MASTER

Consistency in ISpec specifications
interface role diagrams, sequence diagrams and inheritance

van Gogh, K.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Consistency in ISpec specifications
Interface Role Diagrams, Sequence Diagrams and Inheritance

by

Koos van Gogh

Supervisor: Dr. R. Kuiper

Eindhoven, August 2003

Preface

This thesis is the result of my final assignment, completing my study at the Eindhoven University of Technology, Department of Mathematics and Computing Science, Section Formal Methods. To complete this assignment I was given the opportunity to be stationed at the Embedded Systems Institute (ESI) in Eindhoven. I would therefore like to thank all members of the ESI staff for granting me a very pleasant working environment.
I owe great gratitude to both L.C.M. van Gool and E.E. Roubtsova, who both had a major influence on this thesis by providing guidelines and suggestions for improvement of my work as well as answering my questions. Furthermore I would like to thank my supervisor R. Kuiper for granting me this assignment and investing time in helping and guiding me through it.
Special thanks also go to H.B.M. Jonkers and M.A. Reniers for taking part in the graduation committee.
Finally I would like to thank my parents for supporting me both mentally and financially during my study. Gratitude also goes to all my friends and especially my girlfriend for their inexhaustible support.

Contents

Chapter 1

Introduction

Modern industry defines new software demands, notably complex functionality, high degree of correctness and short development time. This requires new ways of software development. An important approach in this field is component technology. One of the main ideas of component technology is to provide the functionality of a piece of software (component) as a well-defined set of interfaces where an interface is a set of operations. Components can interact in a system only through these interfaces. Interfaces abstract from implementation details of a component and allow selecting a part of the functionality of the component. Another advantage of interfaces is that they can be reused on several components. This means shorter development times, but also the use of standardised interfaces, resulting in components that are easier to use. Interfaces also allow for the specification of interactions of (parts of) a system at the early stage when the components have not been chosen. A collection of interfaces, called an interface suite, together with a corresponding set of interactions thus becomes a new building block.

An important issue is consistency. This is the main topic of our investigations. We assess this issue at various levels. The overall approach is the following:
To precisely define what the consistency requirements are we provide formal semantic models, one for structure and one for behaviour. From that we argue what the consistency checks should be. Finally, we assess a tool to see how consistency checks can be automated. We identify which checks apply and what the input for such checks is. This is, e.g., relevant for tool development.

In Chapter 2 we consider consistency at the level of the structure descriptions of interface suites, described by interface role diagrams. We represent these formally, using sets and relations. We then investigate in this model consistency of single suites. Furthermore, consistency of inheritance is addressed, to enable combining, extending and refining interface suites.

In Chapter 3 we turn to behaviour, described by sequence diagrams. Again we first consider consistency for single suites. For inheritance the situation is quite open, various notions exist. In a separate Chapter, 4, we discuss some alternatives and propose a version that we consider practicable. Some examples illustrate the notion.

In Chapter 5 we assess a tool that is under development (a joint TU/e Computer Science/Technology Management and Philips effort). It appears that subtle differences, in the choice of the definition of consistency as well as in the chosen algorithm for checking, influence the outcome of checks in our example cases. The examples from Chapter 4 are used to show this.

Chapter 2

Interface Role Diagram(s)

2.1 Introduction

An interface role diagram identifies the *interfaces* and the *roles* associated with them. The interfaces are introduced to provide communication between "components", where these components are called roles in ISpec. When we compare this to Object Oriented Modelling, the roles can be seen as the object classes and an interface role diagram can more or less be compared to a (UML) Class Diagram. [[OO,UML]]
An interface can then be seen as a set of methods of one particular class and several interfaces will arrange the methods of a role (class) in any desired way. The roles can also have private methods, being methods not belonging to any of the interfaces of the role but rather methods of the role itself. These methods are not meant for external use, communication between the roles, but for internal use only.
When an interface "belongs" to a role we call it a *Provided Interface* of that particular role. The role *provides* the interface.
On the other side of the communication there can be a roles "using" an interface. We call this a *Required Interface* of those roles.
Furthermore we can have inheritance relations between roles for reusing or specialising purposes, like the inheritance relation we know from Object Oriented Modelling. We'll discuss inheritance further on in this chapter.

2.2 IR-diagram: What's in the picture?

In this section we discuss the ISpec interface role diagram through three examples. We start with a simple example,"One suite without inheritance", to introduce all the basic information an interface role diagram consists of. We introduce inheritance in section 2.2.2: "One suite, with inheritance between roles". Finally we discuss all aspects of an IR-diagram in section 2.2.3: "Multiple suites, with inheritance between suites and roles".

2.2.1 One suite without inheritance

An example of an interface role diagram is given in the picture below. It consists of only one suite and there are no inheritance relations between roles.

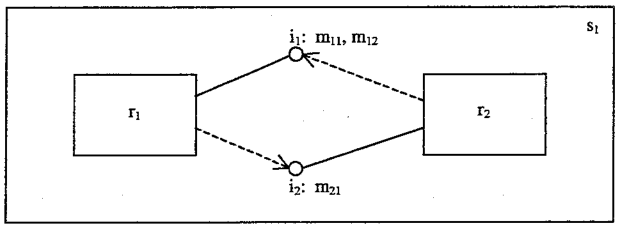


fig. 1: interface suite s_1

An interface role diagram without any inheritance relations (like the one in figure 24) consists of:

Suite	: A set of roles, interfaces and methods (and the relations between them) forms a suite, represented by a large rectangle and identified by a suitename (s_1 in the picture above).
Roles	: The smaller rectangles represent the roles of an interface-role diagram. They are identified by (unique) rolenames. $\{r_1, r_2\}$ is the set of roles in the picture above.
Provided	: The roles are drawn within a suite, meaning that they belong to that particular suite. A role is therefore *provided by* a suite, or a suite *provides* roles.
Interfaces	: Interfaces are depicted by little circles. All interfaces have unique names. ($\{i_1, i_2\}$)
Provided	: Each interface has to "belong" to one specific role. The interface is *provided* by that role. This relation is depicted by a solid line between a role and an interface.
Required	: A dashed arrow from a role to an interface means that this interface can be "used" by the role. This interface is called a *Required Interface* of a role. There may be several roles that require a certain interface.
Methods	: An interface consists of a set of methods. All methods have unique names. The interface role diagram in the picture above has the set $\{m_{11}, m_{12}, m_{21}\}$ of methods.
Provided	: Each method "belongs" to a unique interface. The interface *provides* the method.

From the above we can conclude that a suite has to contain the following sets and relations:

Definition 1. One suite without inheritance

1.1 A set of *suite identifiers.* All suites (in this case only one) have unique names.
1.2 A set of *role identifiers.* All roles have unique names.
1.3 A set of *interface identifiers.*All interfaces have unique names.
1.4 A set of *method identifiers.*All methods have unique names.

1.5 A relation *Provides Role.* Each role is related to its suite by this relation, PR.
1.6 A relation *Provides Interface.* This relation, PI, relates each interface to its role.
1.7 A relation *Requires Interface.* An interface is provided by one role, but can be used by several ones. The Requires interface relation, RI, between roles and interfaces is introduced to capture this information.
1.8 A relation *Provides Method.* Each method belongs to one, unique, interface. This relation, PM, captures this information and links each method to its interface.

2.2.2 One suite, with inheritance between roles

In this section we introduce inheritance. In ISpec interface role diagrams can have inheritance relations between two roles. These roles can be of the same suite or of different suites. The difference between the two cases lies on the implementation level, where a suite can be seen as a black box. Once a role inherits from a role of another suite (discussed in the following section) it only "knows" the specification details of that role, but not the implementation details. In some cases it can be useful, although not modular, if we do know implementation details of roles we inherit from. We can use our inheritance relation within the suite for this purpose, since one suite will be implemented as a whole.

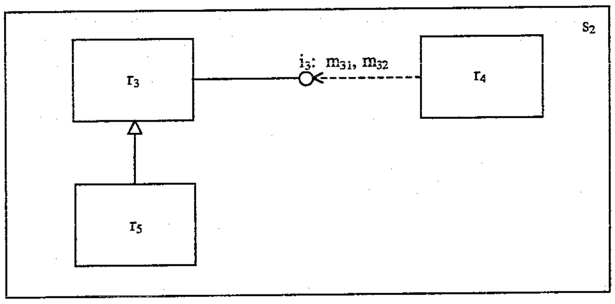


fig. 2: interface suite s_2

Suite s_2 provides roles r_3, r_4 and r_5. Methods m_{31} and m_{32} are provided by interface i_3. Interface i_3 is provided by r_3 and required by r_4.

A triangle head arrow shows the inheritance relation. It points *from* the role that is a specialisation *to* the role that is specialised. We want this relation to satisfy the following two properties:
- Irreflexivity: We do not want a role to be a specialisation of itself. - Transitivity: If a role inherits from a parent-role, it also inherits from all grandparent-roles.
These two properties, or restrictions, also imply that our inheritance relation will not be cyclic. This is nice, since we don't want inheritance to be cyclic, because inheritance means increase of information.

We will now give a definition of a suite including the inheritance relation between roles, thereby extending the definition of the previous section.

Definition 2. One suite with inheritance between roles.

2.1 A set of *suite identifiers, S.* {1.1}
2.2 A set of *role identifiers, R.* {1.2}
2.3 A set of *interface identifiers, I.* {1.3}
2.4 A set of *method identifiers, M.* {1.4}

2.5 A relation *Provides Role, PR*, between a role and its suite. {1.5}
2.6 A relation *Provides Interface, PI*, between an interface and its role. {1.6}
2.7 A relation *Requires Interface, RI*, between an interface and one or more roles. {1.7}
2.8 A relation *Provides Method, PM*, between a method and its interface. {1.8}

2.9 A relation *Specialises Role, SR*, to capture the inheritance between two roles. The first role specialises the second role, like a son inherits from its father.

2.10 Restriction: Relation SR has to be irreflexive.
2.11 Restriction: Relation SR has to be transitive.

2.2.3 Multiple suites, with inheritance between suites and roles

As we mentioned in the previous section, we can both have inheritance relations between roles of the same suite and between roles of different suites. The latter is the more restrictive one in the sense that we only know the specification of the included, inherited, suite. In this section we introduce the inheritance relation between suites, used to specify which suite will be included by the other. The roles of a suite can then inherit from the roles of the included suite. In the following example the suites of our two previous examples will be combined to one new interface role diagram.

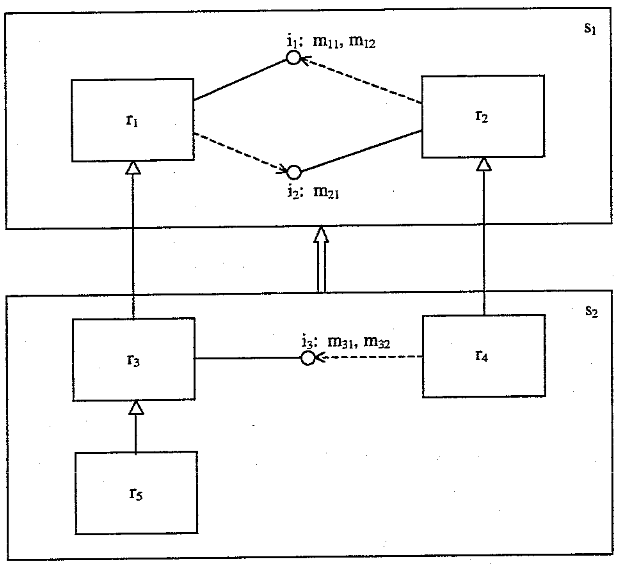


fig. 3: interface suites s_1 and s_2 combined

The IR-diagram of figure 3 describes an extension of suite s_1 by suite s_2 using inheritance. Suites s_1 and s_2 are the ones described in the previous two sections. We can now deduce two kinds of inheritance relations, between suites and between roles:
- suites: The fact that there is a double arrow from s_2 to s_1, tells us that suite s_2 *inherits from* suite s_1. This means that roles of s_2 can inherit from roles of s_1.
- roles: The triangle head arrows identify the inheritance relation between roles. We see that these arrows can be drawn within a single suite, discussed in the previous section, as well as between suites.

Restrictions/requirements:

Multiple suites
If a role requires an interface, this interface has to be part of the same suite. Therefore the role that provides the interface has to be provided by the same suite as the requiring role. In other words: One suite provides both the providing role as the requiring role(s) of an interface. Graphically this means that if we draw a dashed arrow across suite borders, we will not create a valid IR-diagram. We are then violating the so called *closed world assumption.*

Inheritance
Both inheritance relations have to be irreflexive and transitive. A suite cannot be a specialisation of itself (irreflexive) and when a suite (son) inherits from a suite (father) that in its turn inherits from yet another suite (grandfather) the son inherits from the grandfather as well (transitive). The same goes for roles.

If a role is a specialisation of a role from another suite, then the suites have to inherit from each other accordingly. In our example this means that role r_3 can only inherit from role r_1 (from another suite) as long as the suite of role r_3, being s_2, inherits from the suite of role r_1, being s_1. We can see that this requirement is met because of the double arrow that points from s_2 to s_1.

Our three given examples have discussed all properties and requirements our ISpec interface role diagrams have. We will summarize these in the following definition.

Definition 3. Interface Role Diagram.

3.1 A set of *suite identifiers, S.* {2.1}
3.2 A set of *role identifiers, R.* {2.2}
3.3 A set of *interface identifiers, I.* {2.3}
3.4 A set of *method identifiers, M.* {2.4}

3.5 A relation *Provides Role, PR*, between a role and its suite. {2.5}
3.6 A relation *Provides Interface, PI*, between an interface and its role. {2.6}
3.7 A relation *Requires Interface, RI*, between an interface and one or more roles. {2.7}
3.8 A relation *Provides Method, PM*, between a method and its interface. {2.8}

3.9 A relation *Specialises Role, SR*, between two roles. {2.9}
3.10 A relation *Specialises Suite, SS*, to capture the inheritance between two suites.

3.11 Restriction: Relation SR has to be irreflexive. {2.10}
3.12 Restriction: Relation SR has to be transitive. {2.11}
3.13 Restriction: Relation SS has to be irreflexive.
3.14 Restriction: Relation SS has to be transitive.
3.15 Restriction: Interfaces are required and provided by the same suite.
3.16 Restriction: Specialisation of roles (SR) of different suites has to imply that the according suites are also related (SS).

Remark: ISpec visualisation In this paper we use a visualisation of interface role diagrams with multiple suites, that slightly differs from the usual ISpec visualisation. The difference is shown below.

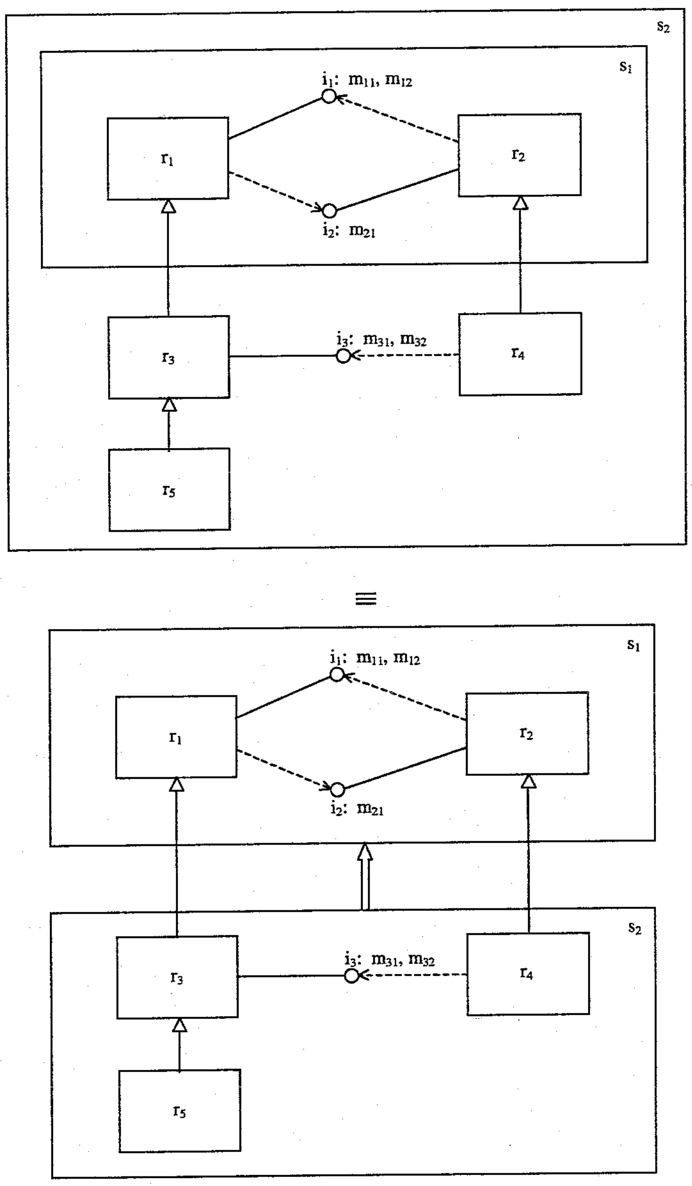


fig. 4: Two identical representations of an interface suite

As we can see there are two different (but equivalent) ways of representing the inheritance of an interface suite to create a new one. Although the ISpec-way of drawing corresponds to the uppermost diagram, we prefer the lower one for two reasons:
- Inheriting an interface suite corresponds with including a software component in Object Oriented programming. We use the double arrow to explicitly describe this includes relation, where the box-in-box notation leaves it implicit.
- Multiple inheritance will lead to the drawing of boxes in boxes in boxes using the ISpec-way. Using the double arrow for suite inheritance will clarify the

picture.
Both ways are identical however and have just as much expressive power.

2.3 Formal representation

In this section we discuss a formal representation of interface role diagrams. First we introduce some formal notation we are going to use.

Notation

We will introduce types $\mathbb{S}, \mathbb{R}, \mathbb{I}$ and $\mathbb{M}$ for respectively Suite-, Role-, Interface and Method-identifiers. All elements of our set of suite identifiers S will have to be of type $\mathbb{S}$, which means that S has to be an element of the *power set* of $\mathbb{S}$: $S \in \wp(\mathbb{S})$.

Regarding relations we introduce following notation:
The type $A \mathrel{\succ\!\!\!\!-\!\!\!\!\prec} B$ consists of all relations whose range is contained in A and whose domain is contained in B.
The type $A \mathrel{\succ\!\!\!\!-\!\!\!\!\dashv} B$ consists of all relations whose range is contained in A and whose domain is equal to B.
The type $A \mathrel{\leftarrow\!\!\!\!\prec} B$ consists of all relations whose range is contained in A and whose domain is contained in B *and* where each input ($\in B$) is related to at most one output ($\in A$).
The type $A \mathrel{\leftarrow\!\!\!\dashv} B$ consists of all relations whose range is contained in A and whose domain is equal to B *and* where each input ($\in B$) is related to exactly one output ($\in A$).
$R\!>$: The domain of relation R. If $R \in A \mathrel{\leftarrow\!\!\!\dashv} B$, $R\!> = B$.
$R\!<$: The range of relation R. If $R \in A \mathrel{\leftarrow\!\!\!\dashv} B$, $R\!< \subseteq A$.
If we have a relation R, $R \in A \mathrel{\leftarrow\!\!\!\dashv} B$ means that R is a relation of type $A \mathrel{\leftarrow\!\!\!\dashv} B$.
With regard to a relation we will define the following notation:
$\{a \frown b\} \subseteq R \;\equiv\; a\,(R)\,b$: Output a is linked to input b by relation R. We can use the notation $R.b{=}a$ as well, but only in the case of relations that relate input b to exactly one output a.

An Interface Role diagram, consisting of a set of interface suites, with their inheritance relations, is correct in terms of ISpec rules if it is of type $\mathbb{PROJ}$, which stands for Project. We introduce this project type here to formally specify the properties and restrictions a set of interface suites has to satisfy.
We construct a 10-tuple to represent a project, consisting of:
- 4 sets of identifiers: Suites (S), Roles (R), Interfaces (I) and Methods (M).
- 4 relations not concerning inheritance: Provides Role (PR), Provides Interface (PI), Requires Interface (RI) and Provides Method (PM).
- 2 relations concerning Inheritance: Specialises Suite (SS) and Specialises Role (SR).

This leads to a 10-tuple of the form $(S, R, I, M, PR, PI, RI, PM, SS, SR)$. Now we have to formulate the properties and restrictions our 10-tuple has to obey to be a correctly defined project. For example we want the relation PI (provides interface) to have as domain all interface identifiers (I), a range consisting of role identifiers (R) and we want the relation to be functional. Together this means that each interface is provided by exactly one role.

