d \wedge i \in I_p \wedge (\exists r, x \in R_p, \mid (r_d \dashv r \wedge x_d \dashv x \wedge (r, i) \in PI_p \wedge (x, (r, i)) \in RI_p)\}$.
Sets RI_p and $(RI_{new} \cup RI_d)$ are disjoint, i.e. $RI_p \cap (RI_{new} \cup RI_d) = \emptyset$.

The inheritance relation RR_d in the interface-role diagram IR_q defines a duplicating function $\rho_{RI_p}^{RI_d}$ which maps the parent require relations onto the subset of the child require relations. Elements of the set RI_d are not depicted in the interface-role diagram IR_{new} , although they are inherited from the parent.

As we have mentioned already, a name of an action in our specification includes names of the role-provider and the role-requirer. If both the $role_{provider}$ and the $role_{requirer}$ of a parent component are inherited by a new $role_{inheritor-of-the-provider}$ and a new $role_{inheritor-of-the-requirer}$ correspondingly, then the actions defined by the role-provider and the role-requirer are inherited by the new component.

The sequence diagrams of the child component are constructed from the actions specified by its interface-role diagram. If the parent process is inherited, then its actions are renamed using the duplicating function $\rho_{RI_p}^{RI_d}$ and the parent process p is transformed to the duplicated process $p' = \rho_{RI_p}^{RI_d}(p)$ which is equivalent to p under duplicating. Here and later we indicate a duplicated process by the prime mark (e.g. p').

2.4 An Example of Component Specification by Inheritance

Let us design component *Corporate Provider* which inherits component *Web Service*. The new component enables all the possibilities of component *Web Service* but "for members only". Membership is supposed to be obtained somewhere outside the Web, using security ID cards, for example. For non-members a corporate server should simply deny access. Of course, the alternative behaviour is quite rudimentary but it could easily be extended by some predefined service, a kind of survey for guests, for example. The behaviour of component *Web Service* should be inherited by the new component just in one case, for a corporate member. Therefore, the predefined process of a membership check must come before the choice of a service.

Fig. 2 shows the interface-role diagram of component *Corporate Provider*. Two new roles *Member* and *Corporate Server* interact via the two new interfaces *IMemberAccess* and *IMemberInfo*: *Member* asks for access and afterward *Corporate Server* requests information via *IMemberInfo*. Depending on the return value of *IMemberInfo* role *Corporate Server* allows or denies access.

Role *Member* inherits role *Web Client*; role *Corporate Server* inherits *Web Server* from component *Web Service*. This means that roles *Member* and *Corporate Server* are able to perform all actions which roles *Web Client* and *Web Server* use in communication. So, the new component *Corporate Provider* is able to utilize the complete parent behaviour of component *Web Service*. This is depicted in sequence diagrams shown in Fig. 3. All three sequence diagrams are started by obligatory process $q = \textit{Check Membership}$. The first two sequence diagrams are started with subprocess $s = \textit{CorrectMembership}$ after which the inherited process $p' = \textit{WebService}' = \rho_{RI_p}^{RI_d}(p)$ (which is equivalent to the parent process $p = \textit{WebService}$ under duplication) fulfils itself. The third sequence

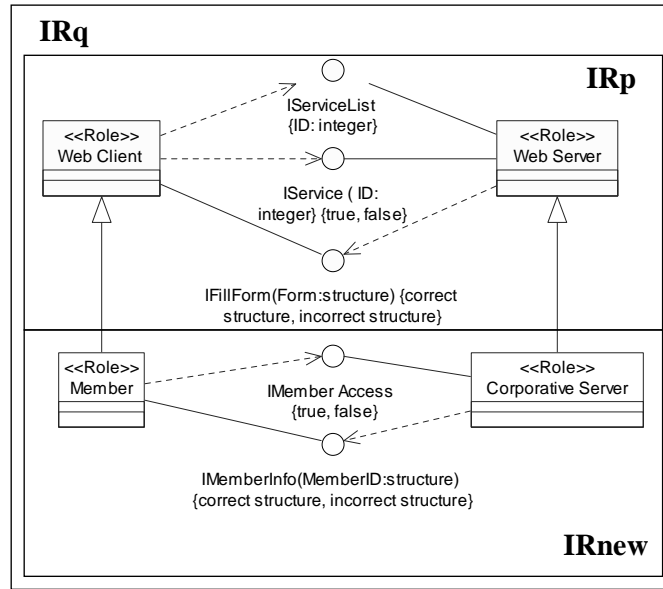


Fig. 2. The interface-role diagram of component *Corporate Provider*

diagram in Fig. 3 shows the alternative process *Incorrect Memberships*, i.e. the membership is not confirmed.

Thus, our UML profile allows a designer to specify the processes that should be inherited. The next section discusses how to help him/her formally specify the goal of inheritance and how to prove that this goal is achieved.

3 A Logic of Behavioural Inheritance

3.1 An Existential Definition of Behavioural Inheritance

Behavioural inheritance has been defined in [5]. Generalizing this definition we have the following:

Process c inherits process p if and only if there exist disjoint sets of actions $H, I \subseteq A_c \setminus A_p$, such that it is possible to derive process p from process c by

- *blocking actions from H in process c using blocking function $\delta_H(c)$;*
 $\delta_H(c) : P \rightarrow P$;
 $a \in H \rightarrow \delta_H(a) = \delta$; δ is a blocked action;
 $a \notin H \rightarrow \delta_H(a) = a$;
- *hiding actions from I in process $\delta_H(c)$ using hiding function $\tau_I(\delta_H(c))$;*
 $\tau_I(\delta_H(c)) : P \rightarrow P$;
 $a \in I \rightarrow \tau_I(a) = \tau$; τ is a hidden action;
 $a \notin I \rightarrow \tau_I(a) = a$;

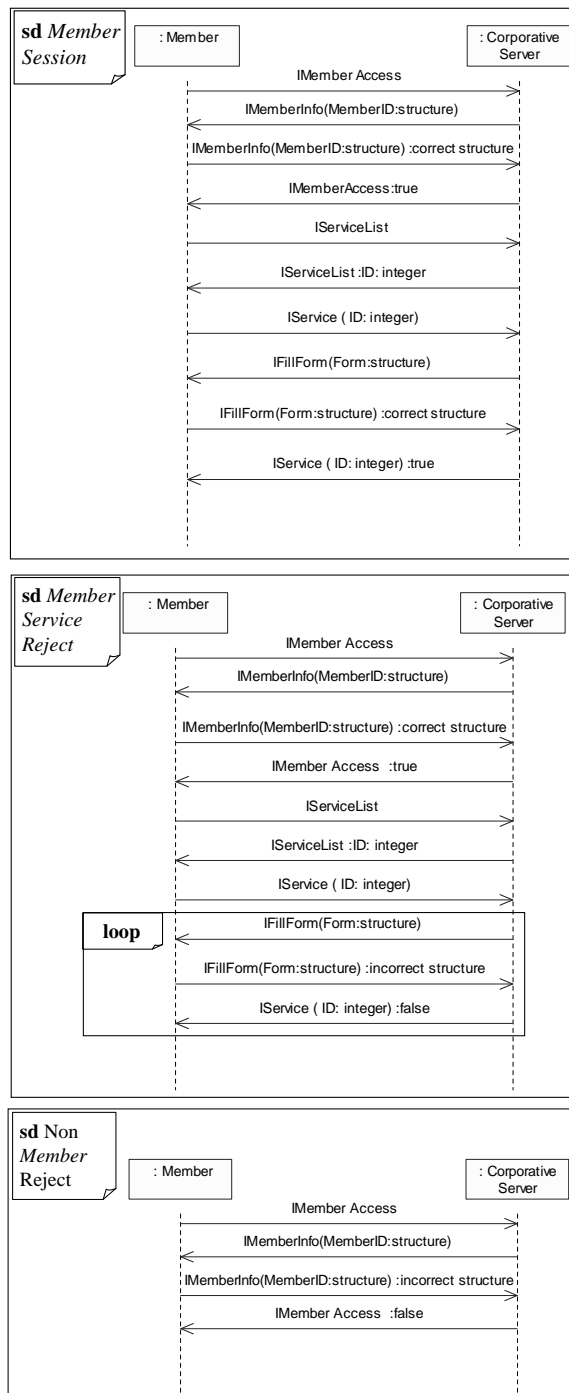


Fig. 3. The sequence diagrams of component *Corporate Provider*

- and simplifying the resultant process using axioms of ACP (Algebra of Communicating Processes) with replacement of the parallel composition by the left merge [9] and axioms for hidden and blocked actions [5]:

$$\begin{aligned}
x + \delta &= x; & \delta \cdot x &= \delta; \\
x \cdot \tau &= x; & x \cdot (\tau \cdot (y + z) + y) &= x \cdot (y + z);
\end{aligned}$$

where x, y, z are actions; δ is a blocked action; τ is a hidden action.