We will now introduce the following formal definition of a project (a set of interface suites):

Definition 4. Formal definition of an Interface Role Diagram (Project).

$$
\begin{array}{ll}
\mathbb{PROJ} = \{ & (S, R, I, M, PR, PI, RI, PM, SS, SR) \\
 & \mid S \in \wp(\mathbb{S}) \\
 & \;\; R \in \wp(\mathbb{R}) \\
 & \;\; I \in \wp(\mathbb{I}) \\
 & \;\; M \in \wp(\mathbb{M}) \\
 & \;\; PR \in S \leftarrow\!\!\!\dashv R \\
 & \;\; PI \in R \leftarrow\!\!\!\dashv I \\
 & \;\; RI \in R \rightarrowtail\!\!\!\dashv I \\
 & \;\; PM \in I \leftarrow\!\!\!\dashv M \\
 & \;\; SS \in S \rightarrowtail\!\!\!\prec S \\
 & \;\; SR \in R \rightarrowtail\!\!\!\prec R \\
 & \;\; \forall(x, y, z \mid y \,(SS)\, x \wedge x \,(SS)\, z \mid y \,(SS)\, z) \\
 & \;\; \forall(x, y, z \mid y \,(SR)\, x \wedge x \,(SR)\, z \mid y \,(SR)\, z) \\
 & \;\; \forall(x \mid\mid \neg(x \,(SS)\, x)) \\
 & \;\; \forall(x \mid\mid \neg(x \,(SR)\, x)) \\
 & \;\; \forall(r, r' \mid r' \,(SR)\, r \mid PR.r' = PR.r \vee PR.r' \,(SS)\, PR.r) \\
 & \;\; \forall(i, r \mid r \,(RI)\, i \mid PR.r = PR.(PI.i)) \\
\} &
\end{array}
$$

Meaning:

$S \in \wp(\mathbb{S})$	: set of suite identifiers. All suite identifiers are of type $\mathbb{S}$ {3.1}
$R \in \wp(\mathbb{R})$	: set of role identifiers, of type $\mathbb{R}$ {3.2}
$I \in \wp(\mathbb{I})$	: set of interface identifiers, of type $\mathbb{I}$ {3.3}
$M \in \wp(\mathbb{M})$	: set of method identifiers, of type $\mathbb{M}$ {3.4}
$PR \in S \leftarrow\!\!\!\dashv R$	: provides role. Each role belongs to exactly one suite. {3.5}
$PI \in R \leftarrow\!\!\!\dashv I$	: provides interface. Each interface is provided by exactly one role. {3.6}
$RI \in R \rightarrowtail\!\!\!\dashv I$	: requires interface. An interface is required by 1 or more roles. {3.7}
$PM \in I \leftarrow\!\!\!\dashv M$	: provides method. A method is provided by exactly one interface. {3.8}
$SS \in S \rightarrowtail\!\!\!\prec S$	: specialises suite. A suite can use and/or can be used by zero or more other suites for specialisation purposes. {3.10}
$SR \in R \rightarrowtail\!\!\!\prec R$	: specialises role. A role can be a specialisation of zero or more other roles. {3.9}

$\forall(x, y, z \mid y\,(SS)\,x \wedge x\,(SS)\,z \mid y\,(SS)\,z)$	: The relation SS (Specialises Suite) is transitive. {3.14}
$\forall(x, y, z \mid y\,(SR)\,x \wedge x\,(SR)\,z \mid y\,(SR)\,z)$	: The relation SR (Specialises Role) is transitive. {3.12}
$\forall(x \mid\mid \neg(x\,(SS)\,x))$	: The relation SS is irreflexive. {3.13}
$\forall(x \mid\mid \neg(x\,(SR)\,x))$	: The relation SR is irreflexive as well. {3.11}
$\forall(r, r' \mid r'\,(SR)\,r \mid PR.r' = PR.r \vee PR.r'\,(SS)\,PR.r)$	: If a role r' specialises a role r, either roles r and r' are from the same suite or the suite that provides r' has to be a specialisation of the suite that provides r. {3.16}
$\forall(i, r \mid r\,(RI)\,i \mid PR.r = PR.(PI.i))$	: If an interface is required by a role, the interface has to be provided by a role of the same suite. {3.15}

One of the main reasons we use inheritance is to reuse the interfaces of the parent roles. If such a parent role provides an interface, we say that a child role provides that interface as well. The child role may also overwrite (specialise) the interface, as long as it meets the specification requirements of the interface. We will not treat the detailed specification of interfaces and its methods in this paper, for that we refer to [[6]].
From the above we could conclude that an interface can be provided by more than one role, where in the (formal) definition we explicitly state that an interface has to be provided by exactly one role. When we said that a child role provides the interface as well, we should have said that a child role inherits the providing of the interface. A child role has the *obligation* to "provide" its parents interfaces as well.
Almost the same holds for the requiring of an interface. The only difference is that a child role is not *obligated* to require all interfaces that its parent requires, but it has the *right* to do so. One (or several) role(s) of a particular suite will require an interface, they wil be related by the Requires Interface relation (RI), and all descendants of the(se) role(s) will have the right to require this interface as well. The descendants will not be related by the interface directly (i.e. via the Requires Interface relation), but indirectly. They will be related to the role that requires the interface via the Specialises Role relation (SR) and therefore earn the right to the interface as well.
We will now construct two functions *obligs* and *rights* that assign to a role of a project all interfaces that role has to "provide" respectively may "require".

$$obligs \in (\wp(\mathbb{I}) \longleftarrow\!\!\!\prec \mathbb{R}) \longleftarrow\!\!\!\mid \mathbb{PROJ}$$

$$r \in (obligs.(S, R, I, M, PR, PI, RI, PM, SS, SR))\!>$$

$$\equiv$$

$$r \in R$$

$$(obligs.(S, R, I, M, PR, PI, RI, PM, SS, SR)).r'$$

$$=$$

$$\{i \mid r' \langle PI \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI \rangle\, j\} \mid r' \langle SR \rangle\, r \mid r\}$$

$$rights \in (\wp(\mathbb{I}) \longleftarrow\!\!\!\prec \mathbb{R}) \longleftarrow\!\!\!\mid \mathbb{PROJ}$$

$$r \in (rights.(S, R, I, M, PR, PI, RI, PM, SS, SR))\!>$$

$$\equiv$$

$$r \in R$$

$$(rights.(S, R, I, M, PR, PI, RI, PM, SS, SR)).r'$$

$$=$$

$$\{i \mid r' \langle RI \rangle\, i\} \cup \bigcup\{\{j \mid r \langle RI \rangle\, j\} \mid r' \langle SR \rangle\, r \mid r\}$$

obligs: A role has to provide its 'own' interfaces, but besides that it also has the obligation to provide the interfaces of its ancestors. The function *obligs* returns the set of interfaces a role (of a certain project) has to provide.

rights: If a role requires an interface, the role has the right to use that interface. Rights are inherited through specialisation as well, the function *rights* returns the interfaces a role may require.

2.3.1 Checklist

- All Suite names are disjoint.
- All Role names are disjoint.
- All Interface names are disjoint.
- All Method names are disjoint.

- All roles belong to a suite and to one suite only.
- All interfaces belong to a role and to one role only.
- All methods belong to an interface and to one interface only.

- The suite-inheritance relation is both irreflexive and transitive.
- The role-inheritance relation is both irreflexive and transitive.

- A role may inherit from a role of the same suite, or from a role of another suite. In the latter case the suites of the roles have to be related by the suite-inheritance relation accordingly.

- Interfaces are required and provided by roles of the same suite.

2.4 Examples

We will now discuss some examples of projects to illustrate that the complex looking 10-tuple is actually really straightforward. We will revisit the examples used in chapter 2.2 and show how a (set of) interface suite(s) will be represented formally.

2.4.1 One suite without inheritance

We will start again with the simple interface role diagram of chapter 2.2.1 that consists of one suite only, where there is no inheritance between roles. We will construct a project, $Proj_1$, and show that our interface role diagram is indeed correctly defined since all our requirements are met.

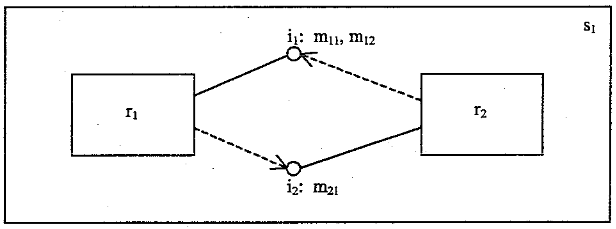


fig. 5: interface suite s_1

$Proj_1 = (S_1, R_1, I_1, M_1, PR_1, PI_1, RI_1, PM_1, SS_1, SR_1)$

With:

$$\begin{aligned}
S_1 &= \{s_1\}\\
R_1 &= \{r_1, r_2\}\\
I_1 &= \{i_1, i_2\}\\
M_1 &= \{m_{11}, m_{12}, m_{21}\}\\
PR_1 &= \{s_1 \rightsquigarrow r_1, s_1 \rightsquigarrow r_2\}\\
PI_1 &= \{r_1 \rightsquigarrow i_1, r_2 \rightsquigarrow i_2\}\\
RI_1 &= \{r_1 \rightsquigarrow i_2, r_2 \rightsquigarrow i_1\}\\
PM_1 &= \{i_1 \rightsquigarrow m_{11}, i_1 \rightsquigarrow m_{12}, i_2 \rightsquigarrow m_{21}\}\\
SS_1 &= \varnothing\\
SR_1 &= \varnothing
\end{aligned}$$

As we can see all suite-, role-, interface- and methodnames are automatically disjoint, since our sets cannot contain two identical elements. Two roles with the identical names drawn in an interface role diagram, an error, would become one and the same role in our formal representation.
Furthermore we can see that the relations all satisfy their domain and range restrictions:

$PR_1 \in S_1 \leftarrow\!\!\dashv R_1$: $PR_1{}^{>} = R_1$ and $PR_1{}^{<} \subseteq S_1$. All elements of R_1 are mapped to exactly one element of S_1, $\forall(r, s_i, s_j \mid s_i \,(PR_1)\, r \wedge s_j \,(PR_1)\, r \mid s_i{=}s_j)$.

$PI_1 \in R_1 \leftarrow\!\!\dashv I_1$: $PI_1{}^{>} = I_1$, $PI_1{}^{<} \subseteq R_1$ and $\forall(i, r_i, r_j \mid r_i \,(PR_1)\, i \wedge r_j \,(PR_1)\, i \mid r_i{=}r_j)$.

$RI_1 \in R_1 \rightarrowtail\!\!\dashv I_1$: $RI_1{}^{>} = I_1$ and $RI_1{}^{<} \subseteq R_1$. Note: If we had an additional role r_3 here, we would have been allowed to add $\{r_3 \rightsquigarrow i_1\}$ to relation RI_1 but not to relation PI_1. This because an interface may be required by several roles, but only provided by one.

$PM_1 \in I_1 \leftarrow\!\!\dashv M_1$: $PM_1{}^{>} = M_1$, $PM_1{}^{<} \subseteq I_1$ and $\forall(m, i_i, i_j \mid i_i \,(PR_1)\, m \wedge i_j \,(PR_1)\, m \mid i_i{=}i_j)$.

$SS_1 \in S_1 \rightarrowtail\!\!\prec S_1$: Since relation SS_1 is empty, both its domain and its range are contained in S_1.

$SR_1 \in R_1 \rightarrowtail\!\!\prec R_1$: Analogous.

Since both relations SS_1 and SR_1 are empty, the four restrictions concerning transitivity and irreflexivity of the inheritance relations are automatically satisfied.
$\forall(i, r \mid r \,(RI_1)\, i \mid PR_1.r = PR_1.(PI_1.i))$: This restriction also holds:

$r_1 \,(RI_1)\, i_2$: $PR_1.r_1{=}s_1$ and $PR_1.(PI_1.i_2){=}PR_1.r_2{=}s_1$
$r_2 \,(RI_1)\, i_1$: $PR_1.r_2{=}s_1$ and $PR_1.(PI_1.i_1){=}PR_1.r_1{=}s_1$

We can conclude dat $Proj_1$ is a correctly defined project, $Proj_1 \in \mathbb{PROJ}$, since all restrictions are met.
Since the inheritance relation SR_1 is empty, the obligations and rights of the roles of $Proj_1$ can be deduced easily, they will simply be the provided respectively required interfaces of those roles.

$(\mathit{obligs}.\mathit{Proj}_1).r_1$
$=$ {definition of *obligs*}
$\{i \mid r_1\,(PI_1)\,i\} \cup \bigcup\{\{j \mid r\,(PI_1)\,j\} \mid r_1\,(SR_1)\,r \mid r\}$
$=$ $\{SR_1{=}\varnothing\}$
$\{i \mid r_1\,(PI_1)\,i\}$
$=$ $\{\{r_1{\leadsto}i_1\} \subseteq PI_1\}$
$\{i_1\}$

$(\mathit{obligs}.\mathit{Proj}_1).r_2$
$=$ {definition of *obligs*}
$\{i \mid r_2\,(PI_1)\,i\} \cup \bigcup\{\{j \mid r\,(PI_1)\,j\} \mid r_2\,(SR_1)\,r \mid r\}$
$=$ $\{SR_1{=}\varnothing\}$
$\{i \mid r_2\,(PI_1)\,i\}$
$=$ $\{\{r_2{\leadsto}i_2\} \subseteq PI_1\}$
$\{i_2\}$

$(\mathit{rights}.\mathit{Proj}_1).r_1$
$=$ {definition of *rights*}
$\{i \mid r_1\,(RI_1)\,i\} \cup \bigcup\{\{j \mid r\,(RI_1)\,j\} \mid r_1\,(SR_1)\,r \mid r\}$
$=$ $\{SR_1{=}\varnothing\}$
$\{i \mid r_1\,(RI_1)\,i\}$
$=$ $\{\{r_1{\leadsto}i_2\} \subseteq RI_1\}$
$\{i_2\}$

$(\mathit{rights}.\mathit{Proj}_1).r_2$
$=$ {definition of *rights*}
$\{i \mid r_2\,(RI_1)\,i\} \cup \bigcup\{\{j \mid r\,(RI_1)\,j\} \mid r_2\,(SR_1)\,r \mid r\}$
$=$ $\{SR_1{=}\varnothing\}$
$\{i \mid r_2\,(RI_1)\,i\}$
$=$ $\{\{r_2{\leadsto}i_1\} \subseteq RI_1\}$
$\{i_1\}$

2.4.2 One suite, with inheritance between roles

We will construct project Proj_2, from the example of 2.2.1.

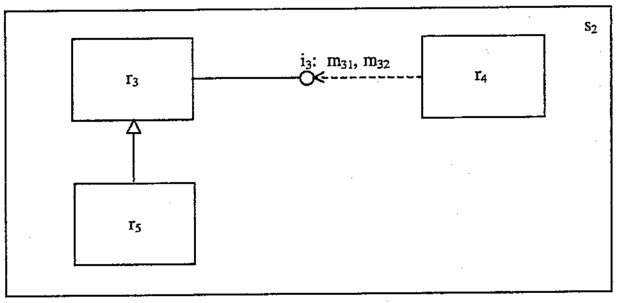


fig. 6: interface suite s_2

$Proj_2 = (S_2, R_2, I_2, M_2, PR_2, PI_2, RI_2, PM_2, SS_2, SR_2)$

With:

$$\begin{array}{ll}
S_2 & = \{s_2\} \\
R_2 & = \{r_3, r_4, r_5\} \\
I_2 & = \{i_3\} \\
M_2 & = \{m_{31}, m_{32}\} \\
PR_2 & = \{s_2 \rightsquigarrow r_3, s_2 \rightsquigarrow r_4, s_2 \rightsquigarrow r_5\} \\
PI_2 & = \{r_3 \rightsquigarrow i_3\} \\
RI_2 & = \{r_4 \rightsquigarrow i_3\} \\
PM_2 & = \{i_3 \rightsquigarrow m_{31}, i_3 \rightsquigarrow m_{32}\} \\
SS_2 & = \varnothing \\
SR_2 & = \{r_5 \rightsquigarrow r_3\}
\end{array}$$

All project restrictions are met for $Proj_2$ also:
- All suite-, role-, interface- and methodnames are disjoint.
- Relations $PR_2, PI_2, RI_2, PM_2, SS_2$ and SR_2 satisfy the domain and range properties.

Both SS_2 and SS_1 are irreflexive and transitive.
The required and provided interfaces are of the same suite.

The *obligs* and *rights* functions of $Proj_2$:

$$\begin{array}{ll}
 & (obligs.Proj_2).r_3 \\
= & \quad \{\text{definition of } obligs\} \\
 & \{i \mid r_3\,(PI_2)\,i\} \cup \bigcup\{\{j \mid r\,(PI_2)\,j\} \mid r_3\,(SR_2)\,r \mid r\} \\
= & \quad \{\neg\exists(r :: r_3\,(SR_2)\,r)\} \\
 & \{i \mid r_3\,(PI_2)\,i\} \\
= & \quad \{\{r_3 \rightsquigarrow i_3\} \subseteq PI_2\} \\
 & \{i_3\}
\end{array}$$

$(obligs.Proj_2).r_4$

$=$ $\{$definition of *obligs*$\}$

$\{i \mid r_4 \,(PI_2)\, i\} \cup \bigcup\{\{j \mid r \,(PI_2)\, j\} \mid r_4 \,(SR_2)\, r \mid r\}$

$=$ $\{\neg\exists(r \mathrel{::} r_4 \,(SR_2)\, r\}$

$\{i \mid r_4 \,(PI_2)\, i\}$

$=$ $\{\neg\exists(i \mathrel{::} r_4 \,(PI_2)\, i\}$

$\varnothing$

$(obligs.Proj_2).r_5$

$=$ $\{$definition of *obligs*$\}$

$\{i \mid r_5 \,(PI_2)\, i\} \cup \bigcup\{\{j \mid r \,(PI_2)\, j\} \mid r_5 \,(SR_2)\, r \mid r\}$

$=$ $\{r_5 \,(SR_2)\, r_3\}$

$\{i \mid r_5 \,(PI_2)\, i\} \cup \{j \mid r_3 \,(PI_2)\, j\}$

$=$ $\{\{r_3 \rightsquigarrow i_3\} \subseteq PI_2\}$

$\{i_3\}$

$(rights.Proj_2).r_3$

$=$ $\{$definition of *rights*$\}$

$\{i \mid r_3 \,(RI_2)\, i\} \cup \bigcup\{\{j \mid r \,(RI_2)\, j\} \mid r_3 \,(SR_2)\, r \mid r\}$

$=$ $\{\neg\exists(r \mathrel{::} r_3 \,(SR_2)\, r)\}$

$\{i \mid r_3 \,(RI_2)\, i\}$

$=$ $\{\neg\exists(i \mathrel{::} r_3 \,(RI_2)\, i)\}$

$\varnothing$

$(rights.Proj_2).r_4$

$=$ $\{$definition of *rights*$\}$

$\{i \mid r_4 \,(RI_2)\, i\} \cup \bigcup\{\{j \mid r \,(RI_2)\, j\} \mid r_4 \,(SR_2)\, r \mid r\}$

$=$ $\{\neg\exists(r \mathrel{::} r_4 \,(SR_2)\, r)\}$

$\{i \mid r_4 \,(RI_2)\, i\}$

$=$ $\{\{r_4 \rightsquigarrow i_3\} \subseteq RI_2\}$

$\{i_3\}$

$(rights.Proj_2).r_5$

$=$ $\{$definition of *rights*$\}$

$\{i \mid r_5 \,(RI_2)\, i\} \cup \bigcup\{\{j \mid r \,(RI_2)\, j\} \mid r_5 \,(SR_2)\, r \mid r\}$

$=$ $\{r_5 \,(SR_2)\, r_3\}$

$\{i \mid r_5 \,(RI_2)\, i\} \cup \{j \mid r_3 \,(RI_2)\, j\}$

$=$ $\{\neg\exists(i \mathrel{::} r_5 \,(RI_2)\, i \vee r_3 \,(RI_2)\, i)\}$

$\varnothing$

2.4.3 Multiple suites, with inheritance between suites (specialisation) and roles

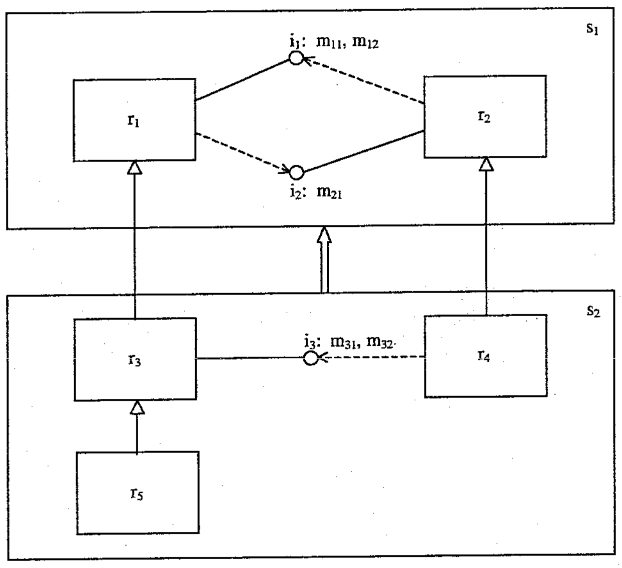


fig. 7: s_2 inherits from s_1

$Proj_3 = (S_3, R_3, I_3, M_3, PR_3, PI_3, RI_3, PM_3, SS_3, SR_3)$

With:

$$\begin{array}{lll}
S_3 & = \{s_1, s_2\} & = S_1 \cup S_2 \\
R_3 & = \{r_1, r_2, r_3, r_4, r_5\} & = R_1 \cup R_2 \\
I_3 & = \{i_1, i_2, i_3\} & = I_1 \cup I_2 \\
M_3 & = \{m_{11}, m_{12}, m_{21}, m_{31}, m_{32}\} & = M_1 \cup M_2 \\
PR_3 & = \{s_1 \frown r_1, s_1 \frown r_2, s_2 \frown r_3, s_2 \frown r_4, s_2 \frown r_5\} & = PR_1 \cup PR_2 \\
PI_3 & = \{r_1 \frown i_1, r_2 \frown i_2, r_3 \frown i_3\} & = PI_1 \cup PI_2 \\
RI_3 & = \{r_1 \frown i_2, r_2 \frown i_1, r_4 \frown i_3\} & = RI_1 \cup RI_2 \\
PM_3 & = \{i_1 \frown m_{11}, i_1 \frown m_{12}, i_2 \frown m_{21}, i_3 \frown m_{31}, i_3 \frown m_{32}\} & = PM_1 \cup PM_2 \\
SS_3 & = \{s_2 \frown s_1\} & \\
SR_3{}^{\bullet} & = \{r_3 \frown r_1, r_4 \frown r_2, r_5 \frown r_3\} &
\end{array}$$

The 10-tuple $Proj_3$ is *not* a project. All restrictions are met, except one, transitivity of relation $SR_3{}^{\bullet}$. We have $r_5 \,(SR_3)\, r_3 \wedge r_3 \,(SR_3)\, r_1$ which should imply that we have $r_5 \,(SR_3)\, r_1$ as well. If we look at figure 7 we can see that there is indeed no arrow from role r_5 to role r_1, but we can also see that r_5 *does* inherit from r_1, because r_1 is the grandfather of r_5.
If we change $SR_3{}^{\bullet}$ into $SR_3 = \{r_3 \frown r_1, r_4 \frown r_2, r_5 \frown r_3, r_5 \frown r_1\}$, all restrictions are met and $Proj_3$ is a well defined project.

The *obligs* and *rights* functions of $Proj_3$:

$$\begin{aligned}
& (\mathit{obligs}.\mathit{Proj}_3).r_1 \\
= & \quad \{\text{definition of } \mathit{obligs}\} \\
& \{i \mid r_1 \langle PI_3 \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI_3 \rangle\, j\} \mid r_1 \langle SR_3 \rangle\, r \mid r\} \\
= & \quad \{\neg\exists(r :: r_1 \langle SR_3 \rangle\, r)\} \\
& \{i \mid r_1 \langle PI_3 \rangle\, i\} \\
= & \quad \{\{r_1 \leadsto i_1\} \subseteq PI_3\} \\
& \{i_1\}
\end{aligned}$$

$$\begin{aligned}
& (\mathit{obligs}.\mathit{Proj}_3).r_2 \\
= & \quad \{\text{definition of } \mathit{obligs}\} \\
& \{i \mid r_2 \langle PI_3 \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI_3 \rangle\, j\} \mid r_2 \langle SR_3 \rangle\, r \mid r\} \\
= & \quad \{\neg\exists(r :: r_2 \langle SR_3 \rangle\, r)\} \\
& \{i \mid r_2 \langle PI_3 \rangle\, i\} \\
= & \quad \{\{r_2 \leadsto i_2\} \subseteq PI_3\} \\
& \{i_2\}
\end{aligned}$$

$$\begin{aligned}
& (\mathit{obligs}.\mathit{Proj}_3).r_3 \\
= & \quad \{\text{definition of } \mathit{obligs}\} \\
& \{i \mid r_3 \langle PI_3 \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI_3 \rangle\, j\} \mid r_3 \langle SR_3 \rangle\, r \mid r\} \\
= & \quad \{r_3 \langle SR_3 \rangle\, r_1\} \\
& \{i \mid r_3 \langle PI_3 \rangle\, i\} \cup \{j \mid r_1 \langle PI_3 \rangle\, j\} \\
= & \quad \{\{r_3 \leadsto i_3\} \subseteq PI_3 \,;\, \{r_1 \leadsto i_1\} \subseteq PI_3\} \\
& \{i_3, i_1\}
\end{aligned}$$

$$\begin{aligned}
& (\mathit{obligs}.\mathit{Proj}_3).r_4 \\
= & \quad \{\text{definition of } \mathit{obligs}\} \\
& \{i \mid r_4 \langle PI_3 \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI_3 \rangle\, j\} \mid r_4 \langle SR_3 \rangle\, r \mid r\} \\
= & \quad \{r_4 \langle SR_3 \rangle\, r_2\} \\
& \{i \mid r_4 \langle PI_3 \rangle\, i\} \cup \{j \mid r_2 \langle PI_3 \rangle\, j\} \\
= & \quad \{\{r_2 \leadsto i_2\} \subseteq PI_3\} \\
& \varnothing \cup \{i_2\} = \{i_2\}
\end{aligned}$$

$$\begin{aligned}
& (\mathit{obligs}.\mathit{Proj}_3).r_5 \\
= & \quad \{\text{definition of } \mathit{obligs}\} \\
& \{i \mid r_5 \langle PI_3 \rangle\, i\} \cup \bigcup\{\{j \mid r \langle PI_3 \rangle\, j\} \mid r_5 \langle SR_3 \rangle\, r \mid r\} \\
= & \quad \{r_5 \langle SR_3 \rangle\, r_3 \wedge r_5 \langle SR_3 \rangle\, r_1\} \\
& \{i \mid r_5 \langle PI_3 \rangle\, i\} \cup \{j \mid r_3 \langle PI_3 \rangle\, j\} \cup \{k \mid r_1 \langle PI_3 \rangle\, k\} \\
= & \quad \{\{r_3 \leadsto i_3\} \subseteq PI_3 \,;\, \{r_1 \leadsto i_1\} \subseteq PI_3\} \\
& \varnothing \cup \{i_3\} \cup \{i_1\} = \{i_3, i_1\}
\end{aligned}$$

As we can see role r_5 should indeed have interface i_1 in it's obligation-set, because (grandfather) role r_1 provides this interface.

$$\begin{array}{ll}
 & (\mathit{rights}.\mathit{Proj}_3).r_1 \\
= & \quad \{\text{definition of } \mathit{rights}\} \\
 & \{i \mid r_1 \,(RI_3)\, i\} \cup \bigcup\{\{j \mid r \,(RI_3)\, j\} \mid r_1 \,(SR_3)\, r \mid r\} \\
= & \quad \{\neg\exists(r \;\|\; r_1 \,(SR_3)\, r)\} \\
 & \{i \mid r_1 \,(RI_3)\, i\} \\
= & \quad \{\{r_1 \mapsto i_2\} \subseteq RI_3\} \\
 & \{i_2\}
\end{array}$$

$$\begin{array}{ll}
 & (\mathit{rights}.\mathit{Proj}_3).r_2 \\
= & \quad \{\text{definition of } \mathit{rights}\} \\
 & \{i \mid r_2 \,(RI_3)\, i\} \cup \bigcup\{\{j \mid r \,(RI_3)\, j\} \mid r_2 \,(SR_3)\, r \mid r\} \\
= & \quad \{\neg\exists(r \;\|\; r_2 \,(SR_3)\, r)\} \\
 & \{i \mid r_2 \,(RI_3)\, i\} \\
= & \quad \{\{r_2 \mapsto i_1\} \subseteq RI_3\} \\
 & \{i_1\}
\end{array}$$

$$\begin{array}{ll}
 & (\mathit{rights}.\mathit{Proj}_3).r_3 \\
= & \quad \{\text{definition of } \mathit{rights}\} \\
 & \{i \mid r_3 \,(RI_3)\, i\} \cup \bigcup\{\{j \mid r \,(RI_3)\, j\} \mid r_3 \,(SR_3)\, r \mid r\} \\
= & \quad \{r_3 \,(SR_3)\, r_1\} \\
 & \{i \mid r_3 \,(RI_3)\, i\} \cup \{j \mid r_1 \,(RI_3)\, j\} \\
= & \quad \{\{r_1 \mapsto i_2\} \subseteq RI_3\} \\
 & \varnothing \cup \{i_2\} = \{i_2\}
\end{array}$$

$$\begin{array}{ll}
 & (\mathit{rights}.\mathit{Proj}_3).r_4 \\
= & \quad \{\text{definition of } \mathit{rights}\} \\
 & \{i \mid r_4 \,(RI_3)\, i\} \cup \bigcup\{\{j \mid r \,(RI_3)\, j\} \mid r_4 \,(SR_3)\, r \mid r\} \\
= & \quad \{r_4 \,(SR_3)\, r_2\} \\
 & \{i \mid r_4 \,(RI_3)\, i\} \cup \{j \mid r_2 \,(RI_3)\, j\} \\
= & \quad \{\{r_4 \mapsto i_3\} \subseteq RI_3\,;\, \{r_2 \mapsto i_1\} \subseteq RI_3\} \\
 & \{i_3\} \cup \{i_1\} = \{i_1, i_3\}
\end{array}$$

$$\begin{array}{ll}
 & (\mathit{rights}.\mathit{Proj}_3).r_5 \\
= & \quad \{\text{definition of } \mathit{rights}\} \\
 & \{i \mid r_5 \,(RI_3)\, i\} \cup \bigcup\{\{j \mid r \,(RI_3)\, j\} \mid r_5 \,(SR_3)\, r \mid r\} \\
= & \quad \{r_5 \,(SR_3)\, r_3 \wedge r_5 \,(SR_3)\, r_1\} \\
 & \{i \mid r_5 \,(RI_3)\, i\} \cup \{j \mid r_3 \,(RI_3)\, j\} \cup \{k \mid r_1 \,(RI_3)\, k\} \\
= & \quad \{\{r_1 \mapsto i_2\} \subseteq RI_3\} \\
 & \varnothing \cup \varnothing \cup \{i_2\} = \{i_2\}
\end{array}$$

2.4.4 An erroneous example

So far we have only seen interface role diagrams that are well defined. We have discussed a number of restrictions that have to be satisfied on creation of an interface role diagram. Now we will discuss an example where we have deliberately made a number of mistakes to show that our restrictions can be violated. We will show what these mistakes are and explain why they will lead to a faulty interface role diagram.

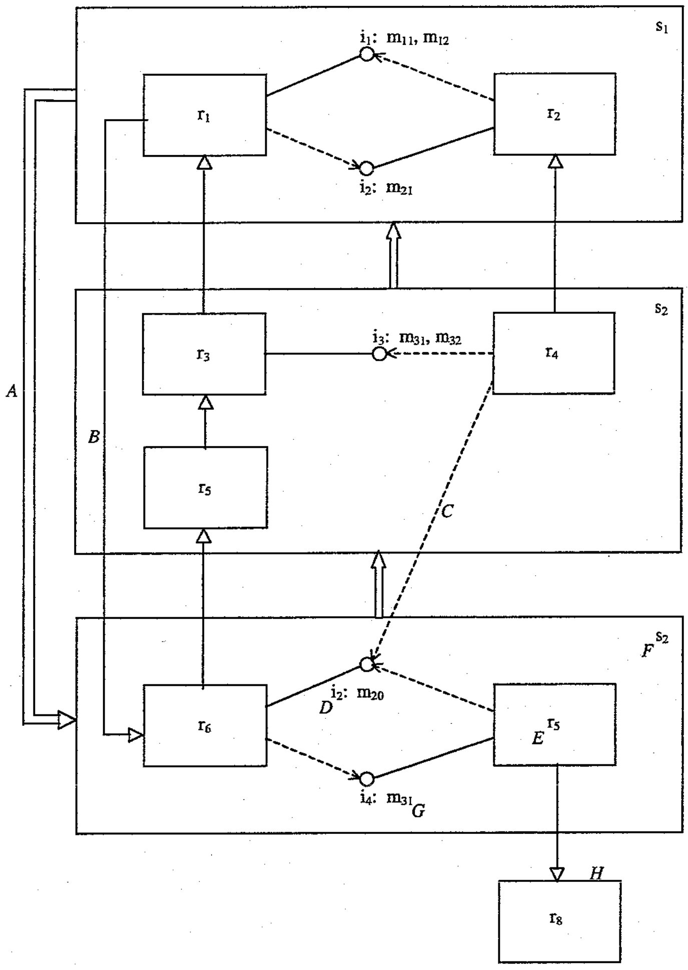


fig. 8: An interface role diagram containing some errors.

There are eight errors in the picture above, A-H.

A: The suite inheritance relation is cyclic. Since we demand our inheritance relations to be both transitive and irreflexive, an interface role diagram containing such a cycle will not be a correctly defined project.
B: The role inheritance relation is cyclic and therefore cannot be both transitive

and irreflexive.
C: Role r_4 requires an interface that is provided by a role of a different suite. Note that both the suite and the interface have names that are not unique.
D: There are two interfaces called i_2. All interfaces should have disjoint names.
E: Two roles r_5. Rolenames should be disjoint.
F: Two suites s_2. Suitenames should be disjoint.
G: Two methods m_{31}. Methodnames should be disjoint.
H: Role r_8 does not belong to a suite. It should belong to exactly one suite.

We will now represent all information we get from figure 8 formally in terms of sets and relations and verify our project restrictions.

$Proj_{err} = (S_e, R_e, I_e, M_e, PR_e, PI_e, RI_e, PM_e, SS_e, SR_e)$

With:

$$
\begin{aligned}
S_e &= \{s_1, s_2, s_2\} = \{s_1, s_2\} \\
R_e &= \{r_1, r_2, r_3, r_4, r_5, r_6, r_8\} \\
I_e &= \{i_1, i_2, i_3, i_4\} \\
M_e &= \{m_{11}, m_{12}, m_{20}, m_{21}, m_{31}, m_{32}\} \\
PR_e &= \{s_1 \backsim r_1, s_1 \backsim r_2, s_2 \backsim r_3, s_2 \backsim r_4, s_2 \backsim r_5, s_2 \backsim r_6\} \\
PI_e &= \{r_1 \backsim i_1, r_2 \backsim i_2, r_3 \backsim i_3, r_6 \backsim i_2, r_5 \backsim i_4\} \\
RI_e &= \{r_1 \backsim i_2, r_2 \backsim i_1, r_4 \backsim i_3, r_4 \backsim i_2, r_6 \backsim i_4, r_5 \backsim i_2\} \\
PM_e &= \{i_1 \backsim m_{11}, i_1 \backsim m_{12}, i_2 \backsim m_{21}, i_3 \backsim m_{31}, i_3 \backsim m_{32}, i_2 \backsim m_{20}, i_4 \backsim m_{31}\} \\
SS_e{}^{\bullet} &= \{s_2 \backsim s_1, s_2 \backsim s_2, s_1 \backsim s_2\} \\
SS_e &= \{s_2 \backsim s_1, s_2 \backsim s_2, s_1 \backsim s_2, s_1 \backsim s_1\} \\
SR_e{}^{\bullet} &= \{r_3 \backsim r_1, r_4 \backsim r_2, r_5 \backsim r_3, r_6 \backsim r_5, r_1 \backsim r_6, r_5 \backsim r_8\} \\
SR_e &= \{r_3 \backsim r_1, r_4 \backsim r_2, r_5 \backsim r_3, r_6 \backsim r_5, r_1 \backsim r_6, r_5 \backsim r_8, \\
&\quad r_5 \backsim r_1, r_6 \backsim r_1, r_1 \backsim r_1, r_6 \backsim r_3, r_1 \backsim r_3, r_3 \backsim r_3, r_1 \backsim r_5, r_3 \backsim r_5, r_5 \backsim r_5, r_3 \backsim r_6, r_5 \backsim r_6, r_6 \backsim r_6\}
\end{aligned}
$$

$^{\bullet}$Inheritance relations without the inclusion of transitivity.

We will now verify our project restrictions, $Proj_{err} \in \mathbb{PROJ}$?

Disjoint names:
As we can see we still have sets of (automatically disjoint) Suite-, Role-, Interface- and Methodnames. Our set of suites however consists of only two elements, where we should have three, since we have named two suites the same. This means we should be very careful with the disjointness of the names, since a union of two identical names will not automatically lead to a violation of a restriction.

Domain and range restrictions:
$PR_e \in S_e \longleftarrow\!\!\mid R_e$: The domain restriction of the relation PR_e is not met. Role r_8 is an element of the set of roles R_e, but is not related to a suite by the Provides Role relation PR_e. Error *H*.
$PI_e \in R_e \longleftarrow\!\!\mid I_e$: This restriction is not met either. Interface i_2 is related to both r_2 and r_6. Each interface has to be provided by exactly one role. We can see that this error is a result of the fact that two interfaces are called i_2, error *D*.

$RI_e \in R_e \succ\!\!\!-\!\!\!\dashv I_e$: This restriction is met. Each interface is required by at least one role.
$PM_e \in I_e \leftarrow\!\!\!\dashv M_e$: Method m_{31} is provided by both i_3 and i_4. Error G indirectly falsifies this restriction.
$SS_e \in S_e \succ\!\!\!-\!\!\!\prec S_e$: Both the domain and the range of relation SS_e consist of suites, i.e. elements of S_e. Therefore this restriction is met.
$SR_e \in R_e \succ\!\!\!-\!\!\!\dashv R_e$: Both the domain and the range of relation SR_e consist of roles. The restriction is met.

Transitivity and irreflexivity restrictions:
$\forall(x, y, z \mid y \,(SS_e)\, x \wedge x \,(SS_e)\, z \mid y \,(SS_e)\, z)$: The Suite inheritance relation SS_e is transitive. If we just use our suite inheritance arrows from the picture to form our suite inheritance relation, we would have the relation ${SS_e}^{\bullet}$. The relation ${SS_e}^{\bullet}$ is *not* transitive. We've seen this occur in the last section as well. For now, we'll just add transitivity to this relation by hand, we will look into this when we discuss the extension of a project in chapter 2.7.
$\forall(x, y, z \mid y \,(SR_e)\, x \wedge x \,(SR_e)\, z \mid y \,(SR_e)\, z)$: Role inheritance encounters exactly the same problem as suite inheritance. When we take all role inheritance arrows we acquire relation ${SR_e}^{\bullet}$, which is *not* transitive. We constructed the transitive relation SR_e on behalf of testing irreflexivity.
$\forall(x \parallel \neg(x \,(SS_e)\, x))$: The suite inheritance relation SS_e is *not* irreflexive. Our suite inheritance is cyclic, error A, as can be seen easily in the picture. When we add our transitivity to this, we'll have each suite within the cycle also inheriting from itself and therefore being reflexive. In fact, the relation ${SS_e}^{\bullet}$ (where transitivity was not incorporated) is not irreflexive either. The reason for this is simply because we have two suites named s_2 inheriting from each other.
$\forall(x \parallel \neg(x \,(SR_e)\, x))$: Role inheritance is not irreflexive either. Again we have a cycle (error B) and all four roles in the cycle inherit from all their ancestors, which includes themselves. We can see that when we omit the inherent transitivity of the inheritance, relation ${SR_e}^{\bullet}$, we will not detect reflexivity. This is quite obvious, because we don't see any inheritance arrows pointing from a role to itself.

The two remaining restrictions:
$\forall(r, r' \mid r' \,(SR_e)\, r \mid PR_e.r' = PR_e.r \vee PR_e.r' \,(SS_e)\, PR_e.r)$: All except one of our inheritance relations between roles are either within the same suite, or between suites that inherit from each other by suite inheritance and in the same direction. The same direction means that when suite s_1 inherits from suite s_2, a role of s_1 may inherit from a role of suite s_2, but not the other way round.
The one inheritance relation that falsifies this restriction is the one involving role r_8, since $PR_e.r_8$ does not exist. $\forall(i, r \mid r \,(RI_e)\, i \mid PR_e.r = PR_e.(PI_e.i))$: Error C should directly violate this restriction, because role r_4 requires an interface that is provided by a role from a different suite. However, this suite was not named properly and was also called s_2. Furthermore the interface that role r_4 wrongly requires, interface i_2, was also named incorrectly, since it also occurs in suite s_1. This will lead to a violation of this restriction, since $PI_e.i_2 = \{r_6, r_2\}$ and $\{r_6, r_2\}$ is not an element of the domain of PR_e.

What we can conclude from this erroneous example is that we especially have to be very careful with the names or our suites, roles, interfaces and methods. Furthermore the transitivity of our inheritance relations does not explicitly follow from our interface role diagram.

2.5 Constructing a well defined project

When we create interface role diagrams, we frequently make use of one or more existing interface suites to inherit from. New roles are created to possibly inherit from old roles. Additional interfaces with new methods are created to provide new functionality. In the previous sections we concluded that we encountered problems trying to formalise interface role diagrams consisting of multiple suites. We want to construct our projects through composition and extension of single suites, as the following figure shows:

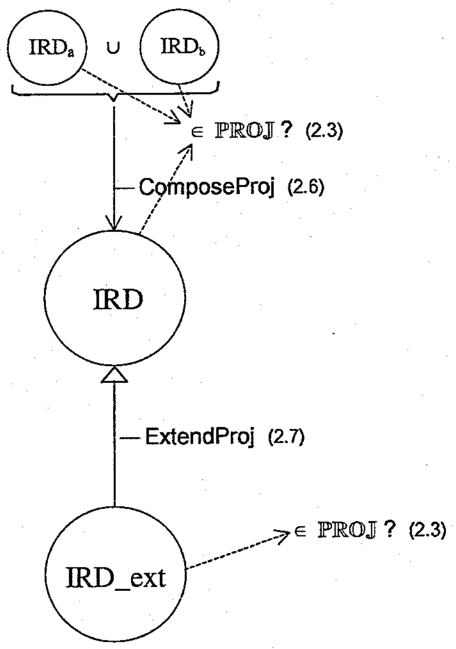


fig. 9: Constructing larger projects.

2.6 Composing projects

In this section we are going to describe the composition of two independent interface role diagrams. Composition of two interface role diagrams, i.e. projects, means merging them together with as result having one larger interface role diagram containing both of them. By independent we mean that there are no relations between the two projects, which in our case means no inheritance relations. The reason for making this composition will become more clear when we do allow using inheritance relations in the next section, where we will discuss the extension of a project.

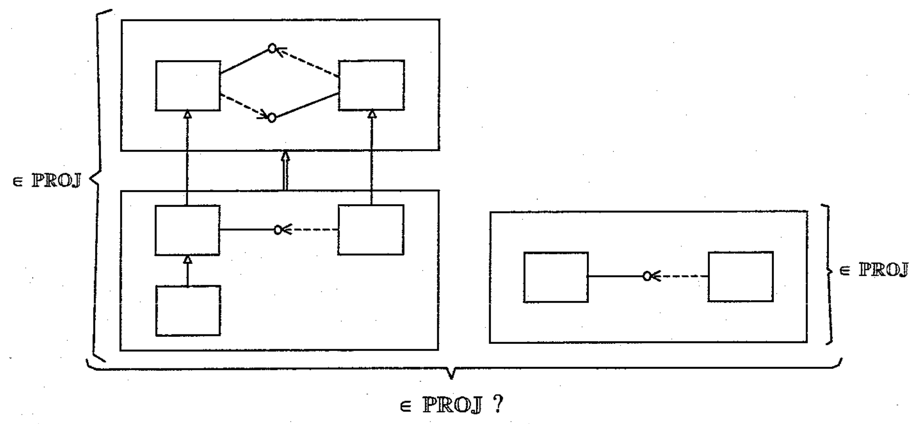


fig. 10: Composition of two projects.

Composition of two projects seems to boil down to a simple union of the two, which is actually almost the case. We do not introduce new inheritance relations, neither do we introduce new roles, interfaces or methods. We only join existing and well defined sets of Suites, Roles, Interfaces and Methods as well as the relations between them together. Well defined means that we have two projects ($\in \mathbb{PROJ}$) where all our previously stated restrictions are met.
A simple union of the two without adding anything new will automatically mean that all restrictions are still met, or not? This is almost true, except for the disjointness restriction of Suitenames, Rolenames, Interfacenames and Methodnames.
We will introduce a function *ComposeProj* to formally state these restrictions as well as the result of composing two projects, which indeed will be "simple" union.

$ComposeProj \in \mathbb{PROJ} \leftarrow\!\!\prec \mathbb{PROJ} \times \mathbb{PROJ}$

$((S_1, R_1, ..., SR_1), (S_2, R_2, ..., SR_2)) \in ComposeProj\rangle$

$\equiv$

$$\begin{array}{l} S_1 \cap S_2 = \varnothing \\ R_1 \cap R_2 = \varnothing \\ I_1 \cap I_2 = \varnothing \\ M_1 \cap M_2 = \varnothing \end{array}$$

$(S, R, ..., SR) = ComposeProj.((S_1, R_1, ..., SR_1), (S_2, R_2, ..., SR_2))$

$\equiv$

$$\begin{array}{lll} S & = S_1 & \cup\, S_2 \\ R & = R_1 & \cup\, R_2 \\ I & = I_1 & \cup\, I_2 \\ M & = M_1 & \cup\, M_2 \\ PR & = PR_1 & \cup\, PR_2 \\ PI & = PI_1 & \cup\, PI_2 \\ RI & = RI_1 & \cup\, RI_2 \\ PM & = PM_1 & \cup\, PM_2 \\ SS & = SS_1 & \cup\, SS_2 \\ SR & = SR_1 & \cup\, SR_2 \end{array}$$

2.7 Extending a project

When we create interface role diagrams, we frequently make use of one or more existing interface suites to inherit from. These interface suites can be successfully described individually as projects. In the previous section we discussed the composition of two projects to form a correctly defined project consisting of both of them. In this section we will discuss what restrictions are to be met when we are making an extension of a project (possibly a composition of several ones), to again construct a correctly defined project.

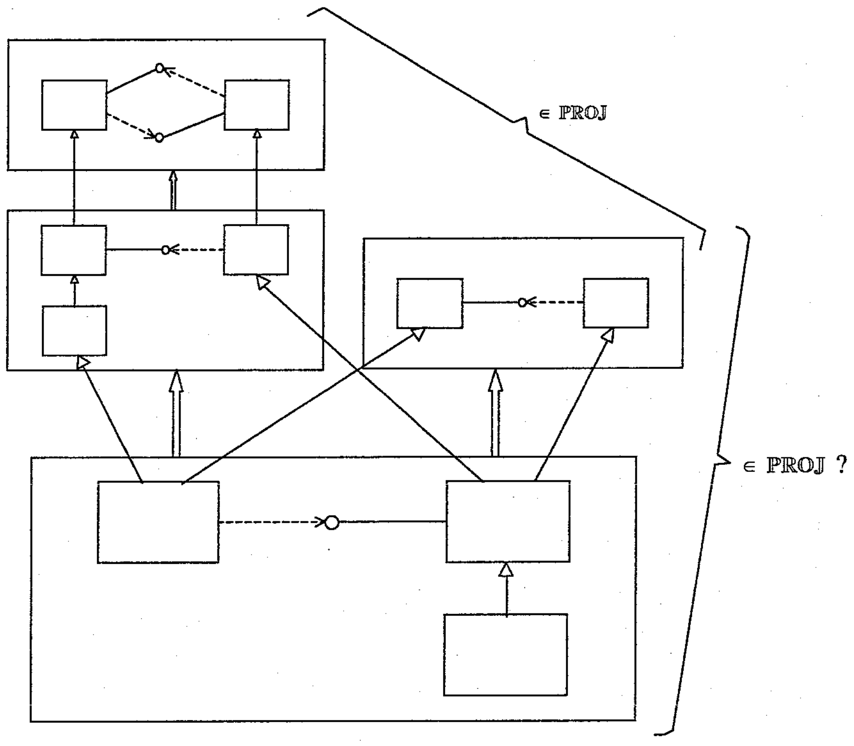


fig. 11: Extending a project by a single suite.