Intuitively, the blocking of action a means that the process which follows this action will not be considered any more. The hiding of action a makes this action invisible, but the rest of the process which follows this action is taken into consideration.

The above definition is an existential one. The definition leaves unanswered an important practical question: how would we like to inherit the parent process? In other words, the definition given in [5] does not address the tasks of architectural design.

3.2 Behavioural Inheritance from the Designer’s Point of View

Studying architectural design in practice [15] we have found that designers think about behavioural inheritance in terms of named processes (e.g. ”Check Membership” or ”Choose a Service from the List”). Composing such processes on the basis of their intuition, designers sometimes make semantic mistakes. So, designers need methodological support to use inheritance intentionally.

In this paper, we propose composing processes in the process tree model. A behavioural inheritance relation on processes of a component-child and a component-parent has to be specified by a constraint in our logic of behavioural inheritance. This logic has the process tree semantics which we define following the semantic definition given in [16].

Let AP be a set of atomic propositions. An atomic proposition $\phi \in AP$ can be of two types: $\beta_p =$ ”process p begins”; or $\epsilon_p =$ ”process p ends”. So, each atomic proposition has a parameter (the name of the parent process) represented by its index. Each inheritance constraint ψ can be specified by a formula inductively defined as follows:

$$\psi ::= \phi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 AU \psi_2 \mid \psi_1 EU \psi_2;$$

To define the semantics of the logic we construct a Kripke structure [10]: $M = (SP, T, \mu)$, where SP is a finite set of states; T is a binary relation on states which defines the initial state and the possible transitions; $\mu : SP \rightarrow 2^{AP}$ assigns true values of atomic propositions to each state.

The satisfaction relation for formulas in states $(M, sp) \models \psi$ is defined inductively:

1. $sp \models \phi$ iff $\phi \in \mu(sp)$.
2. $sp \models \neg\psi$ iff $sp \not\models \psi$.
3. $sp \models \psi_1 \wedge \psi_2$ iff $sp \models \psi_1$ and $sp \models \psi_2$.
4. $sp \models \psi_1 \vee \psi_2$ iff $sp \models \psi_1$ or $sp \models \psi_2$.

5. $sp \models \psi_1 AU \psi_2$ if for every path sp_0, sp_1, \dots with $sp = sp_0$, for some $i \geq 0$ $sp_i \models \psi_2$ and $sp_j \models \psi_1$ for $0 \leq j < i$.
6. $sp_i \models \psi_1 EU \psi_2$ if for some path sp_0, sp_1, \dots with $sp = sp_0$, for some $i \geq 0$ $sp_i \models \psi_2$ and $sp_j \models \psi_1$ for $0 \leq j < i$.

Using this logic we now give our own definition of behavioural inheritance: behavioural inheritance from the designer's point of view.

Definition 2. *Process tree c inherits process tree p if the inheritance constraint ψ_p specified for c is satisfied in the root of process tree c .*

For example, constraint $\beta_p AU \epsilon_p$ means that process tree c inherits process tree p if for every path of process tree c starting from the root there is a node where process p begins, i.e. $\beta_p = true$, and then there is a node where process p ends, i.e. $\epsilon_p = true$.

The definition of a process p includes a reachability relation on the process's states sp, sp_1, \dots, sp_F . Let $sp \xrightarrow{a_1} sp_1$ and $sp_1 \xrightarrow{a_2} sp_2$ specify the reachability relation of a process p . This relation is represented by formula $(\beta_{a_1} AU (\epsilon_{a_1} \wedge \beta_{a_2})) AU \epsilon_{a_2}$. Recursively applying the formulas of our logic to each reachability relation, it is easy to derive the logic formula which describes the reachability relation of a given process. That is why we use predicate Φ_p to represent a complete process p which starts in state sp and literally fulfils itself without interruptions till its final state. The satisfaction relation for predicate Φ_p is the following:

$sp \models \Phi_p$ if for every path sp_0, sp_1, \dots with $sp = sp_0$, for some $i \geq 0$ $sp_i \models \epsilon_p$ and for $0 \leq j < i$ $sp_j \models \beta_p$ (i.e. $sp \models \beta_p AU \epsilon_p$) and every path sp_0, sp_1, \dots, sp_i with $sp = sp_0$ and $sp_i = sp_F$ is a path of process p .