We will now introduce a function *ExtendProj* that constructs a new project (correctly defined interface role diagram) on basis of an old project and a newly added suite. This added suite is also a 10-tuple of sets and relations, but we don't know if any of our restrictions are met yet (especially the ones concerning inheritance).

$$\begin{aligned}\mathbb{SUITE} = {} & \wp(\mathbb{S})\times\wp(\mathbb{R})\times\wp(\mathbb{I})\times\wp(\mathbb{M})\times \\ & (\mathbb{S}\rightarrowtail\!\!\prec\mathbb{R})\times(\mathbb{R}\rightarrowtail\!\!\prec\mathbb{I})\times(\mathbb{R}\rightarrowtail\!\!\prec\mathbb{I})\times(\mathbb{I}\rightarrowtail\!\!\prec\mathbb{M})\times(\mathbb{S}\rightarrowtail\!\!\prec\mathbb{S})\times(\mathbb{R}\rightarrowtail\!\!\prec\mathbb{R})\end{aligned}$$

$$\mathit{ExtendProj} \in \mathbb{PROJ} \longleftarrow\!\!\prec \mathbb{PROJ}\times\mathbb{SUITE}$$

$$((S_{old}, R_{old}, ..., SR_{old}), (S_{new}, R_{new}, ..., SR_{new})) \in \mathit{ExtendProj}\succ$$

$\equiv$

$$\begin{aligned}&\#S_{new} = 1\\ &S_{new} \cap S_{old} = \varnothing\\ &R_{new} \cap R_{old} = \varnothing\\ &I_{new} \cap I_{old} = \varnothing\\ &M_{new} \cap M_{old} = \varnothing\\ &PR_{new} \in S_{new} \longleftarrow\!\!\dashv R_{new}\\ &PI_{new} \in R_{new} \longleftarrow\!\!\dashv I_{new}\\ &RI_{new} \in R_{new} \rightarrowtail\!\!\dashv I_{new}\\ &PM_{new} \in I_{new} \longleftarrow\!\!\dashv M_{new}\\ &SS_{new} \in S_{new} \rightarrowtail\!\!\prec S_{old}\\ &SR_{new} \in R_{new} \rightarrowtail\!\!\prec (R_{old} \cup R_{new})\\ &\forall(r \mathrel{\vdots\vdots} \neg(r \,\{SR_{new}{}^{+}\}\, r))\\ &\forall(r, r' \mathrel{\vdots} r \in R_{old} \wedge r' \,\{SR_{new}\}\, r \mathrel{\vdots} PR_{new}.r' \,\{SS_{new}\}\, PR_{old}.r)\end{aligned}$$

$$(S, R, ..., SR) = \mathit{ExtendProj}.((S_{old}, R_{old}, ..., SR_{old}), (S_{new}, R_{new}, ..., SR_{new}))$$

$\equiv$

$$\begin{aligned}S &= S_{old} \cup S_{new}\\ R &= R_{old} \cup R_{new}\\ I &= I_{old} \cup I_{new}\\ M &= M_{old} \cup M_{new}\\ PR &= PR_{old} \cup PR_{new}\\ PI &= PI_{old} \cup PI_{new}\\ RI &= RI_{old} \cup RI_{new}\\ PM &= PM_{old} \cup PM_{new}\\ SS &= SS_{old} \cup SS_{new} \cup \{s''\backsim s \mid \exists(s' \mathrel{\vdots} s'' \,\{SS_{new}\}\, s' \wedge s' \,\{SS_{old}\}\, s)\}\\ SR &= SR_{old} \cup SR_{new} \cup \{r''\backsim r \mid \exists(r' \mathrel{\vdots} r'' \,\{SR_{new}{}^{+}\}\, r' \wedge (r' \,\{SR_{old}\}\, r \vee r' \,\{SR_{new}\}\, r))\}\end{aligned}$$

Explanation:

domain restrictions:
- Only one new suite is added (at a time).
- All suite-, role-, interface- and methodnames have to be disjoint.
- The 4 Provides and Requires relations have to meet their domain and range properties.
- New suite inheritance is possible between the new suite and one or several old suites.
- New role inheritance is possible between a new role and an old role or between new roles.
- New role inheritance will have to be a-cyclic. The $^{+}$ in $SR_{new}{}^{+}$ denotes the transitive closure of the role inheritance relation.

- Role inheritance between a new role and an old role has to imply that there is also a suite relation between those two roles.

result of *ExtendProj* function:
When all domain restrictions are met, extending a project almost boils down to a simple union of the old project and the new suite. Only the inheritance relations need some additional attention to apply to our transitivity demands. A new suite inheriting from an old suite means the new suite also inherits from all the ancestors of the old suite. The same goes for roles, although roles can also inherit from roles of the same suite. This means we have to make inheritance between new roles transitive as well as inheriting from the ancestors of possible old roles.

2.8 Behaviour

Now we want to specify possible behaviour of a set of IR-diagrams (i.e. a project). In the next chapter we are going to discuss sequence diagrams to specify the exact behaviour of our projects, or that part of behaviour we find relevant. In this section we will discuss what all possible behaviour of a project consists of only having the information of our interface role diagram.
All possible sequences of method-calls and -returns are now possible, as long as these methods are called on the right interface by the right role. We are only discussing single-threaded behaviour here. The behaviour can be seen as a tree-structure, where the first method-call is made from the root. This root can be seen as the user of the system and may call each single method of an interface of a role. The next action can then be either of the folowing:
- The role makes an internal action. We will neglect this, since we are only concerned in interface-calls and -returns.
- The method-call is returned.
- Another method (of an interface of a role) is called.
When a method-call is returned, the role that called the method has to take the next action. When a method is called on an interface of a(nother) role, that role has to take the next action.

$\mathbb{C} = \mathbb{T}\times\mathbb{R}\times\mathbb{I}\times\mathbb{M}$
$\mathbb{T} = seq.\mathbb{C}$

$\mathbb{B} = \wp(\mathbb{C})$

$mainBehaviour \in \mathbb{B} \longleftarrow\!\!\dashv \mathbb{PROJ}$
$(t, r, i, m) = mainBehaviour.proj$
$\equiv \quad \{proj = (S, R, I, M, PR, PI, RI, PM, SS, SR)\}$
$r \in R$
$i \in I$
$m \in M$
$i \in (obligs.proj).r$
$i \; (PM) \; m$
$\forall(c \mid c \; in \; t \mid c \in subBehaviour.(r, proj))$

$subBehaviour \in \mathbb{B} \longleftarrow\!\!\prec \mathbb{R}\times\mathbb{PROJ}$
$(cr, (S, R, I, M, PR, PI, RI, PM, SS, SR)) \in subBehaviour{>} \equiv cr \in R$
$(t, rr, i, m) = subBehaviour.(cr, (S, R, I, M, PR, PI, RI, PM, SS, SR))$
$\equiv \quad \{(S, R, I, M, PR, PI, RI, PM, SS, SR) = proj\}$
$rr \in R$
$i \in I$
$m \in M$
$i \in (rights.proj).cr$
$i \in (obligs.proj).rr$
$i \; (PM) \; m$
$\forall(c \mid c \; in \; t \mid c \in subBehaviour.(rr, proj))$

Chapter 3

Sequence Diagrams

3.1 Introduction

Sequence diagrams are a means to describe behaviour. One sequence diagram, like the name says, describes one possible sequence of actions. The actions we consider are method-calls and -returns on the interfaces. One role calls a method on an interface of a(nother) role. That role may return this call, or call a method on some other interface of some role. The communication we describe with sequence diagrams is therefore the communication between roles, through methods of interfaces. We'll only discuss *single-threaded* behaviour here, meaning that there is only one point of execution at any time, not several ones in parallel. There is one flow of control within our sequences.
As mentioned before, a sequence diagram describes one possible sequence of actions. Of one particular interface role diagram that is. We can use several sequence diagrams to describe more possible sequences of actions that can occur, given a single interface role diagram. This way we could be able to give the complete behaviour of a certain project (described by the interface role diagram). In practise this is generally impossible, since most of the time there will be a huge amount of possible sequences of actions and drawing a sequence diagram for each and every one of them, would be impracticable.
Therefore we use our sequence diagrams to describe a certain, important, part of the behaviour. Especially those parts of the behaviour that are important for inheritance. Our goal is namely not only to verify correct inheritance on the static level (interface role diagrams), but also on the dynamic level (sequence diagrams).

3.2 Sequence diagram representation (Option 1)

At first we try to keep our sequence diagrams and also our formal representation of those sequence diagrams as simple as possible. We want to describe a sequence of method-calls and returns. These method-calls and -returns take place between two roles using an interfaces' method. From this information we can make the following formal representation:

$$\mathbb{SD} = seq.\mathbb{A}$$
$$\mathbb{A} = \mathbb{CR} \times \mathbb{R} \times \mathbb{R} \times \mathbb{I} \times \mathbb{M}$$
$$\mathbb{CR} = \{\mathit{call}\ ,\ \mathit{return}\}$$

A sequence diagram will be of type $\mathbb{SD}$, which is a sequence of actions ($\in \mathbb{A}$). These actions are 5-tuples: The first element of this tuple can be either *call* or *return*. The second element states the role that originates the action, the third the role that receives the action. The fourth and fifth are the action itself, namely an interface method combination.

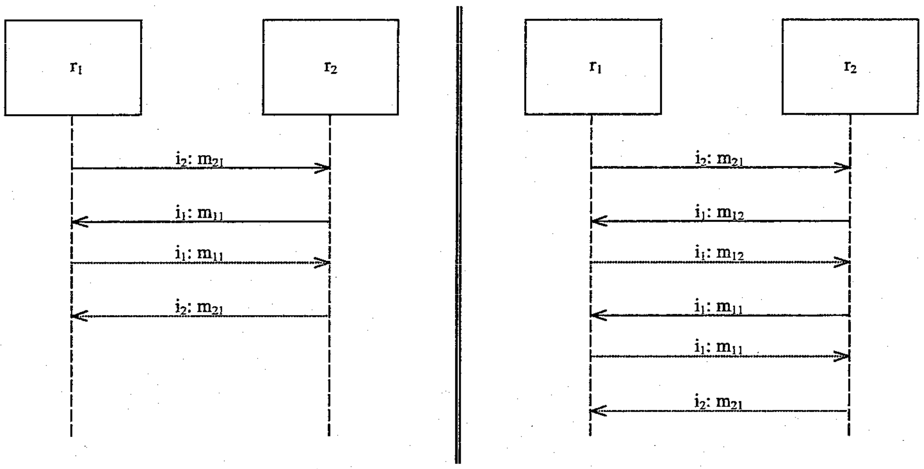


fig. 12: Two sequence diagrams describing behaviour of $Proj_1$

The picture above shows two possible sequences of actions that could belong to the interface role diagram we showed in 5, consisting of only one suite.
We can see the roles that are involved in the communication at the top of a sequence diagram, depicted by boxes with role-identifiers we also saw in our interface role diagrams. The dashed vertical lines represent time, passing from top to bottom. A method-call is depicted by a solid arrow, from the role that originates the call to the role that receives the call. A return-action is depicted by a dashed arrow towards to role that receives the return. This should be the role that originated the accompanying call-action.
Both calls and returns include their method name as well as the name of the methods' interface. We note that in our case the interface names could be omitted, since all our methods have different names and belong to exactly one interface.

The sequences of figure 12 can be formalised as follows, with SD_{1a} representing the left sequence and SD_{1b} the right.

$$SD_{1a} = [(call, r_1, r_2, i_2, m_{21}), (call, r_2, r_1, i_1, m_{11}), (return, r_1, r_2, i_1, m_{11}),\\ (return, r_2, r_1, i_2, m_{21})]$$

$$SD_{1b} = [(call, r_1, r_2, i_2, m_{21}), (call, r_2, r_1, i_1, m_{12}), (return, r_1, r_2, i_1, m_{12}),\\ (call, r_2, r_1, i_1, m_{11}), (return, r_1, r_2, i_1, m_{11}), (return, r_2, r_1, i_2, m_{21})]$$

We now give some sequences belonging to an interface role diagram containing inheritance, we discussed in 7.

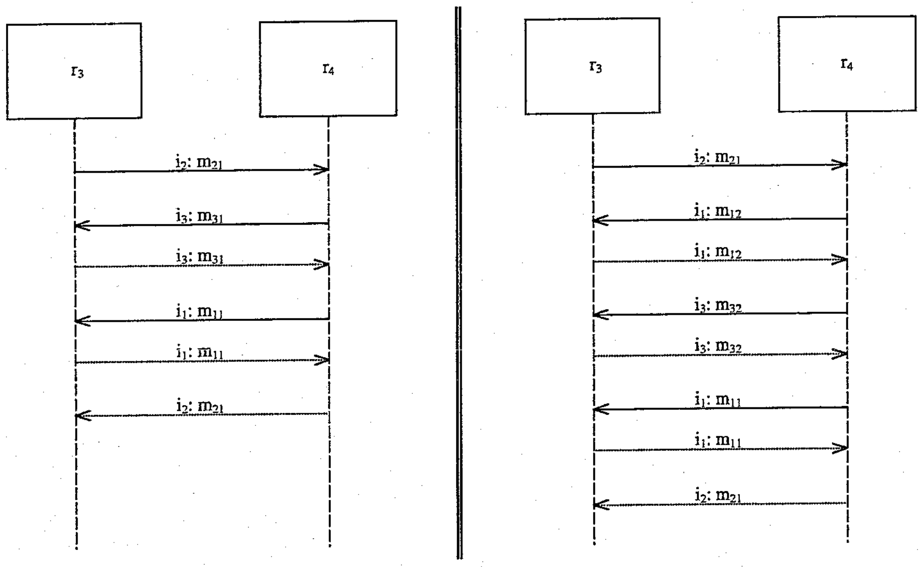


fig. 13: Sequences of *Proj*$_3$

Roles r_3 and r_4 inherit from r_1 and r_2 respectively, of the previous example. A formal representation of these sequences:

$$SD_{2a} = [(call, r_3, r_4, i_2, m_{21}), (call, r_4, r_3, i_3, m_{31}), (return, r_3, r_4, i_3, m_{31}),\\ (call, r_4, r_3, i_1, m_{11}), (return, r_3, r_4, i_1, m_{11}), (return, r_4, r_3, i_2, m_{21})]$$

$$SD_{2b} = [(call, r_3, r_4, i_2, m_{21}), (call, r_4, r_3, i_1, m_{12}), (return, r_3, r_4, i_1, m_{12}),\\ (call, r_4, r_3, i_3, m_{32}), (return, r_3, r_4, i_3, m_{32}), (call, r_4, r_3, i_1, m_{11}),\\ (return, r_3, r_4, i_1, m_{11}), (return, r_4, r_3, i_2, m_{21})]$$

Verification of the correctness of a sequence diagram has to be with respect to interface role diagram it belongs to. This means that the methods are indeed provided by the interfaces and the interfaces are provided and required by the right roles. We also have to verify that all call-actions are followed by an appropriate return and we have to make sure that our sequence diagrams describe single-threaded behaviour. We now introduce a function *checkSD* to verify a sequence diagram ($\in \mathbb{SD}$) in accordance with its project ($\in \mathbb{PROJ}$, the formal representation of its interface role diagram).

$$
\begin{array}{ll}
 & checkSD \in \mathbb{B} \longleftarrow \mathbb{SD}\times\mathbb{PROJ} \\
 & checkSD.(SD, proj) \\
= & \quad \{proj = (S, R, I, M, PR, PI, RI, PM, SS, SR)\} \\
 & \forall(r_f, r_t, i, m \mid (call, r_f, r_t, i, m) \ in \ SD \mid \\
 & \qquad\qquad i \,\{PM\}\, m \wedge i \in (obligs.proj).r_t \wedge i \in (rights.proj).r_f) \\
\wedge & \\
 & \forall(r_f, r_t, i, m \mid (return, r_f, r_t, i, m) \ in \ SD \mid \\
 & \qquad\qquad (call, r_t, r_f, i, m) \in preCalls.((return, r_f, r_t, i, m), SD)) \\
\wedge & \\
 & \forall(r_f, r_t, i, m \mid (call, r_f, r_t, i, m) \ in \ SD \mid \\
 & \qquad\qquad (return, r_t, r_f, i, m) \ in \ SD)
\end{array}
$$

There are three conjuncts in the definition of *checkSD*:

- $\forall(r_f, r_t, i, m \mid (call, r_f, r_t, i, m) \ in \ SD \mid$
  $i \,\{PM\}\, m \wedge i \in (obligs.proj).r_t \wedge i \in (rights.proj).r_f)$

A call-action of method m on interface i from role r_f to role r_t has to meet the following restrictions:
-$i \,\{PM\}\, m$: The method has to be provided by the interface.
-$i \in (obligs.proj).r_t$: The role that receives the call must have the obligation to provide the interface, either provide it himself or through one of its ancestors.
-$i \in (rights.proj).r_f$): The role that invokes the call-action must have the right to use the interface.

- $\forall(r_f, r_t, i, m \mid (return, r_f, r_t, i, m) \ in \ SD \mid$
  $(call, r_t, r_f, i, m) \in preCalls.((return, r_f, r_t, i, m), SD))$

For each return-action $(return, r_f, r_t, i, m)$ in the sequence there has to be call-action of the same method (and interface) with the roles in opposite direction. Moreover, this call-action has to be prior to the return-action. We introduce a function *preCalls* for this below.

- $\forall(r_f, r_t, i, m \mid (call, r_f, r_t, i, m) \ in \ SD \mid (return, r_f, r_t, i, m) \ in \ SD)$

Besides the fact that all returns-actions have an appropriate call-action prior to them, all calls also have to be returned correctly.

$preCalls \in \wp(\mathbb{A}) \leftarrow\!\!\!\prec \mathbb{A} \times \mathbb{SD}$

```
    $preCalls.(a, [\,])$
$=$
    $[\,]$

    $preCalls.(\ (cr_a, rf_a, rt_a, i_a, m_a)\ ,\ [(cr_b, rf_b, rt_b, i_b, m_b)]{+}{+}seq\ )$
$=$
    $if\ (cr_a, rf_a, rt_a, i_a, m_a) = (cr_b, rf_b, rt_b, i_b, m_b) \rightarrow$
      $[\,]$
    $[]\ (cr_a, rf_a, rt_a, i_a, m_a) \neq (cr_b, rf_b, rt_b, i_b, m_b) \wedge cr_b = return \rightarrow$
      $preCalls((cr_a, rf_a, rt_a, i_a, m_a), seq)$
    $[]\ (cr_a, rf_a, rt_a, i_a, m_a) \neq (cr_b, rf_b, rt_b, i_b, m_b) \wedge cr_b = call \rightarrow$
      $\{(cr_b, rf_b, rt_b, i_b, m_b)\} \cup preCalls.((cr_a, rf_a, rt_a, i_a, m_a), seq)$
    $fi$
```

The *preCalls* function checks a given sequence from the beginning to the end (empty sequence) until it encounters the given action. All call-actions found on this path are united.

Now we encounter a difficulty when we have two identical call-actions and only one return-action. We will not find an error when we check this. We could change our *checkSD* function to verify the number of calls and returns to solve this problem. There is however another problem we found using our current representation of sequence diagrams. Being a single-threaded sequence, we will have to make sure that every return belongs to the last *unreturned* call. Therefore we opt to leave this representation and use a slightly more comprehensive one.

3.3 Sequence diagram representation (Option 2)

A problem that arises when we use the representation described in the previous section is that we cannot detect which return-action belongs to which call. Once we have two identical calls (and returns) in a certain sequence, there are two possible interpretations and we cannot distinguish them. The figure below illustrates this problem.

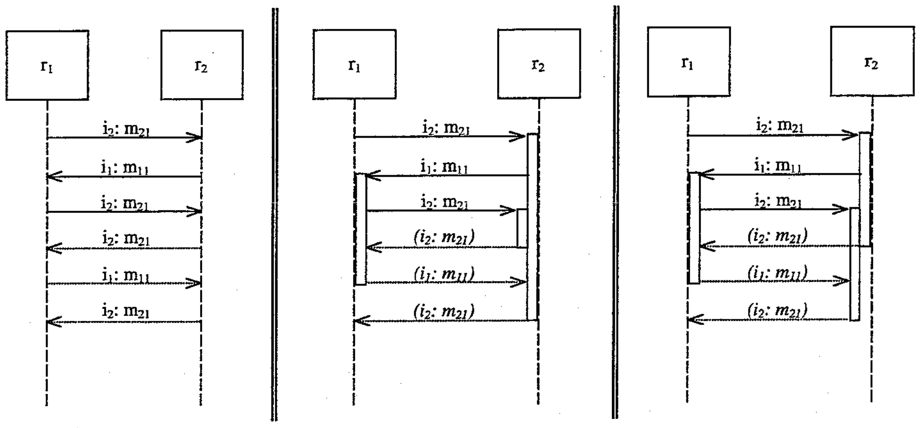


fig. 14: Two possible interpretations of one sequence

The rightmost picture does not describe single threaded behaviour, where the picture in the middle does. Our formal representation therefore has to include a way to identify the vertical boxes, showing the execution of a method. We will introduce a type $\mathbb{X}$ for this.
These boxes automatically connect the call-action that opens the box to the return-action that closes it. It can also show the execution time of a method, but we are not interested in that here.
We are now able to omit the interface and method parameters of a return-action, since they are already given by the connected call-action.

We now give a formal representation of a sequence of actions including boxes and omitting interface- and methodnames on return actions. Note that we will give a definition of any (sub)sequence of call- and return-actions, not necessarily a complete sequence described by a sequence diagram. We mention this because the name, $\mathbb{SD}$, might suspect otherwise.

$$\mathbb{SD} = seq.\mathbb{A}$$
$$\mathbb{A} = \mathbb{R}\times\mathbb{R}\times\mathbb{I}\times\mathbb{M}\times\mathbb{X} + \mathbb{R}\times\mathbb{R}\times\mathbb{X}$$

An action ($\in \mathbb{A}$) can either be a call-action (consisting of two role-identifiers, one interface-, one method- and one box-identifier) or a return-action (consisting of two role-identifiers and one box-identifier). The + denotes that an action can either be of the first form ($\curvearrowright\textcircled{0}$) or of the second ($\curvearrowright\textcircled{1}$).
We introduce the following *call-* and *return*-functions to clarify and simplify notation.

$$call \in \mathbb{A} \longmapsfrom \mathbb{R}\times\mathbb{R}\times\mathbb{I}\times\mathbb{M}\times\mathbb{X}$$
$$call.(r,r,i,m,x) = \{(r,r,i,m,x)\curvearrowright\textcircled{0}\}$$

$$return \in \mathbb{A} \longmapsfrom \mathbb{R}\times\mathbb{R}\times\mathbb{X}$$
$$return.(r,r,x) = \{(r,r,x)\curvearrowright\textcircled{1}\}$$

Example

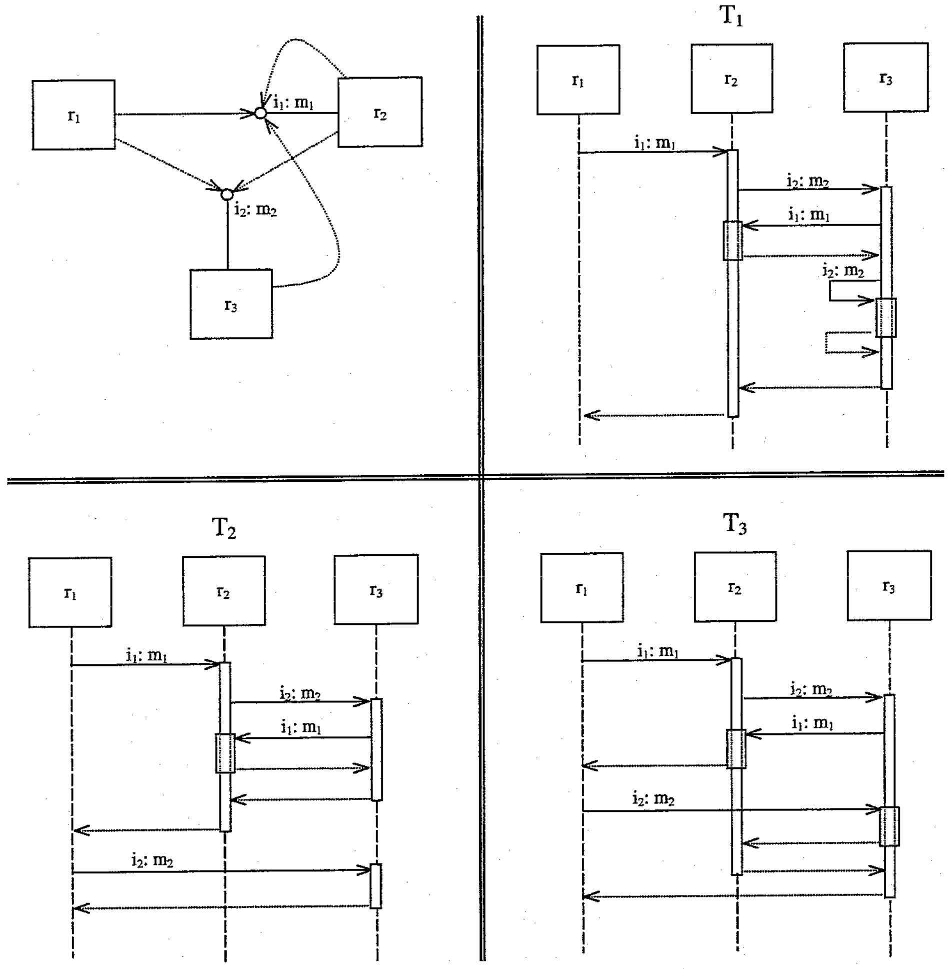


fig. 15: An interface role diagram and three sequence diagrams

$$T_1 = [\mathit{call}.(r_1, r_2, i_1, m_1, x_1), \mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3), \\ \mathit{return}.(r_2, r_3, x_3), \mathit{call}.(r_3, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_3, x_4), \\ \mathit{return}.(r_3, r_2, x_2), \mathit{return}.(r_2, r_1, x_1)]$$

$$T_2 = [\mathit{call}.(r_1, r_2, i_1, m_1, x_1), \mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3), \\ \mathit{return}.(r_2, r_3, x_3), \mathit{return}.(r_3, r_2, x_2), \mathit{return}.(r_2, r_1, x_1), \\ \mathit{call}.(r_1, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_1, x_4)]$$

$$T_3 = [\mathit{call}.(r_1, r_2, i_1, m_1, x_1), \mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3), \\ \mathit{return}.(r_2, r_1, x_3), \mathit{call}.(r_1, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_2, x_4), \\ \mathit{return}.(r_2, r_3, x_1), \mathit{return}.(r_3, r_1, x_2)]$$

3.4 Sequence Diagram Verification

In the previous chapter we showed the verifications we need to make to construct well defined projects. Now we want to verify well defined sequence diagrams that are *based* on a project.

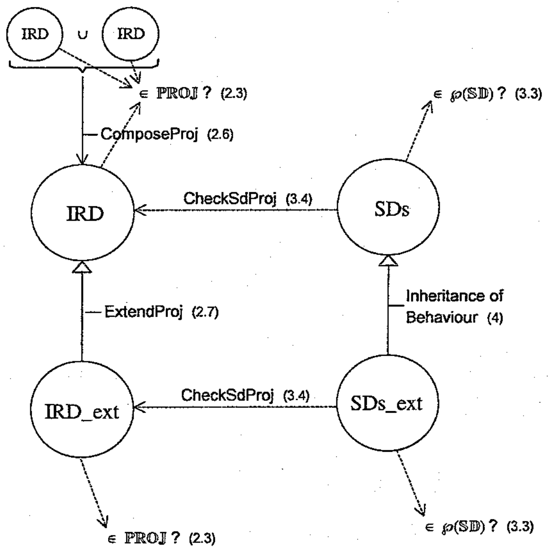


fig. 16: An overview of all the our verifications.

The picture above puts together the verifications we made in the previous chapter, with the ones we are going to make in this and the next chapter. A basic formalisation of sequences was introduced in section 3.3. In this section we will verify correctness of a sequence by two checks:
- *CheckSdProj*: Checks if a sequence correctly belongs to a project, i.e. the right methods are called on the right interfaces of the right roles.
- *ThreadSd*: Checks a sequence for single-threadedness.
After that we are going to verify inheritance of behaviour by comparing a set of sequence diagrams of one project to a set of sequence diagrams of an extension (by inheritance) of that project. This will be the topic of the following chapter.

$$CheckSdProj \in \mathbb{B} \longleftarrow\!\!\dashv \mathbb{SD}\times\mathbb{PROJ}$$

$$\begin{array}{ll} & CheckSdProj.(SD, proj) \\ = & \quad \{proj = (S, R, I, M, PR, PI, RI, PM, SS, SR)\} \\ & \forall(r_f, r_t, i, m, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ SD = A_b +\!\!+ [call.(r_f, r_t, i, m, x)] +\!\!+ A_a \mid \\ & \qquad\qquad i \,(PM)\, m \wedge i \in (obligs.proj).r_t \wedge i \in (rights.proj).r_f) \end{array}$$

For each call-action in the sequence the following three properties must hold:
- $i\,(PM)\,m$: The method will have to be provided by the interface.

- $i \in (obligs.proj).r_t$: The role receiving the call must have the obligation to "provide" the interface.
- $i \in (rights.proj).r_f$: The calling role must have the right to use that interface.

$$ThreadSd \in \mathbb{B} \longleftarrow \mathbb{SD}$$

$$
\begin{array}{ll}
 & ThreadSd.(SD) \\
= & \quad \{\} \\
 & \forall(r_f, r_t, i, m, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \;\wedge\; SD = A_b{+\!\!+}[call.(r_f, r_t, i, m, x)]{+\!\!+}A_a \mid \\
 & \qquad\qquad \exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid A_a = A_{b2}{+\!\!+}[return.(r_t, r_f, x)]{+\!\!+}A_{a2})) \\
\wedge & \\
 & \forall(r_f, r_t, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \;\wedge\; SD = A_b{+\!\!+}[return.(r_f, r_t, x)]{+\!\!+}A_a \mid \\
 & \qquad\qquad \exists(i, m \mid\mid LastCall.(A_b) = call.(r_t, r_f, i, m, x))) \\
\wedge & \\
 & FlowOfControlSD.(SD)
\end{array}
$$

For each call-action in the sequence:
- $\exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid A_a = A_{b2}{+\!\!+}[return.(r_t, r_f, x)]{+\!\!+}A_{a2})$:
A return-action has to follow the call-action. This return has to be:
- from the role that received the call (r_t).
- to the role that send the call (r_f).
- closing the right box (x).

For each return-action in the sequence:
- $\exists(i, m \mid\mid LastCall(A_b) = call.(r_f, r_t, i, m, x))$
A return-action has to close the box that was opened by the last unreturned call, prior to that return-action. We will show the details of the introduced function *LastCall* below.

FlowOfControlSD.(SD):
Since we want our sequences to be single-threaded, we need more than just requiring that each return belongs to the last unreturned call. We also want each action to be initiated by the receiving role of the previous action. We will introduce the boolean function *FlowOfControlSD* for this.

$LastCall \in \mathbb{A} \longleftarrow \mathbb{SD}$

$$
\begin{array}{ll}
 & LastCall.[\,] \\
= & \\
 & \varnothing
\end{array}
$$

$$
\begin{array}{ll}
 & LastCall.(SD_i +\!\!+ [a]) \\
= & \\
 & \begin{array}{lll}
 \textit{if}\ \exists(r_1, r_2, i, m, x \mid\mid a = call.(r_1, r_2, i, m, x)) & \rightarrow & a \\
 [\,]\ \exists(r_1, r_2, x \mid\mid a = return.(r_1, r_2, x)) & \rightarrow & LastCall.(RemoveBox.(x, SD_i)) \\
 \textit{fi} & &
 \end{array}
\end{array}
$$

The function *LastCall* takes the last action of a sequence (returning nothing when encountering an empty sequence), and immediately returns it if that last action is a call. If that last action is a return however (the only other possible action), we want to have the last call prior to the box that this return-action closes. This is because the call that opens this box, will be a *returned* call, while we are looking for the last *unreturned* one.
We are now left with defining the *RemoveBox* function.

$RemoveBox \in \mathbb{SD} \longleftarrow \mathbb{X} \times \mathbb{SD}$

$$
\begin{array}{ll}
 & RemoveBox.(x, [\,]) \\
= & \\
 & [\,]
\end{array}
$$

$$
\begin{array}{ll}
 & RemoveBox.(x, (SD_i +\!\!+ [a])) \\
= & \\
 & \begin{array}{lll}
 \textit{if}\ \exists(r_1, r_2, i, m, x_1 \mid x_1 \neq x \mid a = call.(r_1, r_2, i, m, x_1)) & \rightarrow & [\,] \\
 [\,]\ \exists(r_1, r_2, i, m, x_1 \mid x_1 = x \mid a = call.(r_1, r_2, i, m, x_1)) & \rightarrow & SD_i \\
 [\,]\ \exists(r_1, r_2, x_1 \mid x_1 \neq x \mid a = return.(r_1, r_2, x_1)) & \rightarrow & \\
 \qquad RemoveBox.(x, (RemoveBox.(x_1, SD_i)) & & \\
 [\,]\ \exists(r_1, r_2, x_1 \mid x_1 = x \mid a = return.(r_1, r_2, x_1)) & \rightarrow & [\,] \\
 \textit{fi} & &
 \end{array}
\end{array}
$$

The *RemoveBox* function is given a box identifier and a sequence prior to the return action of that box. The last action of this sequence can either be a call-action with the same box identifier, or another return with (another) box identifier. It cannot be a call with another box, because then we would not be returning the last unreturned call. It is an obvious error to encounter another return-action with the same box identifier
When the right call is encountered, we return the sequence prior to that (we've succesfully removed the box). When another return is encountered, we have to remove its box as well.

The third conjunct of our *ThreadSd* function uses a boolean function *FlowOfControl* to verify, like the name says, the flow of control of a sequence of actions. It is formally defined as follows:

$$\mathit{FlowOfControlSD} \in \mathbb{B} \longleftarrow \mathbb{SD}$$

$$\mathit{FlowOfControlSD}.[\,]$$
$$=$$
$$\mathit{true}$$

$$\mathit{FlowOfControlSD}.([a]{+\!\!+}SD_i)$$
$$=$$
$$\begin{array}{lll} \mathit{if} & \exists(r_1,r_2,i,m,x \;||\; a = \mathit{call}.(r_1,r_2,i,m,x)) & \rightarrow \mathit{FlowRoleSD}.(r_2, SD_i) \\ [\,] & \exists(r_1,r_2,x \;||\; a = \mathit{return}.(r_1,r_2,x)) & \rightarrow \mathit{FlowRoleSD}.(r_2, SD_i) \\ \mathit{fi} & & \end{array}$$

The first action of an (unempty) sequence can be from any role to any role, where the second action has to originate from the receiving role of the first. *FlowRole* is introduced.

$$\mathit{FlowRoleSD} \in \mathbb{B} \longleftarrow \mathbb{R}\times\mathbb{SD}$$

$$\mathit{FlowRoleSD}.(r, [\,])$$
$$=$$
$$\mathit{true}$$

$$\mathit{FlowRoleSD}.(r_i, [a{+\!\!+}SD_j])$$
$$=$$
$$\begin{array}{lll} \mathit{if} & \exists(r_1,r_2,i,m,x \;||\; a = \mathit{call}.(r_1,r_2,i,m,x) \;\wedge\; r_i{=}r_1) & \rightarrow \mathit{FlowRole}.(r_2, SD_j) \\ [\,] & \exists(r_1,r_2,i,m,x \;||\; a = \mathit{call}.(r_1,r_2,i,m,x) \;\wedge\; r_i{\neq}r_1) & \rightarrow \mathit{false} \\ [\,] & \exists(r_1,r_2,x \;||\; a = \mathit{return}.(r_1,r_2,x) \;\wedge\; r_i{=}r_1) & \rightarrow \mathit{FlowRole}.(r_2, SD_j) \\ [\,] & \exists(r_1,r_2,x \;||\; a = \mathit{return}.(r_1,r_2,x) \;\wedge\; r_i{\neq}r_1) & \rightarrow \mathit{false} \\ \mathit{fi} & & \end{array}$$

The first action of the sequence has to be from the same role as the given action and furthermore the receiving role of that first action now is the "flowing" role.

We will now verify the correctness of the three sequences of figure 15.
At first we need to formalise the project, also given by the interface role diagram in figure 15, we will call this $Proj_4$.

$Proj_4 = (S_4, R_4, I_4, M_4, PR_4, PI_4, RI_4, PM_4, SS_4, SR_4)$

With:

$$
\begin{array}{ll}
S_4 & = \{s_1\} \quad \text{note: There should have been a suite in the picture!} \\
R_4 & = \{r_1, r_2, r_3\} \\
I_4 & = \{i_1, i_2\} \\
M_4 & = \{m_1, m_2\} \\
PR_4 & = \{s_1 \backsim r_1, s_1 \backsim r_2, s_1 \backsim r_3\} \\
PI_4 & = \{r_2 \backsim i_1, r_3 \backsim i_2\} \\
RI_4 & = \{r_1 \backsim i_1, r_1 \backsim i_2, r_2 \backsim i_1, r_2 \backsim i_2, r_3 \backsim i_1\} \\
PM_4 & = \{i_1 \backsim m_1, i_2 \backsim m_2\} \\
SS_4 & = \varnothing \\
SR_4 & = \varnothing
\end{array}
$$

It is left to the reader to verify that $Proj_4$ is a correctly defined project, $\in \mathbb{PROJ}$. Sequences T_1 and T_2 are correct, although we will not show their verification details here.
We will show the incorrectness the sequences T_3 due to incorrectly returning call-actions. T_3 is correct with respect to its interface role diagram ($Proj_4$). We'll show this first.

$$
\begin{aligned}
T_3 = [&call.(r_1, r_2, i_1, m_1, x_1), call.(r_2, r_3, i_2, m_2, x_2), call.(r_3, r_2, i_1, m_1, x_3), \\
&return.(r_2, r_1, x_3), call.(r_1, r_3, i_2, m_2, x_4), return.(r_3, r_2, x_4), \\
&return.(r_2, r_3, x_1), return.(r_3, r_1, x_2)]
\end{aligned}
$$

$$
\begin{array}{ll}
 & CheckSdProj.(T_3, Proj_4) \\
= & \quad \{\text{definition of } CheckSdProj\} \\
 & \forall (r_f, r_t, i, m, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ T_3 = A_b +\!\!+ [call.(r_f, r_t, i, m, x)] +\!\!+ A_a \mid \\
 & \qquad i \ (PM_4)\ m \wedge i \in (obligs.Proj_4).r_t \wedge i \in (rights.Proj_4).r_f)
\end{array}
$$

We have four call-actions in our sequence T_3, $call.(r_1, r_2, i_1, m_1, x_1)$, $call.(r_2, r_3, i_2, m_2, x_2)$, $call.(r_3, r_2, i_1, m_1, x_3)$ and $call.(r_1, r_3, i_2, m_2, x_4)$. This will lead to the following obligations:

- $i_1 \ (PM_4)\ m_1$ and $i_2 \ (PM_4)\ m_2$. Both ok.

- $i_1 \in (obligs.Proj_4).r_2$ and $i_2 \in (obligs.Proj_4).r_3$. Since we have no inheritance here, the *obligs* function boils down to provides interface relation PI_4. $r_2 \ (PI_4)\ i_1$ and $r_3 \ (PI_4)\ i_2$, Both ok.

- $i_1 \in (rights.Proj_4).r_1$, $i_2 \in (rights.Proj_4).r_2$, $i_1 \in (rights.Proj_4).r_3$ and $i_2 \in (rights.Proj_4).r_1$. All ok, see required interface relation RI_4.

The sequence is correct as far as our project is concerned: No incorrect calls of methods on interfaces.

$$\begin{array}{ll}
& \mathit{ThreadSd}.(T_3) \\
= & \quad \{\text{definition of } \mathit{ThreadSd}\} \\
& \forall(r_f, r_t, i, m, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ T_3 = A_b{+\!\!+}[\mathit{call}.(r_f, r_t, i, m, x)]{+\!\!+}A_a \mid \\
& \qquad \exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid A_a = A_{b2}{+\!\!+}[\mathit{return}.(r_t, r_f, x)]{+\!\!+}A_{a2})) \\
\wedge & \\
& \forall(r_f, r_t, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ T_3 = A_b{+\!\!+}[\mathit{return}.(r_f, r_t, x)]{+\!\!+}A_a \mid \\
& \qquad \exists(i, m \mid\mid \mathit{LastCall}(A_b) = \mathit{call}.(r_t, r_f, i, m, x))) \\
\wedge & \\
& \mathit{FlowOfControlSD}.(T_3)
\end{array}$$

We now have three conjuncts. We'll discuss them one by one.

$$\begin{array}{l}
\forall(r_f, r_t, i, m, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ T_3 = A_b{+\!\!+}[\mathit{call}.(r_f, r_t, i, m, x)]{+\!\!+}A_a \mid \\
\qquad \exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid A_a = A_{b2}{+\!\!+}[\mathit{return}.(r_t, r_f, x)]{+\!\!+}A_{a2}))
\end{array}$$

- For the first call of the sequence, $\mathit{call}.(r_1, r_2, i_1, m_1, x_1)$, this means:
$\exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid [\mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3),$
$\quad \mathit{return}.(r_2, r_1, x_3), \mathit{call}.(r_1, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_2, x_4), \mathit{return}.(r_2, r_3, x_1),$
$\quad \mathit{return}.(r_3, r_1, x_2)] = A_{b2}{+\!\!+}[\mathit{return}.(r_2, r_1, x_1)]{+\!\!+}A_{a2}))$
NOT ok, we have $\mathit{return}.(r_2, r_3, x_1)$, which means the call is returned to the wrong role. We can see in the picture that this is indeed the case.
- For the second call, $\mathit{call}.(r_2, r_3, i_2, m_2, x_2)$:
$\exists(A_{b2}, A_{a2} \mid A_{b2}, A_{a2} \in \mathbb{SD} \mid [\mathit{call}.(r_3, r_2, i_1, m_1, x_3), \mathit{return}.(r_2, r_1, x_3),$
$\quad \mathit{call}.(r_1, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_2, x_4), \mathit{return}.(r_2, r_3, x_1), \mathit{return}.(r_3, r_1, x_2)]$
$\quad = A_{b2}{+\!\!+}[\mathit{return}.(r_3, r_2, x_2)]{+\!\!+}A_{a2}))$
Also NOT ok. The second call of the sequence T_3 is also returned to the wrong role.
In fact, all return-actions of this sequence are returned incorrectly. We will still verify the other two conjuncts of the *ThreadSd* function, although we already know that the sequence is incorrect.

$$\begin{array}{l}
\forall(r_f, r_t, x, A_b, A_a \mid A_b, A_a \in \mathbb{SD} \ \wedge \ T_3 = A_b{+\!\!+}[\mathit{return}.(r_f, r_t, x)]{+\!\!+}A_a \mid \\
\qquad \exists(i, m \mid\mid \mathit{LastCall}(A_b) = \mathit{call}.(r_t, r_f, i, m, x))).
\end{array}$$

Each return-action in the sequence has to close the box that was opened by the last unreturned call. We'll show the verification of the third return-action of T_3, being $\mathit{return}.(r_2, r_3, x_1)$ only:
$\exists(i, m \mid\mid \mathit{LastCall}.([\mathit{call}.(r_1, r_2, i_1, m_1, x_1), \mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3),$
$\quad \mathit{return}.(r_2, r_1, x_3), \mathit{call}.(r_1, r_3, i_2, m_2, x_4), \mathit{return}.(r_3, r_2, x_4)]) = \mathit{call}.(r_3, r_2, i, m, x_1)))$

We will first work out the LastCall function, with SD_1=$[\mathit{call}.(r_1, r_2, i_1, m_1, x_1),$ $\mathit{call}.(r_2, r_3, i_2, m_2, x_2), \mathit{call}.(r_3, r_2, i_1, m_1, x_3), \mathit{return}.(r_2, r_1, x_3), \mathit{call}.(r_1, r_3, i_2, m_2, x_4)]$.
This leaves $\mathit{LastCall}.(SD_1{+\!\!+}[\mathit{return}.(r_3, r_2, x_4)]$, giving:

if $\exists(r_i, r_j, i, m, x \,\|\, [\mathit{return}.(r_3, r_2, x_4)] = \mathit{call}.(r_i, r_j, i, m, x)) \;\rightarrow\; [\mathit{return}.(r_3, r_2, x_4)]$
$[] \; \exists(r_i, r_j, x_k \,\|\, [\mathit{return}.(r_3, r_2, x_4)] = \mathit{return}.(r_i, r_j, x_k)) \;\rightarrow\; \mathit{LastCall}.(\mathit{RemoveBox}.(x_k, SD_1))$
fi

The second guard is met, the last action, before the return we wanted to verify, is also a return. This return closes a box that was opened by a, now returned, call. We'll have to remove this box, because we want to find the last *un*returned call. *RemoveBox*.(x_4, SD_1) will remove the call that opens box x_4, although it is called by (or returned to) the wrong role. We would have noticed this error on checking the second return of the sequence.
The second invocation of the *LastCall* function will again encounter a return, this time of box x_3. The previous action is also a call on x_3, that will be removed. Then *LastCall* will be invoked for the third time, this time with the following sequence: $[\mathit{call}.(r_1, r_2, i_1, m_1, x_1), \mathit{call}.(r_2, r_3, i_2, m_2, x_2)]$. The LastCall of this sequence is obviously not equal to $\mathit{call}.(r_3, r_2, i_?, m_?, x_1)$, for any interface-method combination.
So we found another flaw in our sequence, although this one was to be expected from the previous verifications.

FlowOfControlSD.(T_3)

The flow of control checks out positively, which can be verified very easily. It is left for the reader to do so.