The logic of behavioural inheritance solves the following problems:

Firstly, this logic allows designers to specify what kind of behavioural inheritance they would like to achieve. Moreover, in the case of composition without modification, the behaviour specified by a constraint can be constructed automatically as a composition of process trees.

Secondly, this logic defines types of constraints (AU or EU) and allows us to set the correspondence between the type of constraints and the proving technique.

For example, to prove constraint $\beta_q AU (\epsilon_q \wedge \Phi_p)$ (Fig.4 (1)), we should choose, one after the other, each s -tree from process q and block all other s -trees. In the path with the chosen s -tree we check that process p starts and literally fulfils itself after finishing of the chosen s -tree. To prove constraint $\beta_q EU (\epsilon_q \wedge \Phi_p)$ we should check that process p starts and literally fulfils itself for at least one s -tree from q (Fig.4 (2)). For the derivable constraints these two basic techniques are combined.

Sometimes designers need to insert new processes into the parent behavioural patterns. Doing this designers usually mean to keep the parent behavioural pattern in form of one or another inheritance constraint. However, they can make mistakes *modifying* the parent process. The correctness check of modification (Fig. 5) begins when a starting node of the parent process tree has been found in the child process tree. From this point the tree transformation rules (Fig.6),

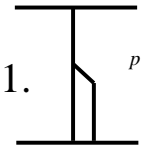
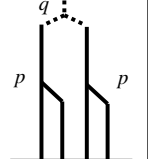
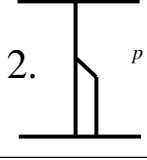
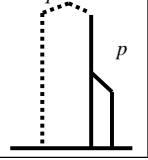
Parent process tree	Child process tree	Constraint	Derivation technique
1. 		$\beta_q AU(\epsilon_q \wedge \Phi_p)$	<ul style="list-style-type: none"> choose, one after the other, each <i>s</i>-tree from <i>q</i>, hide each new action of this <i>s</i>-tree and block all other <i>s</i>-trees; check that process tree <i>p</i> starts and literally fulfils itself after finishing of each such an <i>s</i>-tree.
2. 		$\beta_q EU(\epsilon_q \wedge \Phi_p)$	<ul style="list-style-type: none"> choose, one after the other, each <i>s</i>-tree from <i>q</i>, hide each new action of this <i>s</i>-tree and block all other <i>s</i>-trees; check that process tree <i>p</i> starts and literally fulfils itself after finishing of at least one such an <i>s</i>-tree.

Fig. 4. Examples of Behavioural Inheritance

which we have constructed on the basis of the axioms defined in [5], can be used to check correctness of insertions. Sequential insertions of new actions are hidden (axiom 3). Alternatives started by new actions with the structure corresponding to axiom 4 are transformed according to the transformation rule 4 (Fig.6). Alternatives of other structures starting by new actions are blocked using axiom 1. Axiom 2 is a restrictive one: a blocked action cannot be eliminated from the sequential branch and, this way, the point of incorrect inheritance can be found.

3.3 An Example of Behavioural Inheritance without Modification of the Parent Process

The informal inheritance constraint for component *Corporative Provider* specified in Fig. 2,3 is the following: *Only in the case of correct membership, process $p=Web Service$ is fulfilled without changes.* The formal variant of this constraint is: $\epsilon_s EU \Phi_p \wedge \neg(\neg\epsilon_s EU \Phi_p)$.

Component *Corporative Provider* is a correct inheritor of component *Web Service* if the process tree of *Corporative Provider* has a path starting from the root such that there is a node on this path where process $s = CorrectMembership$ ends and then there is a node on this path where process $p = WebService$ literally fulfils itself. Also the process tree should not contain paths where there is no end of process $s = CorrectMembership$ but process p fulfils itself.