Chapter 4

Inheritance of behaviour

4.1 Introduction

In chapter 2 we've seen a notion of inheritance on the ISpec interface role diagrams. We were dealing with static inheritance there, inheritance on the structure of a system. This form of inheritance is well known and widely used.
In this chapter we want to formulate a notion of inheritance that adresses behaviour, making use of ISpec sequence diagrams. This form of inheritance, dynamic inheritance, is not conclusively formalised. Research has been done in this area however, like in [2], [3], [4], [7], [9], [10] and [11].
We give some definitions of inheritance of behaviour:

Protocol Inheritance, from [4]
If it is not possible to distinguish the external behaviour of *son* and *father* when only methods of *son* that are also present in *father* are executed, then *son* is a subclass of *father.*

In essence, this form of inheritance boils down to *blocking* the actions that are *new* to the son, resulting in equal behaviour with the father.

Projection Inheritance, from [4]
If it is not possible to distinguish the external behaviour of *son* and *father* when arbitrary methods of *son* are executed, but only the effects of methods that are also present in *father* are considered, then *son* is a subclass of *father.*

This means, that instead of blocking *new* actions, we are now *hiding* them, silently ignoring them in our behavioural description. After hiding the actions, parent and child behaviour have to be exactly the same again.

Life-cycle Inheritance, from [4]
First some set of actions is *blocked*, then some *disjoint* set of actions is *hidden.*

The result has to equal parent behaviour.

Both projection and protocol inheritance are contained by Life-cycle inheritance, by hiding all *new* actions (and blocking none) respectively blocking all *new* actions (and hiding none).

Consistent Inheritance, from [11]
Consistent inheritance requires that each possible sequence of a *son*, disregarding newly added activities and states, must also be a sequence of the *father*. Or in other words, consistent inheritance requires that a sequence of the *father* reflects in each sequence of the *son*.

This is almost the same definition as the projection inheritance from [4], it only allows the son to inherit a part of the fathers' behaviour, instead of requiring it to inherit all of the fathers behaviour.
Below we introduce a notion of inheritance of behaviour of an ISpec model, consisting of a static part, an interface role diagram, and a dynamic part, a set of sequence diagrams. As mentioned before, a set of sequence diagrams only describes some (important) subset of all possible behaviour. When we are verifying a set of sequence diagrams of a son to that of a father, we do not require the sons behaviour to be equal, but we do want the fathers behaviour to reflect the sons, like the consistent inheritance above. This will become quite obvious when we think about inheriting from a composition of projects. We should be able to inherit a part of both composed projects, instead of having to inherit all possible behaviour.
Therefore we adopt the definition of consistent inheritance, with only a small modification, or better, extension. We also want our son to be allowed to have sequences consisting of new actions only. This leads to the following definition:

Sequence Inheritance
Sequence inheritance requires that each possible sequence of a *son*, disregarding newly added actions, must be either a sequence of the *father* or an empty sequence.

In the next section we will formalise this notion of inheritance of behaviour. Afterwards, we'll discuss some test cases. In the following chapter we will use the same test cases to analyse an ISpec based tool [12] that tries to verify life-cycle inheritance in accordance to the definition of [4].

4.2 The formal verifications

We will introduce a boolean function *InhOfBeh* to verify *sequence inheritance*. We have to compare one set of sequence diagrams to another set of sequence diagrams. Both sets of sequence diagrams have to be correct: belong to the right project and describe a correct thread, which we can verify using the *CheckSdProj* respectively *ThreadSd* functions of the former chapter. One of these projects (the *childs*) has to be the result of extending the other one (the *parents*). We

can verify this by using the *ExtendProj* function of chapter 2.7.
We'll use abbreviations for our 10-tuple projects below, assuming $Proj_X$ to be $(S_X, R_X, I_X, M_X, PR_X, PI_X, RI_X, PM_X, SS_X, SR_X)$.

$$InhOfBeh \in \mathbb{B} \longleftarrow \mathbb{PROJ} \times \wp(\mathbb{SD}) \times \mathbb{PROJ} \times \wp(\mathbb{SD})$$

$$(Proj_{old},\ SDs_{old},\ Proj_{new},\ SDs_{new}) \in InhOfBeh\rangle$$

$$\equiv$$

$$\exists(Suite_{new} \mid Suite_{new} \in \mathbb{SUITE} \mid Proj_{new} = ExtendProj.(Proj_{old}, Suite_{new})$$
$$\forall(SD_i \mid SD_i \in SDs_{old} \mid CheckSdProj.(SD_i, Proj_{old}) \wedge ThreadSd.Sd_i)$$
$$\forall(SD_j \mid SD_j \in SDs_{new} \mid CheckSdProj.(SD_j, Proj_{new}) \wedge ThreadSd.Sd_j)$$

$$InhOfBeh.(Proj_{old}, SDs_{old}, Proj_{new}, SDs_{new})$$

$$\equiv$$

$$\forall(SD_i \mid SD_i \in SDs_{new} \mid Filter.(SD_i, Proj_{new}, Proj_{old}, \emptyset) = [\,] \ \vee\ \exists(SD_j \mid SD_j \in SDs_{old}$$
$$\mid Filter.(SD_i, Proj_{new}, Proj_{old}, \emptyset) = Rename.(SD_j, SD_i, Proj_{new}, \emptyset, \emptyset)))$$

The domain restrictions of the *InhOfBeh* function are already discussed above. The result of the inheritance verification boils down to checking all new (*child*) sequences individually. The following will be checked:
Filter.(SD_i, $Proj_{new}$, $Proj_{old}$, $\emptyset$): A new sequence (SD_i) may contain new actions that weren't available to the old roles. These actions have to be disregarded, filtered. If this filtering leads to an empty sequence, we are dealing with a sequence that only contains new actions. Inheritance will be automatically correct, since there are no inherited actions in the sequence. If the filtering does not result in an emtpy sequence, the remaining actions will have to be in accordance with one of the parent sequences.
Rename.(SD_j, SD_i, $Proj_{new}$, $\emptyset$, $\emptyset$): A sequence SD_j (old sequence) will be renamed, such that rolenames and boxnames will become equal to that of the other (new) sequence SD_i. This renaming takes place only when these new roles inherit from the old roles and for actions that coincide (same method-calls or -returns). The result will be an old sequence, having the rolenames and boxnames of the new one, that needs to be checked for inheritance.
All nonempty filtered child sequences should be equal to one of the renamed parent sequences to derive correct inheritance.
We discuss the *Filter* function first, the *Rename* function afterwards.

$$\mathit{Filter} \in \mathbb{SD} \leftarrowtail \mathbb{SD}\times\mathbb{PROJ}\times\mathbb{PROJ}\times\wp(\mathbb{X})$$

$$\mathit{Filter}.([\,], \mathit{Proj}_n, \mathit{Proj}_o, \mathit{Xs})$$
$$=$$
$$[\,]$$

$$\mathit{Filter}.([a]{+\!\!+}\mathit{SD}_i, \mathit{Proj}_n, \mathit{Proj}_o, \mathit{Xs})$$
$$=$$
$$\begin{array}{lll}
\mathit{if} & \exists(r_1, r_2, i, m, x \mathrel{\|} a{=}\mathit{call}.(r_1, r_2, i, m, x) \wedge i \in I_o) & \rightarrow\; [a]{+\!\!+}\mathit{Filter}.(\mathit{SD}_i, \mathit{Proj}_n, \mathit{Proj}_o, \mathit{Xs}) \\
[\,] & \exists(r_1, r_2, i, m, x \mathrel{\|} a{=}\mathit{call}.(r_1, r_2, i, m, x) \wedge i \notin I_o) & \rightarrow\; \mathit{Filter}.(\mathit{SD}_i, \mathit{Proj}_n, \mathit{Proj}_o, \{\mathit{Xs} \cup x\}) \\
[\,] & \exists(r_1, r_2, x \mathrel{\|} a{=}\mathit{return}.(r_1, r_2, x) \wedge x \notin \mathit{Xs}) & \rightarrow\; [a]{+\!\!+}\mathit{Filter}.(\mathit{SD}_i, \mathit{Proj}_n, \mathit{Proj}_o, \mathit{Xs}) \\
[\,] & \exists(r_1, r_2, x \mathrel{\|} a{=}\mathit{return}.(r_1, r_2, x) \wedge x \in \mathit{Xs}) & \rightarrow\; \mathit{Filter}.(\mathit{SD}_i, \mathit{Proj}_n, \mathit{Proj}_o, \{\mathit{Xs} \setminus x\}) \\
\mathit{fi} & &
\end{array}$$

This function takes the first action of a sequence. If it's a call of a method of an old interface it remains, new interface-calls are left out. We save the box-identifiers of those calls that filtered out, since we'll have to remove the returns of these calls as well.

$$\mathit{Rename} \in \mathbb{SD} \leftarrowtail \mathbb{SD}\times\mathbb{SD}\times\mathbb{PROJ}\times\wp(\mathbb{X})\times\wp(\mathbb{X})$$

$$\mathit{Rename}.([\,], \mathit{SD}_n, \mathit{Proj}_n, X_c, X_r)$$
$$=$$
$$[\,]$$

$$\mathit{Rename}.([a]{+\!\!+}\mathit{SD}_i, \mathit{SD}_n, \mathit{Proj}_n, X_c, X_r)$$
$$=$$
$$\begin{array}{ll}
\mathit{if} & \exists(r_{o1}, r_{o2}, r_{n1}, r_{n2}, i, m, x_o, x_n, A_b, A_a \mid x_n \notin X_c \mid a{=}\mathit{call}.(r_{o1}, r_{o2}, i, m, x_o) \\
& \quad \wedge\, r_{n1} \langle \mathit{SR}_n \rangle\, r_{o1} \wedge r_{n2} \langle \mathit{SR}_n \rangle\, r_{o2} \wedge \mathit{SD}_n{=}[A_b{+\!\!+}[\mathit{call}.(r_{n1}, r_{n2}, i, m, x_n)]{+\!\!+}A_a] \\
& \quad \wedge\, \neg\exists(A_{bb}, A_{aa}, x_{nn} \mid x_{nn} \notin X_c \mid A_b{=}[A_{bb}{+\!\!+}[\mathit{call}.(r_{n1}, r_{n2}, i, m, x_{nn})]{+\!\!+}A_{aa}])) \\
& \rightarrow \\
& [\mathit{call}.(r_{n1}, r_{n2}, i, m, x_n)]{+\!\!+}\mathit{Rename}.(\mathit{SD}_i, \mathit{SD}_n, \mathit{Proj}_n, \{X_c \cup x_n\}, \{X_r \cup x_n\}) \\
[\,] & \exists(r_{o1}, r_{o2}, r_{n1}, r_{n2}, x_o, x_n, A_b, A_a \mid x_n \in X_r \mid a{=}\mathit{return}.(r_{o1}, r_{o2}, x_o) \\
& \quad \wedge\, r_{n1} \langle \mathit{SR}_n \rangle\, r_{o1} \wedge r_{n2} \langle \mathit{SR}_n \rangle\, r_{o2} \wedge \mathit{SD}_n{=}[A_b{+\!\!+}[\mathit{return}.(r_{n1}, r_{n2}, x_n)]{+\!\!+}A_a] \\
& \quad \wedge\, \neg\exists(A_{bb}, A_{aa}, x_{nn} \mid x_{nn} \in X_r \mid A_b{=}[A_{bb}{+\!\!+}\mathit{return}.(r_{n1}, r_{n2}, x_{nn}){+\!\!+}A_{aa}])) \\
& \rightarrow \\
& [\mathit{return}.(r_{n1}, r_{n2}, x_n)]{+\!\!+}\mathit{Rename}.(\mathit{SD}_i, \mathit{SD}_n, \mathit{Proj}_n, X_c, \{X_r \setminus x_n\}) \\
[\,] & \mathit{else} \\
& \rightarrow \\
& [a]{+\!\!+}\mathit{Rename}.(\mathit{SD}_i, \mathit{SD}_n, \mathit{Proj}_n, X_c, X_r) \\
\mathit{fi} &
\end{array}$$

All actions of a parent sequence should have their role- and box-identifiers renamed, to be able to correspond to the (filtered) new sequence. An old call-action also occuring in the new sequence, with the new roles inheriting correctly from the old roles, should be renamed.

Achieved by: $a{=}\mathit{call}.(r_{o1}, r_{o2}, i, m, x_o) \wedge r_{n1}\ \{SR_n\}\ r_{o1} \wedge r_{n2}\ \{SR_n\}\ r_{o2}$
Problems arise when there are several instances of the same method-call in one or both (should be both, if they are to be correctly inheriting) sequences. We want to rename the first call of the old sequence so that the box-identifier coincides with the first call of the new sequence.
Hence: $\neg\exists(A_{bb}, A_{aa}, x_{nn} \mid x_{nn} \notin X_c \mid A_b{=}[A_{bb}{+\!\!+}[\mathit{call}.(r_{n1}, r_{n2}, i, m, x_{nn})]{+\!\!+}A_{aa}])$
Furthermore we want a second (or third, etc.) instance of a call in an old sequence to be renamed to the second one of the new sequence, which is the first one we haven't got already. Therefore we're keeping a set of boxes, X_c and require: $x_n \notin X_c$ and $x_{nn} \notin X_c$.

The same box-identifiers are also stored in another set, X_r. When we encounter a return-action in the old sequence it should be renamed to the first corresponding return-action of the new sequence. After that, we remove its box-identifier from the set X_r, so that the second the possible second identical return-action will also be renamed to the second one of the new sequence.

If we encounter an action in the old sequence that does not correspond with an action of the new sequence, we cannot rename this action. We'll leave it as it is, and rename the rest of the sequence, although we already know this will not be the sequence we're looking for.

4.3 Testing

In this section we do some testing on our notion of inheritance of behaviour. The same tests are executed using a ISpec related tool in the next chapter. We use some simple examples to examplify the differences in our approaches to handle inheritance of behaviour, using sequence diagrams.
We will first give our parent interface role diagram and its behaviour in terms of two sequence diagrams.

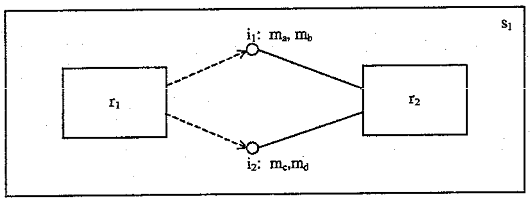


fig. 17: interface role diagram of the parent project.

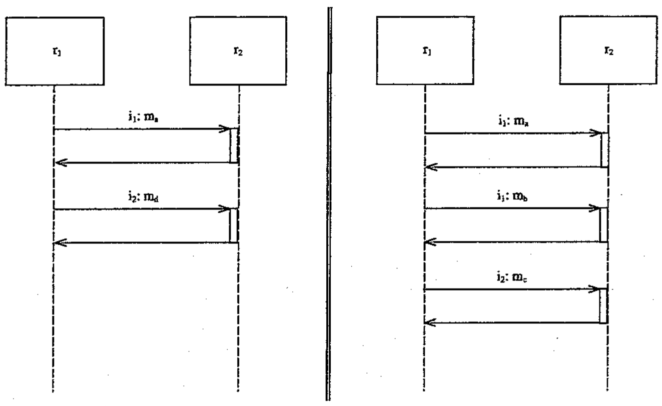


fig. 18: The parent behaviour in terms of sequence diagrams.

Our formal representation of the parent project now becomes:

$Proj_P = (S_P, R_P, I_P, M_P, PR_P, PI_P, RI_P, PM_P, SS_P, SR_P)$

With:

$$\begin{aligned}
S_P &= \{s_1\} \\
R_P &= \{r_1, r_2\} \\
I_P &= \{i_1, i_2\} \\
M_P &= \{m_a, m_b, m_c, m_d\} \\
PR_P &= \{s_1 \backsim r_1, s_1 \backsim r_2\} \\
PI_P &= \{r_2 \backsim i_1, r_2 \backsim i_2\} \\
RI_P &= \{r_1 \backsim i_1, r_1 \backsim i_2\} \\
PM_P &= \{i_1 \backsim m_a, i_1 \backsim m_b, i_2 \backsim m_c, i_2 \backsim m_d\} \\
SS_P &= \emptyset \\
SR_P &= \emptyset
\end{aligned}$$

The parent behaviour:

$$\begin{aligned}
Sd_{P1} = [&call.(r_1, r_2, i_1, m_a, x_{P1a}), return.(r_2, r_1, x_{P1a}), call.(r_1, r_2, i_2, m_d, x_{P1b}), \\
&return.(r_2, r_1, x_{P1b})]
\end{aligned}$$

$$\begin{aligned}
Sd_{P2} = [&call.(r_1, r_2, i_1, m_a, x_{P2a}), return.(r_2, r_1, x_{P2a}), call.(r_1, r_2, i_1, m_b, x_{P2b}), \\
&return.(r_2, r_1, x_{P2b}), call.(r_1, r_2, i_2, m_c, x_{P2c}), return.(r_2, r_1, x_{P2c})]
\end{aligned}$$

We also give the interface role diagram of the child, since we use this for all the tests to come. With each test the chid will have a different set of sequence diagrams, but the're all based on the following interface role diagram.

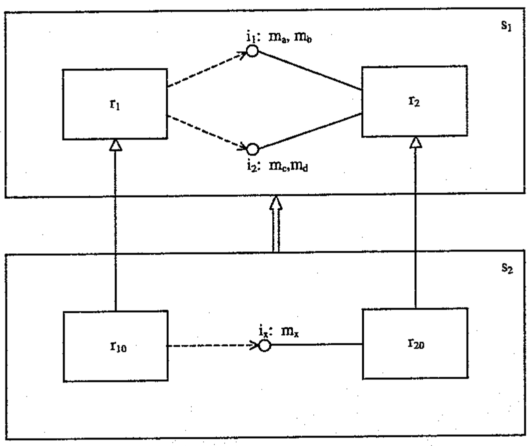


fig. 19: interface role diagram of the child project.

We've added one suite to the project of the parent, being:
$Suite_C = (\{s_2\}, \{r_{10}, r_{20}\}, \{i_x\}, \{m_x\}, \{s_2 \rightsquigarrow r_{10}, s_2 \rightsquigarrow r_{20}\}, \{r_{20} \rightsquigarrow i_x\}, \{r_{10} \rightsquigarrow i_x\}, \{i_x \rightsquigarrow m_x\},$
$\{r_{10} \rightsquigarrow r_1, r_{20} \rightsquigarrow r_2\}, \{s_2 \rightsquigarrow s_1\})$
The project of the child is an extension of the project of the parent by this suite $Suite_C$. The project of the child thus becomes:

$Proj_C = (S_C, R_C, I_C, M_C, PR_C, PI_C, RI_C, PM_C, SS_C, SR_c)$
$= ExtendProj.(Proj_P, Suite_C)$
This yields:

$$\begin{aligned}
S_C &= \{s_1, s_2\}\\
R_C &= \{r_1, r_2, r_{10}, r_{20}\}\\
I_C &= \{i_1, i_2, i_x\}\\
M_C &= \{m_a, m_b, m_c, m_d, m_x\}\\
PR_C &= \{s_1 \rightsquigarrow r_1, s_1 \rightsquigarrow r_2, s_2 \rightsquigarrow r_{10}, s_2 \rightsquigarrow r_{20}\}\\
PI_C &= \{r_2 \rightsquigarrow i_1, r_2 \rightsquigarrow i_2, r_{20} \rightsquigarrow i_x\}\\
RI_C &= \{r_1 \rightsquigarrow i_1, r_1 \rightsquigarrow i_2, r_{10} \rightsquigarrow i_x\}\\
PM_C &= \{i_1 \rightsquigarrow m_a, i_1 \rightsquigarrow m_b, i_2 \rightsquigarrow m_c, i_2 \rightsquigarrow m_d, i_x \rightsquigarrow m_x\}\\
SS_C &= \{r_{10} \rightsquigarrow r_1, r_{20} \rightsquigarrow r_2\}\\
SR_C &= \{s_2 \rightsquigarrow s_1\}
\end{aligned}$$

4.3.1 Deliberately making a mistake

We are now going to test child behaviour with respect to the parent behaviour, using our formal definition of inheritance of behaviour. We'll have two sequences inheriting from parent sequence Sd_{P2}, both also including call- and return-actions of the new method m_x on the new interface i_x. Only difference will be the moment of execution of these new actions. One of these two sequences however, is incomplete. We intentionally leave out, or forget to draw, the last call- and return-actions, calling a method that we inherited from our parent. Finally we'll also have a sequence inheriting from parent sequence Sd_{P2}, without even including a new action. Child behaviour becomes the following:

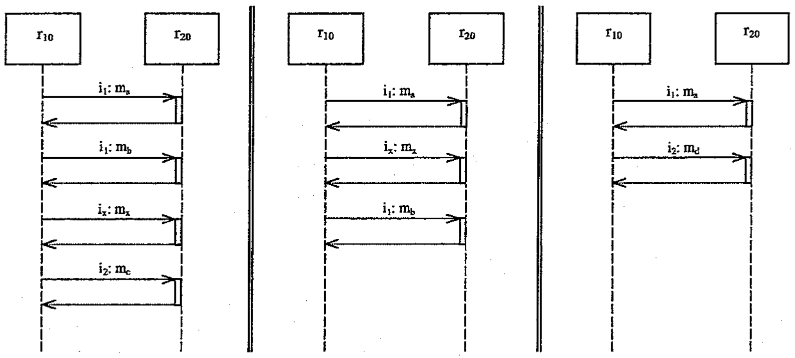


fig. 20: The child behaviour including one inconsistent sequence.

The child behaviour of this test case can be formally represented by following set of sequences: $Sd_{C1}, Sd_{C2}, Sd_{C3}$, with

$$\begin{aligned} Sd_{C1} = [&\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C1a}), \mathit{return}.(r_{20}, r_{10}, x_{C1a}), \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C1b}),\\ &\mathit{return}.(r_{20}, r_{10}, x_{C1b}), \mathit{call}.(r_{10}, r_{20}, i_x, m_x, x_{C1c}), \mathit{return}.(r_{20}, r_{10}, x_{C1c}),\\ &\mathit{call}.(r_{10}, r_{20}, i_2, m_c, x_{C1d}), \mathit{return}.(r_{20}, r_{10}, x_{C1d})] \end{aligned}$$

$$\begin{aligned} Sd_{C2} = [&\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a}), \mathit{call}.(r_{10}, r_{20}, i_x, m_x, x_{C2b}),\\ &\mathit{return}.(r_{20}, r_{10}, x_{C2b}), \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), \mathit{return}.(r_{20}, r_{10}, x_{C2c})] \end{aligned}$$

$$\begin{aligned} Sd_{C3} = [&\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C3a}), \mathit{return}.(r_{20}, r_{10}, x_{C3a}), \mathit{call}.(r_{10}, r_{20}, i_2, m_d, x_{C3b}),\\ &\mathit{return}.(r_{20}, r_{10}, x_{C3b})] \end{aligned}$$

Our inheritance check now becomes:
$\mathit{InhOfBeh}.(\mathit{Proj}_P, \{Sd_{P1}, Sd_{P2}\}, \mathit{Proj}_C, \{Sd_{C1}, Sd_{C2}, Sd_{C3}\})$

The domain restrictions of the *InhOfBeh* function are met, so there will be a result. This result is *false*, which means we have not derived correct inheritance. We will not give a complete derivation of all three child sequences, only the faulty sequence Sd_{C2}.
The following should hold: (for the other two sequences it does)

$$\begin{aligned} &\mathit{Filter}.(Sd_{C2},\ \mathit{Proj}_C,\ \mathit{Proj}_P,\ \varnothing) = [\,] \ \vee\ \exists(SD_j \mid SD_j \in \{Sd_{P1}, Sd_{P2}\}\\ &\quad \mid \mathit{Filter}.(Sd_{C2},\ \mathit{Proj}_C,\ \mathit{Proj}_P,\ \varnothing) = \mathit{Rename}.(SD_j,\ Sd_{C2},\ \mathit{Proj}_C,\ \varnothing,\ \varnothing)) \end{aligned}$$

The sequence does include method calls of old interfaces, so filtering will not result in an empty sequence. Now we have two possible parent sequences that can be renamed, Sd_{P1} and Sd_{P2}. At least one of their renamings should be equal to the result of filtering Sd_{C2}.