The process tree representing behaviour of component *Corporative Provider* is shown in Fig. 7. Thinking in terms of processes designers can "drag-and-drop" processes manually, like in a Lego-game, pointing out the connections between process trees. In such a way the composing of formal constraints can be done automatically by an appropriate tool.

We have developed a prototype of such a tool (see, [17]), which is included into the Rational Rose design environment as an Add-In. The tool allows visualizing

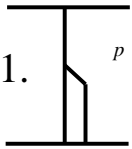
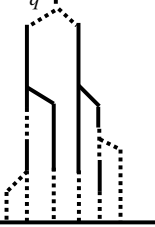
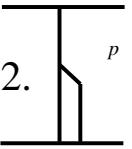
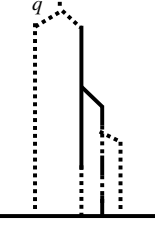
Parent process tree	Child process tree	Constraint	Derivation technique
1. 		$\beta_q AU(\epsilon_q \wedge \Phi_p)$	<ul style="list-style-type: none"> choose, one after the other, each <i>s-tree</i> from <i>q</i>, hide each new action of this <i>s-tree</i> and block all other <i>s-trees</i>; check that process tree <i>p</i> starts and fulfils itself after finishing of each such an <i>s-tree</i> provided that new actions in <i>p</i> are hidden and/or blocked according to the graph transformation rules (Fig. 6)
2. 		$\beta_q EU(\epsilon_q \wedge \Phi_p)$	<ul style="list-style-type: none"> choose, one after the other, each <i>s-tree</i> from <i>q</i>, hide each new action of this <i>s-tree</i> and block all other <i>s-trees</i>; check that process tree <i>p</i> starts and fulfils itself after finishing of at least one such an <i>s-tree</i> provided that new actions in <i>p</i> are hidden and/or blocked according to the graph transformation rules (Fig. 6)

Fig. 5. Examples of Behavioural Inheritance with Modification of a Parent Process

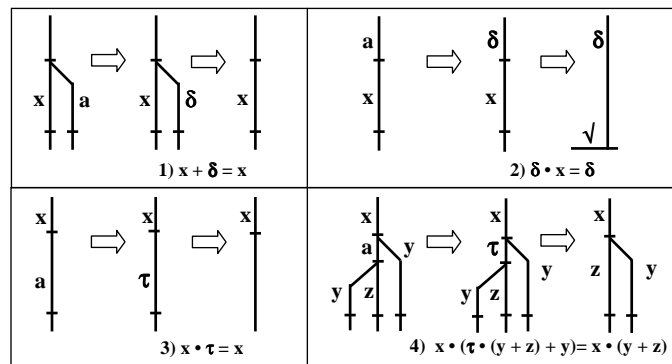


Fig. 6. Graph transformation rules; x, y, z - parent actions; a - new action; δ - blocked action; τ - hidden action.

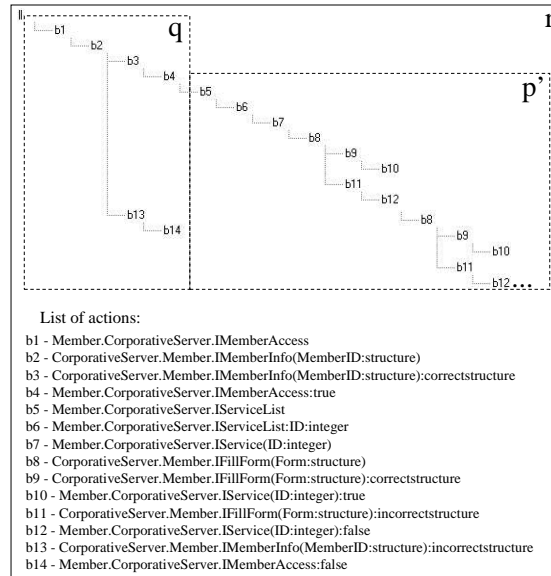
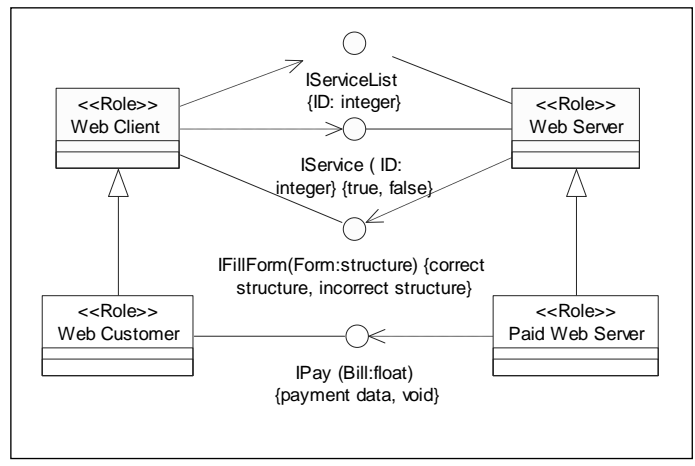


Fig. 7. Process tree of component *Corporate Provider*. Process r comprises processes p' and q .

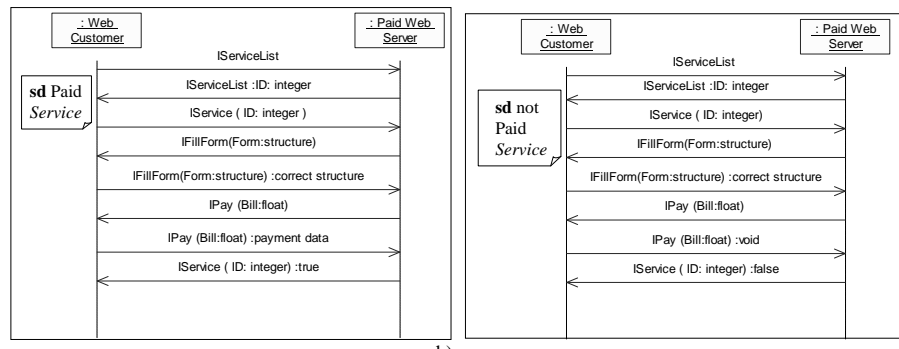
process trees and checking inheritance. The inheritance relation that we have defined on roles in the interface-role diagram (Fig. 2) allows us to duplicate actions of the parent process p . Actions of *Web Service* $a_1 \dots a_4$ (Fig. 1c) are renamed $b_5 \dots b_8$ of *Corporate Provider* (Fig. 7); actions a_5, a_6 are renamed b_9, b_{10} ; actions a_7, a_8 are renamed b_{11}, b_{12} . Predicate $\epsilon_q = \text{Correct Membeship ends}$ is satisfied after action b_4 . This path has all states of process p on it. So, according to the given constraint, component *Corporate Provider* is a correct inheritor of component *Web Service*.

3.4 An Example of Behavioural Inheritance with Modification of the Parent Process

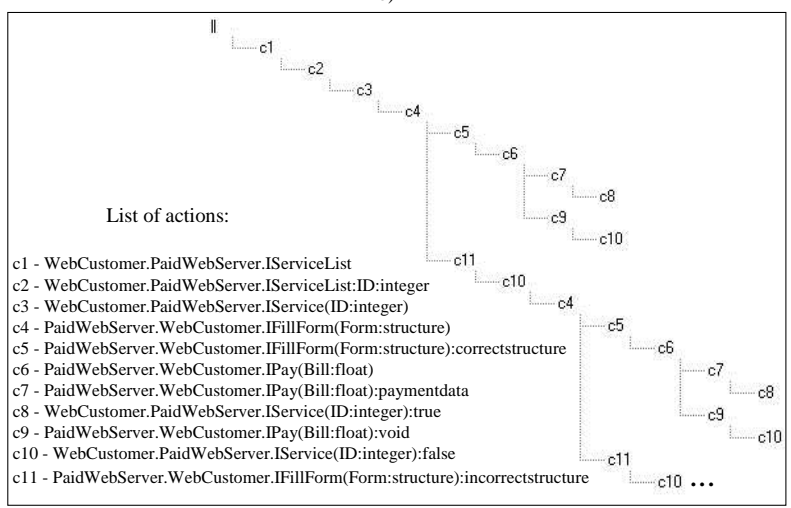
Assume that component *Paid Web Service* is constructed by inheritance from component *Web Service* with some altering: after the choice of service, but before getting one a customer should guarantee his payment sending requested information (e.g. a credit card number) back to component *Paid Web Service*. The new component should be able to utilize process $p = \text{Web Service}$ of component *Web Service* in the case of confirmed payment represented by some process $cf = \text{Confirmed payment}$. If the payment is not guaranteed, then the session has to be terminated. The informal inheritance constraint may look like this: *Only in the case of confirmed payment, component Paid Web Service fulfils process $p = \text{WebService}$.*



a)



b)



c)