$SD_j{=}Sd_{P1}$:

$$\begin{aligned}
&\mathit{Filter}.(Sd_{C2},\ \mathit{Proj}_C,\ \mathit{Proj}_P,\ \emptyset) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a}),\\
&\qquad \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), \mathit{return}.(r_{20}, r_{10}, x_{C2c})]\\
&\neq\\
&\mathit{Rename}.(SD_{P1},\ Sd_{C2},\ \mathit{Proj}_C,\ \emptyset,\ \emptyset) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a})\\
&\qquad \mathit{call}.(r_1, r_2, i_2, m_d, x_{P1b}), \mathit{return}.(r_2, r_1, x_{P1b})]
\end{aligned}$$

$SD_j{=}Sd_{P2}$:

$$\begin{aligned}
&\mathit{Filter}.(Sd_{C2},\ \mathit{Proj}_C,\ \mathit{Proj}_P,\ \emptyset) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a}),\\
&\qquad \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), \mathit{return}.(r_{20}, r_{10}, x_{C2c})]\\
&\neq\\
&\mathit{Rename}.(SD_{P2},\ Sd_{C2},\ \mathit{Proj}_C,\ \emptyset,\ \emptyset) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a}),\\
&\qquad \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), \mathit{return}.(r_{20}, r_{10}, x_{C2c})\\
&\qquad \mathit{call}.(r_1, r_2, i_2, m_c, x_{P2c}), \mathit{return}.(r_2, r_1, x_{P2c})]
\end{aligned}$$

We have derived incorrect inheritance, because the child sequence Sd_{C2} does not correspond to any of the parent sequences. The child is able to forget to call method m_c after calling (and returning) m_a and m_b, whereas the parent is not.

4.3.2 Fixing the mistake

With this example we try to derive correct inheritance, by fixing the mistake we made in the previous example. We now have the following child sequences:

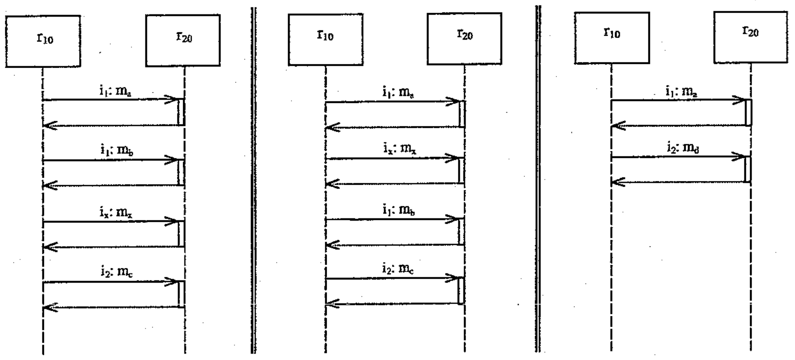


fig. 21: The child sequences, correctly inheriting the parent behaviour.

We've already seen both the leftmost and the rightmost sequence, we called them Sd_{C1} and Sd_{C3} respectively. The sequence in the middle is new, we'll call it Sd_{C4}.

$$\begin{aligned}
Sd_{C4} = [&\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), \mathit{return}.(r_{20}, r_{10}, x_{C2a}), \mathit{call}.(r_{10}, r_{20}, i_x, m_x, x_{C2b}),\\
&\mathit{return}.(r_{20}, r_{10}, x_{C2b}), \mathit{call}.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), \mathit{return}.(r_{20}, r_{10}, x_{C2c}),\\
&\mathit{call}.(r_{10}, r_{20}, i_2, m_c, x_{C1d}), \mathit{return}.(r_{20}, r_{10}, x_{C1d})]
\end{aligned}$$

Child behaviour is now represented by the set of sequence diagrams $\{Sd_{C1}, Sd_{C3}, Sd_{C4}\}$

$InhOfBeh.(Proj_P, \{Sd_{P1}, Sd_{P2}\}, Proj_C, \{Sd_{C1}, Sd_{C3}, Sd_{C4}\}) = true$

Sequence Sd_{C4} inherits (like Sd_{C1}) from parent sequence Sd_{P2}, since:

$$\begin{aligned}
&Rename.(SD_{P2},\ Sd_{C4},\ Proj_C,\ \emptyset,\ \emptyset) = [call.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), return.(r_{20}, r_{10}, x_{C2a}),\\
&\qquad call.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), return.(r_{20}, r_{10}, x_{C2c})\\
&\qquad call.(r_{10}, r_{20}, i_2, m_c, x_{C1d}), return.(r_{20}, r_{10}, x_{C1d})]\\
&=\\
&Filter.(Sd_{C2},\ Proj_C,\ Proj_P,\ \emptyset) = [call.(r_{10}, r_{20}, i_1, m_a, x_{C2a}), return.(r_{20}, r_{10}, x_{C2a}),\\
&\qquad call.(r_{10}, r_{20}, i_1, m_b, x_{C2c}), return.(r_{20}, r_{10}, x_{C2c})\\
&\qquad call.(r_{10}, r_{20}, i_2, m_c, x_{C1d}), return.(r_{20}, r_{10}, x_{C1d})]
\end{aligned}$$

Proving correct inheritance of Sd_{C1} is similar and that of Sd_{C3} is trivial (it mimics parent behaviour Sd_{P1}).
We've indeed derived correct inheritance.

4.3.3 Including a sequence with only new actions

In this test case our child behaviour includes a sequence that only contains new actions, i.e. calling and returning methods of interfaces that are new to the child only. This sequence doesn't mimic any parent sequence, but it doesn't "destroy" parent behaviour either. In our opinion, a sequence with only new actions will not lead to incorrect inheritance, while using *consistent inheritance* or *projection inheritance* would.
We have the following child sequences:

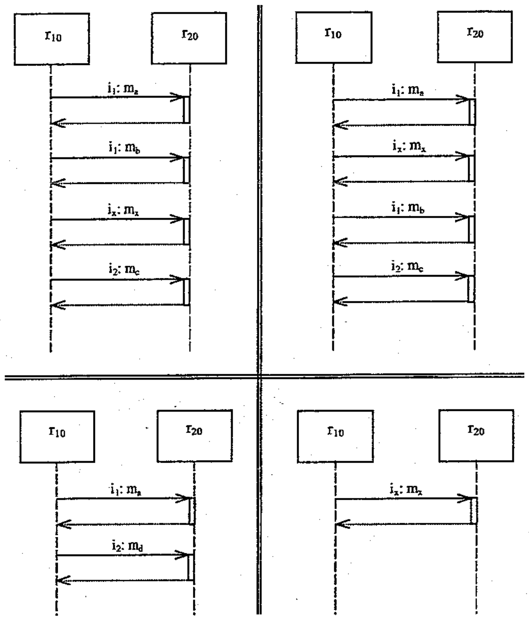


fig. 22: The child behaviour.

Child behaviour is now represented by the set of sequence diagrams $\{Sd_{C1}, Sd_{C4}, Sd_{C3}, Sd_{C5}\}$, where

$Sd_{C5} = [call.(r_{10}, r_{20}, i_x, m_x, x_{C5a}), return.(r_{20}, r_{10}, x_{C5a})]$

$InhOfBeh.(Proj_P, \{Sd_{P1}, Sd_{P2}\}, Proj_C, \{Sd_{C1}, Sd_{C3}, Sd_{C4}, Sd_{C5}\}) = true$

Sequences Sd_{C1}, Sd_{C3} and Sd_{C4} were proven to be correctly inheriting parent behaviour in the previous test case, correctness of Sd_{C5} remains. $Filter.(SD_{C5}, Proj_C, Proj_P, \emptyset) = []$, proves this.

4.3.4 Only inheriting a part of the parent behaviour

As we've mentioned in the beginning of this chapter, we do not require our child project to inherit the complete parent behaviour. Our parent project may for instance be a composition of a couple of projects, consisting of a large number of roles. Our child project only extends this project by a couple of roles that inherit from the parent project. This makes it impossible to inherit all parent behaviour.

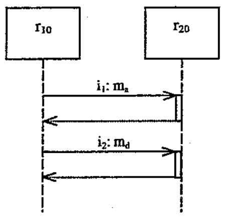


fig. 23: The child sequence.

This time the childs behaviour can be represented by the singleton set $\{Sd_{C3}\}$.

We have:

$$\begin{aligned}&\mathit{Filter}.(Sd_{C3},\ \mathit{Proj}_C,\ \mathit{Proj}_P,\ \varnothing) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C3a}), \mathit{return}.(r_{20}, r_{10}, x_{C3a}),\\ &\qquad \mathit{call}.(r_{10}, r_{20}, i_2, m_d, x_{C3b}), \mathit{return}.(r_{20}, r_{10}, x_{C3b})]\\ &=\\ &\mathit{Rename}.(SD_{P1},\ Sd_{C3},\ \mathit{Proj}_C,\ \varnothing,\ \varnothing) = [\mathit{call}.(r_{10}, r_{20}, i_1, m_a, x_{C3a}), \mathit{return}.(r_{20}, r_{10}, x_{C3a}),\\ &\qquad \mathit{call}.(r_{10}, r_{20}, i_2, m_d, x_{C3b}), \mathit{return}.(r_{20}, r_{10}, x_{C3b})]\end{aligned}$$

Which means that $\mathit{InhOfBeh}.(\mathit{Proj}_P, \{Sd_{P1}, Sd_{P2}\}, \mathit{Proj}_C, \{Sd_{C3}\}) = \mathit{true}$

4.4 A flexible representation

Our intuition has led to the definition of *sequence inheritance*, which allows a child to inherit only a part of the parent behaviour. Some of the other definitions demand the child to inherit all of the parents behaviour. We will show the flexibility of our formal representation by introducing function *TotalInhOfBeh*, that also restricts the child to inherit all behaviour.

$$\mathit{TotalInhOfBeh} \in \mathbb{B} \longleftarrow \mathbb{PROJ} \times \wp(\mathbb{SD}) \times \mathbb{PROJ} \times \wp(\mathbb{SD})$$

$$(\mathit{Proj}_{old},\ \mathit{SDs}_{old},\ \mathit{Proj}_{new},\ \mathit{SDs}_{new}) \in \mathit{TotalInhOfBeh}\rangle$$

$\equiv$

$$\begin{aligned}&\exists(\mathit{Suite}_{new} \mid \mathit{Suite}_{new} \in \mathbb{SUITE} \mid \mathit{Proj}_{new} = \mathit{ExtendProj}.(\mathit{Proj}_{old}, \mathit{Suite}_{new})\\ &\forall(SD_i \mid SD_i \in \mathit{SDs}_{old} \mid \mathit{CheckSdProj}.(SD_i, \mathit{Proj}_{old}) \wedge \mathit{ThreadSd}.Sd_i)\\ &\ \forall(SD_j \mid SD_j \in \mathit{SDs}_{new} \mid \mathit{CheckSdProj}.(SD_j, \mathit{Proj}_{new}) \wedge \mathit{ThreadSd}.Sd_j)\end{aligned}$$

$$\mathit{TotalInhOfBeh}.(\mathit{Proj}_{old}, \mathit{SDs}_{old}, \mathit{Proj}_{new}, \mathit{SDs}_{new})$$

$\equiv$

$$\begin{aligned}&\forall(SD_i \mid SD_i \in \mathit{SDs}_{new} \mid \mathit{Filter}.(SD_i, \mathit{Proj}_{new}, \mathit{Proj}_{old}, \varnothing) = [\,] \ \vee\ \exists(SD_j \mid SD_j \in \mathit{SDs}_{old}\\ &\quad \mid \mathit{Filter}.(SD_i, \mathit{Proj}_{new}, \mathit{Proj}_{old}, \varnothing) = \mathit{Rename}.(SD_j, SD_i, \mathit{Proj}_{new}, \varnothing, \varnothing)))\end{aligned}$$

$\wedge$

$$\begin{aligned}&\forall(SD_j \mid SD_j \in \mathit{SDs}_{old} \mid \exists(SD_i \mid SD_i \in \mathit{SDs}_{new}\\ &\quad \mid \mathit{Filter}.(SD_i, \mathit{Proj}_{new}, \mathit{Proj}_{old}, \varnothing) = \mathit{Rename}.(SD_j, SD_i, \mathit{Proj}_{new}, \varnothing, \varnothing)))\end{aligned}$$

TotalInhOfBeh is just an extension of the *InhOfBeh* function with the second conjunct of the definition. We also want all parent sequences (old) to have at least one child sequence (new) inheriting its behaviour.

Chapter 5

Using a tool

5.1 Introduction

In this chapter we perform the same test cases as we did in the previous chapter. We're going to use a software tool that is still under development. This tool is built as a plugin for Rational Rose, sharing the same goal we have: constructing ISpec interface role and sequence diagrams and verify inheritance.
We'll shortly describe the representation of interface role diagrams and sequence diargams that are the basis of this tool, as well as the intended effect. For more information the reader is referred to [2] and [3].

An interface role diagram is a graph with two kinds of nodes and three kinds of relations:
$IR = (R, I, PI, RI, RR)$
- *Nodes:*
• R is a set of roles. Each role $r \in R$ has a set of players PL_r.
• I is a set of interfaces. Each interface $i \in I$ has a set of results Res_i of the inteface.
Relations:
• $PI = \{(r,i) \mid r \in R,\ i \in I\}$, provided interfaces.
• $RI = \{(r',(r,i)) \mid r', r \in R,\ i \in I,\ (r,i) \in PI\}$, required interfaces.
• $RR = \{(r,r') \mid r, r' \in R\}$, inheritance relation between roles.

A sequence diagram is a tuple:
$s = (R \times PL, A_s)$:
• $R \times PL$ is a set of players of roles. A player of a role is represented by a box with a line drawn from the box.
• $A_s = \{(rp, n, (v,w,l)) \mid rp = \{\omega, st_i, f_i\},\ n = 1,2,...,N\ ,\ (v,w,l) \in (R \times PL) \times (R \times PL) \times (I \times Res)\}$
– rp is a repetition symbol, used for repeated subsequences of actions. st_i and f_i denote the beginning and end of a repeated subsequence i.
– n is used to distinguish several instances of identical actions.

– (v, w, l) corresponds to elements of the required interface set RI from the interface role diagram.

We can see that this representation differs from the one we use. Main differences are the use of instances of roles, i.e. players, and the use of one interface as a method having several possible return values, instead of using an interface to be a collection of methods.

The basic idea behind developing this tool was to automatically verify inheritance of behaviour, comparing one set of sequence diagrams to another. Some theory on this subject had already been developed, studying inheritance of behaviour using process algebra.
To verify inheritance of sets of sequence diagrams the tool essentially had to do two things:
- Construct process algebra terms from sets of sequences diagrams (the parents and the childs). An algorithm was developed for this, described in [3].
- Use the definition of life-cycle inheritance (from [4]), based on process algebra terms, to verify inheritance.

Developing the tool it became clear, that implementing life-cycle inheritance was not straightforward. Some set of actions has to be blocked, while some (other) set of actions has to be hidden, but it remains unclear which actions belong to which set. Furthermore, it was also the developers intuition, that a child had to be able to inherit only a part of the parent behaviour, instead of being equal (after blocking and hiding).
Instead of having the option to automatically verify inheritance, the developer created the option to hide and block actions manually, as well as the ability to hide and block (uninherited) parent actions.

5.2 Testing

We'll go through the same tests we did in de previous chapter, using the tool described above. A screenshot of the parent interface role diagram is shown below.

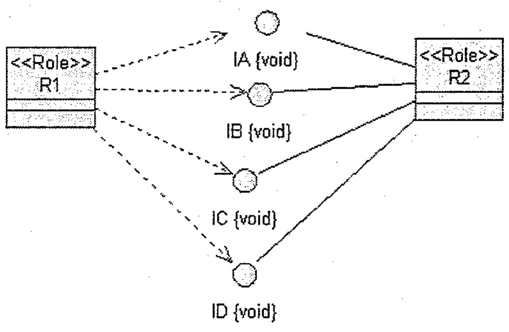


fig. 24: The parent interface role diagram.

As we can see, this interface role diagram is slightly different from the one we used in the previous chapter. This model does not allow to have several methods belonging to one interface, but instead one interface *is* one method, and its parameters are the possible return-values. We are not interested in the value of returns, only in the order of the subsequent call- and return-actions. Therefore we leave all return-parameters empty, *void.*
Nevertheless, we can reach the same behavioural pattern as we did in the previous chapter, illustrated by the sequence diagrams of the parent below.

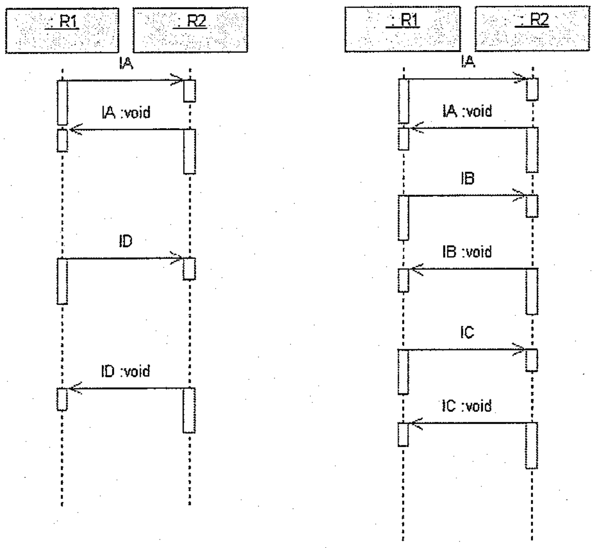


fig. 25: The parent sequences

The tool will convert this set of sequence diagrams of the parent into one, so called, process tree, renaming the actions to small identifiers.

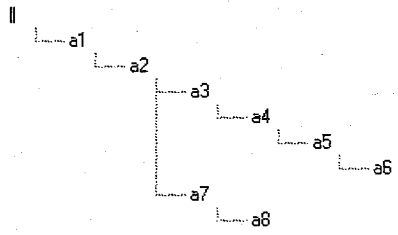


fig. 26: The Process tree of the parent.

with:
$a1 = IA,\ or\ call.A$
$a2 = IA\ {:}void,\ or\ return.A$
$a3 = call.B$
$a4 = return.B$
$a5 = call.C$
$a6 = return.C$
$a7 = call.D$
$a8 = return.D$

Because both sequences start with the same two actions, *call.A* and *return.A*, the tool will unite the first two actions of both sequences, postponing the moment of choice (splitting the branches of the tree) after these two actions.

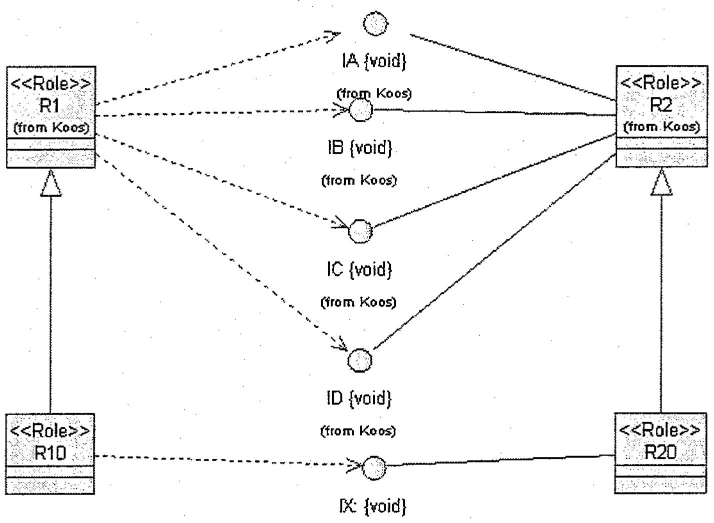


fig. 27: The child interface role diagram, after specialisation.

The parent interface role diagram was called "Koos". We can see that we extended this interface role diagram in the same way we did in the previous chapter, introducing one additional interface (and method) X. We will go over the test cases again, discovering the notion of "inheritance of behaviour" this tool uses.

5.2.1 Deliberately making a mistake

Below we give the three child sequences and the accompanying process tree.

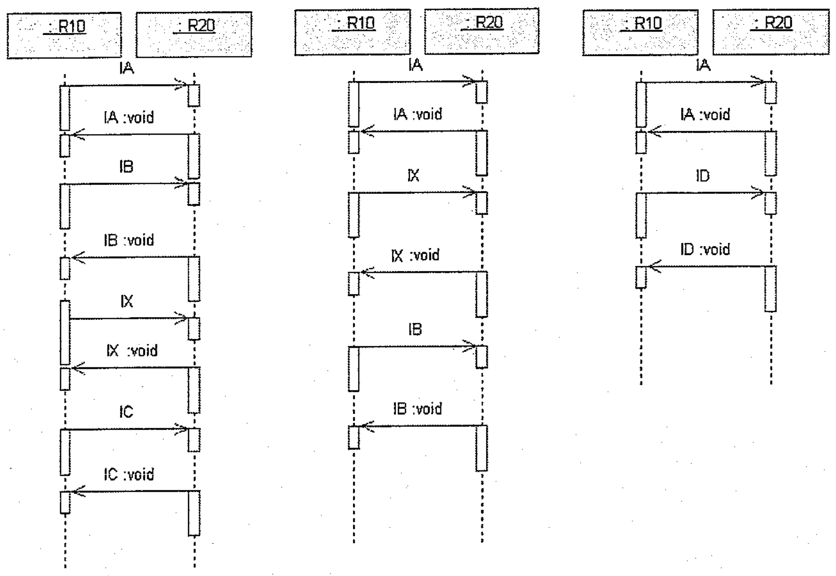


fig. 28: The child sequences

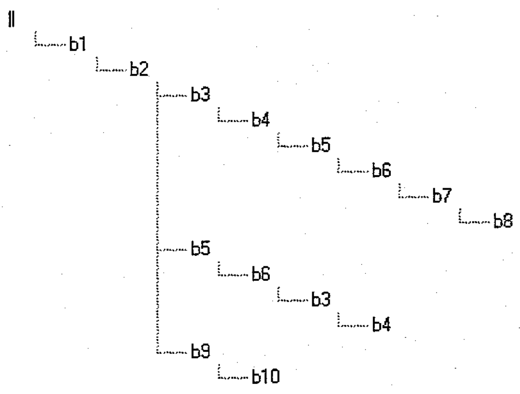
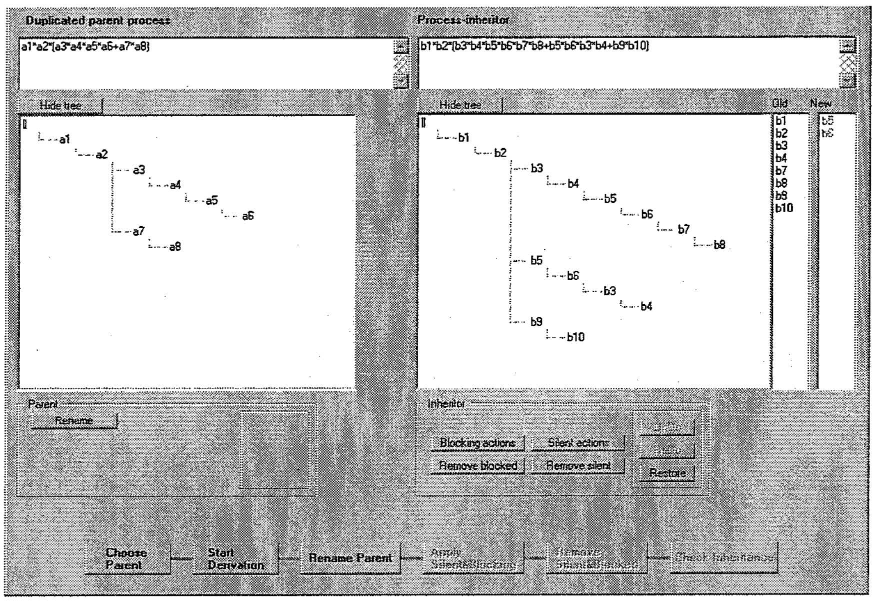


fig. 29: The Process tree of the child.