Fig. 8. The specification of component *Paid Web Service*: a) interface-role diagram; b) sequence diagrams; c) process tree

The formal variant of the constraint is

$$((\beta_p EU \Phi_{cf}) AU \epsilon_p) \wedge \neg((\beta_p EU (\neg\Phi_{cf})) AU \epsilon_p).$$

Component *Paid Web Service* is a correct inheritor of component *Web Service* if the process tree of *Paid Web Service* has a path starting from the root such that there is a node on this path where process *p* begins, then there is a node on this path where process *cf* begins and literally fulfils itself and then for all paths starting from this node there is a node where process *p* ends. Also the process tree should not contain paths where process *cf* is not fulfilled but process *p* is fulfilled. The specification of component *Paid Web Service* is shown in Fig. 8.

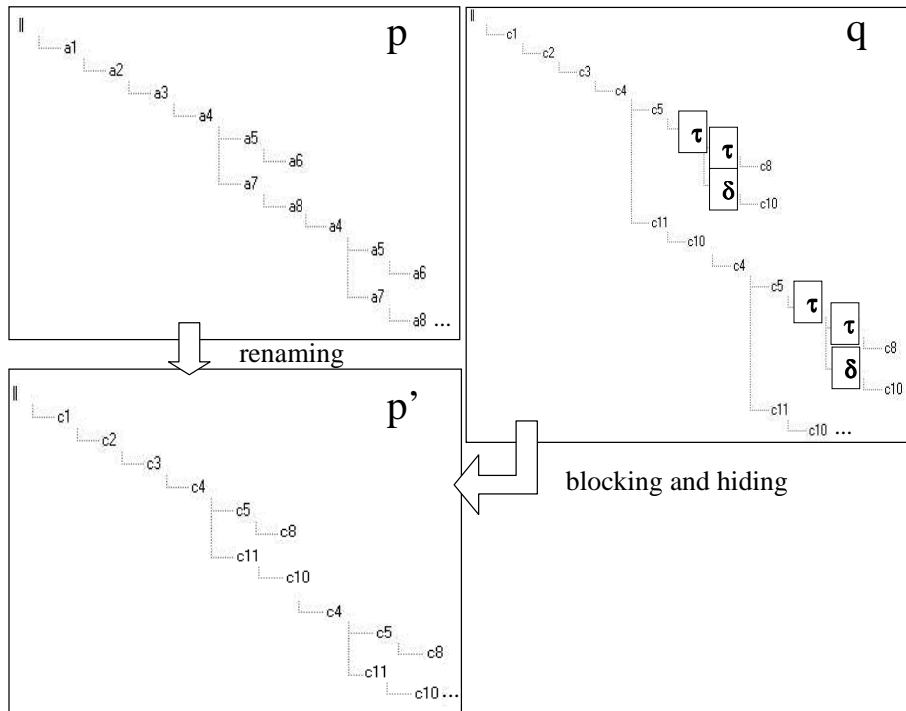


Fig. 9. *Paid Web Service* inherits *Web Service*

The interface-role diagram of component *Paid Web Service* is shown in Fig. 8 a. New roles *Web Customer* and *Paid Web Server* inherit corresponding roles of the component-predecessor. One sequence diagram corresponding to the case when the information required to perform the service is incorrect, is completely the same as for the predecessor (see the second sequence diagram in Fig. 1b.) We do not show this case in Fig. 8b. The new behaviour is provided by new interface *IPay*. Its return value *payment data* is regarded as a confirmed payment

and return value *void* corresponds to a not confirmed payment. Two sequence diagrams (Fig. 8b) show that two actions using interface *IPay* are inserted between inherited actions. (Compare these two diagrams and the first diagram in Fig. 8).

Process tree *q* representing behaviour of component *Paid Web Service* is shown in Fig. 8c. Actions $a_1, \dots, a_5, a_6, a_7, a_8$ of component *Web Service* (Fig. 8) are renamed $c_1, \dots, c_5, c_8, c_{11}, c_{10}$ (Fig. 9). In the specified process tree of the new component (Fig. 8c) process *p* starts from the root and we are looking for states where predicates $\beta_{cf} = \text{"Confirmed Payment begins"}$ and $\epsilon_{cf} = \text{"Confirmed Payment ends"}$ are satisfied. Process *cf= Confirmed Payment* is a sequence constructed from actions $c_6 = \text{IPay}(bill : float)$ and $c_7 = \text{IPay}(bill : float) : payment\ data$. We choose each such path and continue to investigate it. There exists the final state of process *p* on each of these paths. So, our inheritance constraint is satisfied. This case is of process modification without composition and the technique with hiding and blocking described in subsection 3.1 can be used to prove the constraint. Our method supports this technique by information to define the actions that should be blocked and the actions that should be hidden. The alternative *Not Confirmed Payment* is started by action $c_9 = \text{IPay}(bill : float) : void$ of new interface *IPay*. Actions of this alternative are blocked because this process is not considered in our constraint. Actions of process *cf= Confirmed Payment* are hidden because we consider this process in our constraint. This way, process tree $p' = \text{WebService}'$ (Fig. 9) is derived. Hence, we may conclude that according to the given constraint component *Paid Web Service* is a correct inheritor of component *Web Service*.

4 Conclusion

Inheritance of behaviour is a promising technique for component reuse and architectural design. Involving behavioural inheritance in design, we inevitably give birth to additional inheritance constraints on design results and we have to formulate those constraints. This is the price we have to pay to ensure correctness of reuse in design. In this paper, we have proposed to extend existing architectural approaches by specification of how a child-component inherits behaviour of its parent-component. We have defined a logic to represent these relations as behavioural inheritance constraints. The behavioural inheritance constraints can be constructed and proved with the help of tools. The technique described in this paper has been built into the specification tool [17], which we have developed to investigate correctness of components specified by inheritance.

References

1. Medvidovic N., R.Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transaction on Software Engineering **26** (2000)

2. D'Souza D.F., A.C.Wills: Objects, Components and Frameworks with UML. The CATALYSIS Approach. Addison-Wesley (1999)
3. Jonkers H.B.M: Interface-Centric Architecture Descriptions. In proceedings of WICSA, The Working IEEE/IFIP Conference on Software Architecture (2001) 113–124
4. Ommering R. van, F. van der Linden, J. Kramer, J.Magee: The Koala component model for consumer electronics software. IEEE Computer **11(3)** (2000) 78–85
5. Basten T., W.M.P. van der Aalst: Inheritance of behaviour. The Journal of Logic and Algebraic Programming **46** (2001) 47–145
6. Clarke E.M., Jr. O. Grumberg, D. A. Peled : Model Checking, Cambridge (1999)
7. Harel D., O. Kupferman: On Object Systems and Behavioural Inheritance. IEEE Transactions On Software Engineering **28** (2002) 889–903
8. OMG: Unified Modeling Language Specification v.1.5, <http://www.omg.org/technology/documents/formal/uml.htm>. (2003)
9. Baeten J.C.M., W.P. Weijland : Process Algebra. Cambridge University Press (1990)
10. Alur R., C.Courcoubetis, D.L.Dill: Model-Checking in Dense Real-Time. Information and Computation **104(1)** (1993) 2–34
11. Clark T., A. Evans, S. Kent, S. Brodsky, S. Cock: A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. (2000)
12. OMG: Requirements for UML profiles, OMG document ad99-12-32. (1999)
13. Roubtsova E.E., S.A.Roubtsov: Behavioural Inheritance in the UML to Model Software Product Lines. (Accepted for Elsevier journal "Science of Computer Programming", Editor J. Bosch. (2004))
14. OMG: UML2. <http://www.omg.org/uml/>. (2003)
15. Roubtsov S.A., E.E. Roubtsova, P. Abrahamsson: Evolutionary product line modelling. In: Proc. International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands. (2003) 13–24
16. Manna Z., Pnueli A.: The Temporal Logic of Reactive and Concurrent Systems. V.1. Specification. Springer-Verlag (1992)
17. Roubtsova E.E., S.A.Roubtsov: UML-based Tool for Constructing Component Systems via Component Behaviour Inheritance. ENTCS, Editors T.Erts, W. Fokink **V.80** (2003)