The following name substitutions took place:
call.A, *return.A* := *b*1, *b*2
call.B, *return.B* := *b*3, *b*4
call.X, *return.X* := *b*5, *b*6
call.C, *return.C* := *b*7, *b*8
call.D, *return.D* := *b*9, *b*10

On the basis of both the parent process tree and the child process tree, the tool is able to verify inheritance in a couple of steps. We show and explain these steps by screenshots.

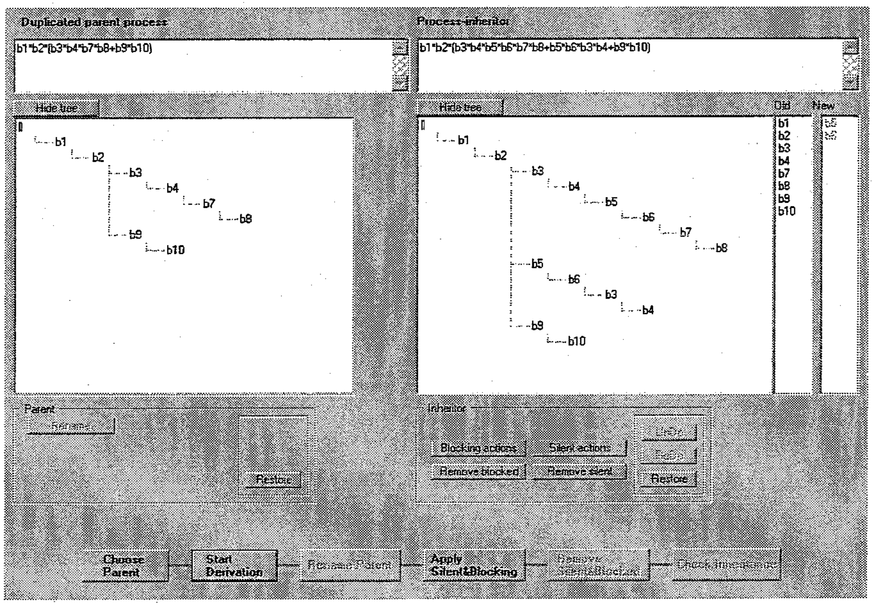


```
a1*a2*(a3*a4*a5*a6+a7*a8)
```

```
b1*b2*(b3*b4*b5*b6*b7*b8+b5*b6*b3*b4+b9*b10)
```

fig. 30: Tool status at the beginning of a proof.

The picture above shows the process tree of the parent on the left and that of the child on the right (called Process-inheritor). We can also see that the tool "knows", from the interface role diagrams and sequence diagrams, that the actions concerning new method X ($b5$ and $b6$ are indeed new actions.
The first step of verifying inheritance is renaming parent actions to comply with child actions. This is comparable to the renaming function we introduced in our formal approach.

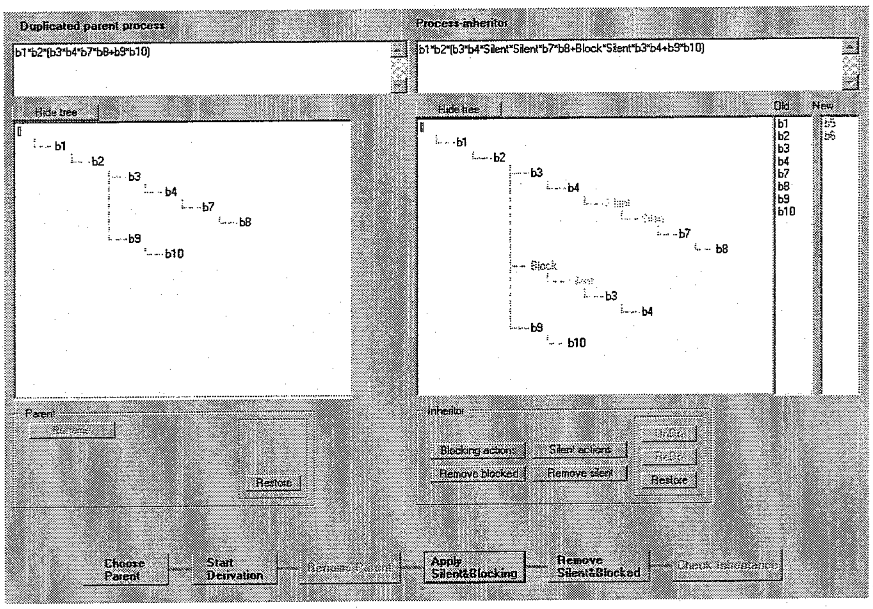


```
b1*b2*(b3*b4*b7*b8+b9*b10)
```

```
b1*b2*(b3*b4*b5*b6*b7*b8+b5*b6*b3*b4+b9*b10)
```

fig. 31: The view after renaming actions.

Parent actions are correctly renamed.
The next step is the blocking and hiding of new (child) actions. The next screenshot illustrates the result.

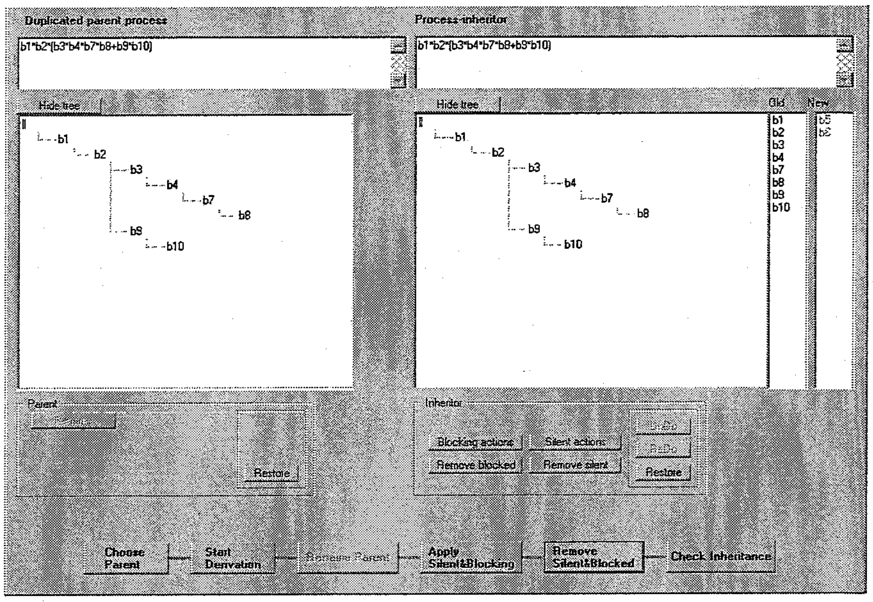


```
b1*b2*(b3*b4*b7*b8+b9*b10)
```

```
b1*b2*(b3*b4*Silent*Silent*b7*b8+Block*Silent*b3*b4+b9*b10)
```

fig. 32: Encapsulating (blocking) and Projecting (making silent) actions.

We can see (if it's still readable) that action *b*6 was made silent twice, whereas action *b*5 was made blocked once and made silent once. Encapsulation (blocking) of *b*5 takes place, because it is the first action of a subsequence. The other instance of action *b*5 has preceding actions in its subsequence, and is therefore hidden.
A block action at the beginning of a branch (of the process tree) will lead to blocking the entire branch. An entirely blocked branch can be left out of the tree. Silent actions that are preceded by another action in the same branch can also be left out. The result of this can be seen in the following picture.


Duplicated parent process

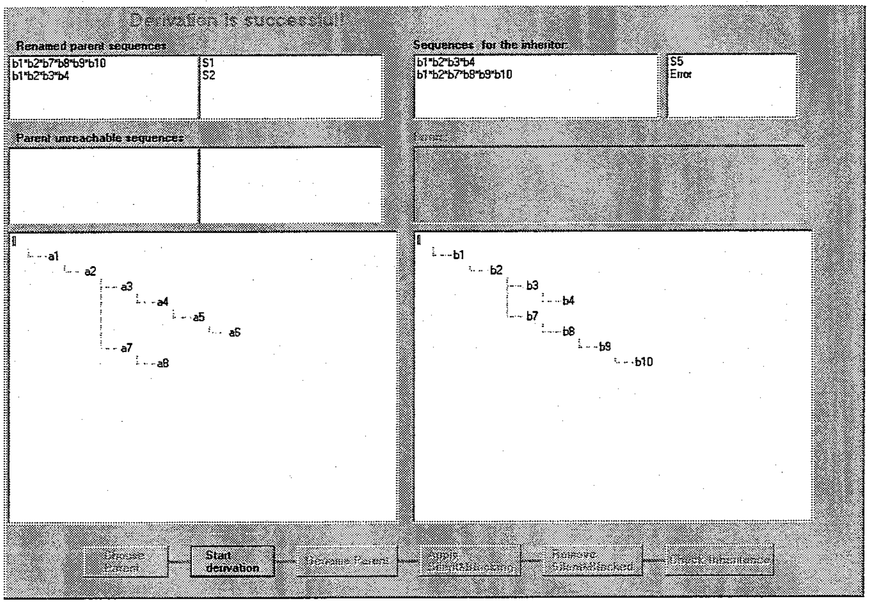
b1*b2*(b3*b4*b7*b8+b9*b10)

Process inheritor

b1*b2*(b3*b4*b7*b8+b9*b10)

fig. 33: The result after projection and encapsulation.

We now have two identical process trees. The following (and final) step will be the "check inheritance" step, which we were actually doing all along.


Derivation is successful!

Renamed parent sequences

b1*b2*b7*b8*b9*b10	S1
b1*b2*b3*b4	S2

Sequences for the inheritor

b1*b2*b3*b4	S5
b1*b2*b7*b8*b9*b10	Error

fig. 34: The result after checking inheritance.

As we can see at the top of the screenshot, our inheritance derivation is successful. The same example in the previous chapter resulted differently.
Although there are different notions of inheritance of behaviour, we think this is

a mistake, not just another point of view. We have the following argumentation for this:
The sequence where we omitted, or forgot, the call (and return) of method C was the one that got encapsulated, because of the presence of a new action ($call.X$). We could have any sequence of actions following this new action, since the entire branch of the tree is always encapsulated. Another sequence also contained this same new action, but there it was hidden instead of encapsulated. The reason for this is the decision to postpone the moment of choice when constructing the process tree (process algebra term). All three child sequences start with the same two actions and therefore the process tree starts with these two actions, before splitting into branches (moment of choice). That made the difference of the new-action being at the beginning of a branche once, while having preceding old-actions on the other occasion.

5.2.2 Fixing the mistake

We include the call and return of interface C in the sequence we believed was wrong. The behaviour of the child is now formed by the following sequence diagrams:

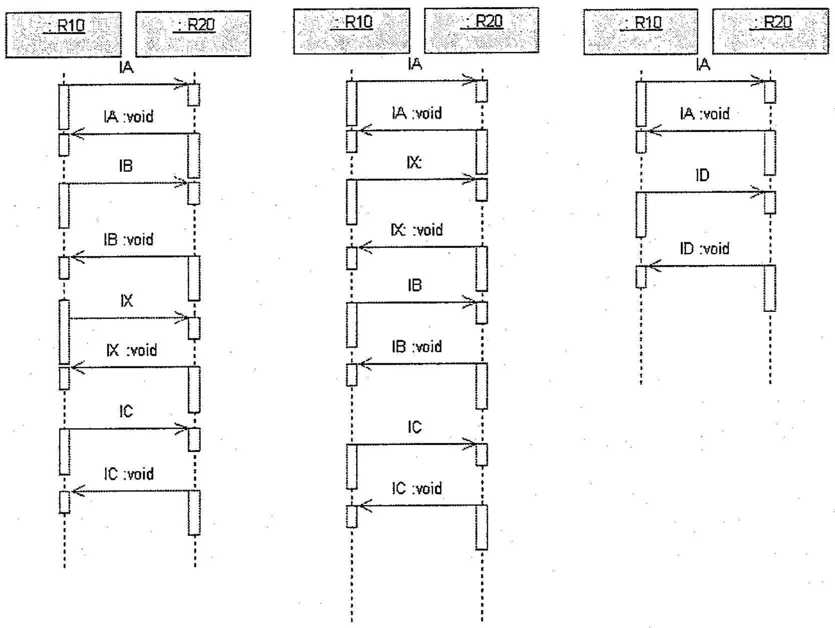


fig. 35: The new child sequences.

This leads, using algorithm A (Appendix A of [3]), to the following process tree:

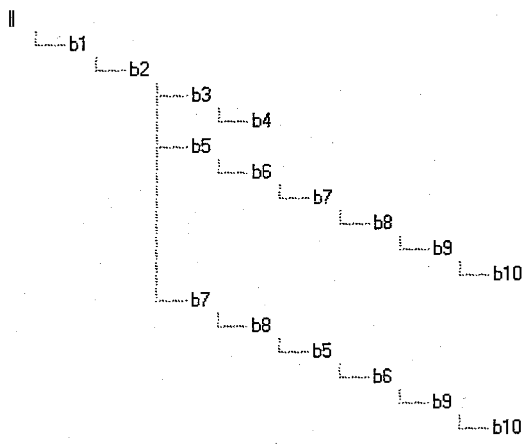
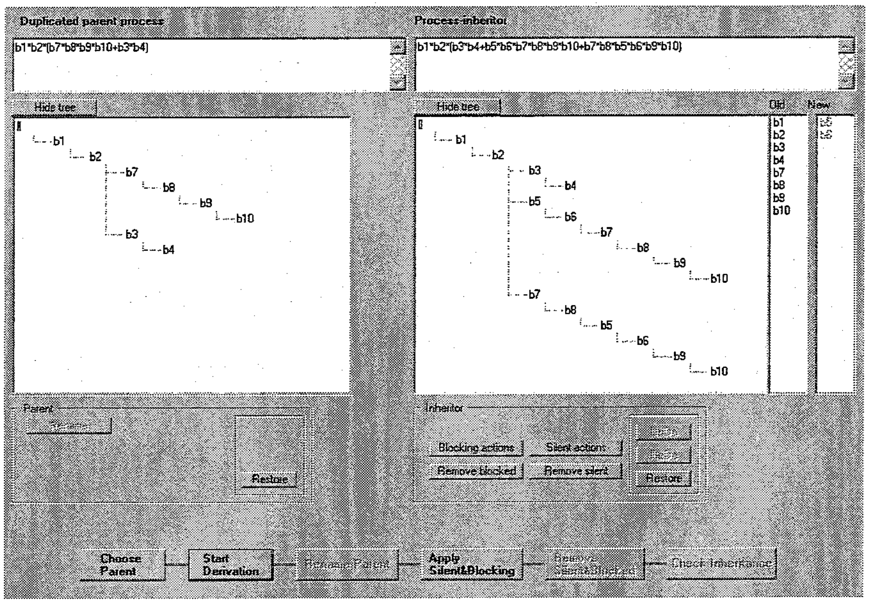


fig. 36: The new process tree of the child.

We use the tool to proof inheritance again, starting with renaming the parent actions.


Duplicated parent process

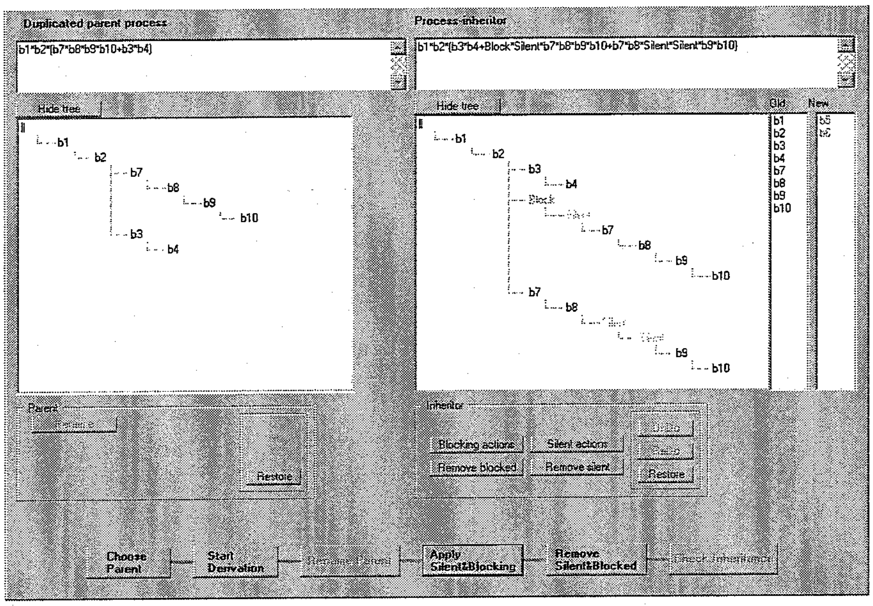
b1*b2*(b7*b8*b9*b10+b3*b4)

Process inheritor

b1*b2*(b3*b4+b5*b6*b7*b8*b9*b10+b7*b8*b5*b6*b9*b10)

fig. 37: The result after renaming actions.

The new actions that are called $b5$ and $b6$, are representing the call and return of interface X. We can see that again, not surprising, one of the two instances of $b5$ is at the start of a subsequence. Encapsulation and abstraction of new actions ("apply silent & blocking"), leads to:


Duplicated parent process

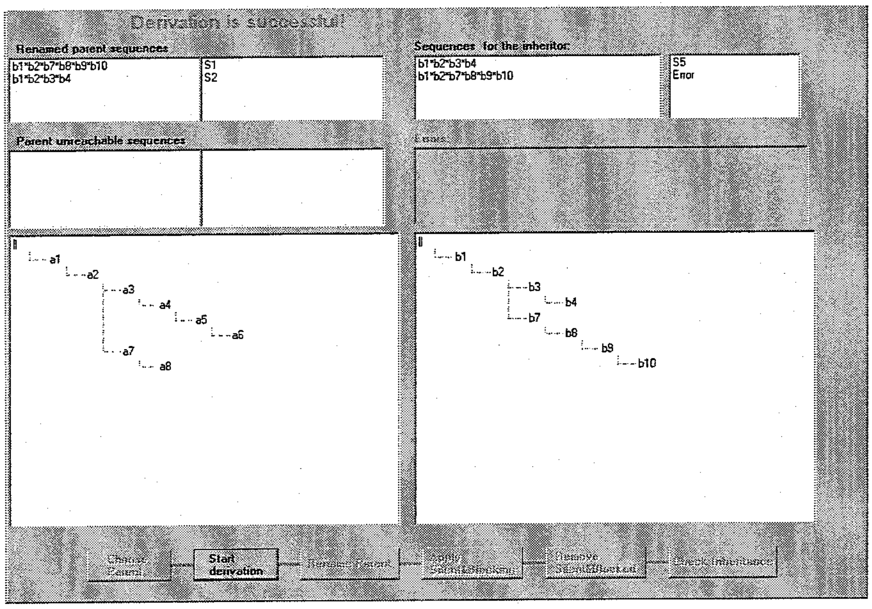
b1*b2*(b7*b8*b9*b10+b3*b4)

Process-inheritor

b1*b2*(b3*b4+Block*Silent*b7*b8*b9*b10+b7*b8*Silent*Silent*b9*b10)

fig. 38: Blocking and silencing actions.

The included actions (calling and returning C) that were ment to fix the mistake of the previous test, being $b9$ and $b10$, now also follow the blocked action ($b5$). Next step is checking inheritance.


Derivation is successful!

Renamed parent sequences

b1*b2*b7*b8*b9*b10	S1
b1*b2*b3*b4	S2

Sequences for the inheritor

b1*b2*b3*b4	S5
b1*b2*b7*b8*b9*b10	Error

fig. 39: The result after checking inheritance.

We've derived successful inheritance again, but not because of fixing the mistake. It's because of the first (leftmost in figure 35) sequence that we are successful here. However, if we would have omitted this sequence, setting the childs behaviour to the other two sequence diagrams, we would *not* derive successful inheritance, where it should have been the case. Unfortunately, we have no screenshots of this test case available.

5.2.3 Including a sequence with only new actions

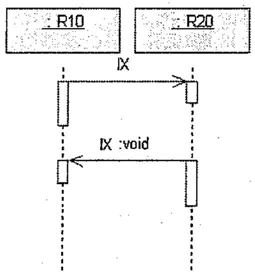


fig. 40: A sequence with only new actions.

This sequence is added to the child sequences of the previous test.
This will result in a process tree expanding figure 36 with an additional branch from the root, consisting of two actions that represent the call and return of interface X.
After blocking and silencing the actions, this process tree will be the one in the figure below:

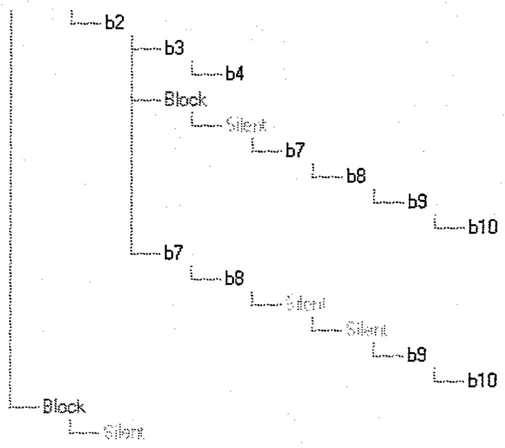


fig. 41: Process tree including blocked and silenced actions.

The sequence containing only new actions is blocked out, with a positive inheritance derivation as a result. Although this result is correct for a sequence with only new actions, we can see again that any sequence starting with a new action, possibly followed by all sorts of inherited actions in any order, will always be blocked. Inheritance of such a sequence will not be verified.

5.2.4 Only inheriting a part of the parent behaviour

When we are specialising parent behaviour we can select to inherit only a part of the functionality and therefore only a part of the accompanying behaviour of our parent role(s). We will test this scenario here with again a very simple example. We will only inherit one of the sequences of our parent in this test case.

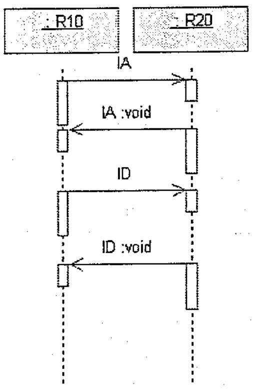
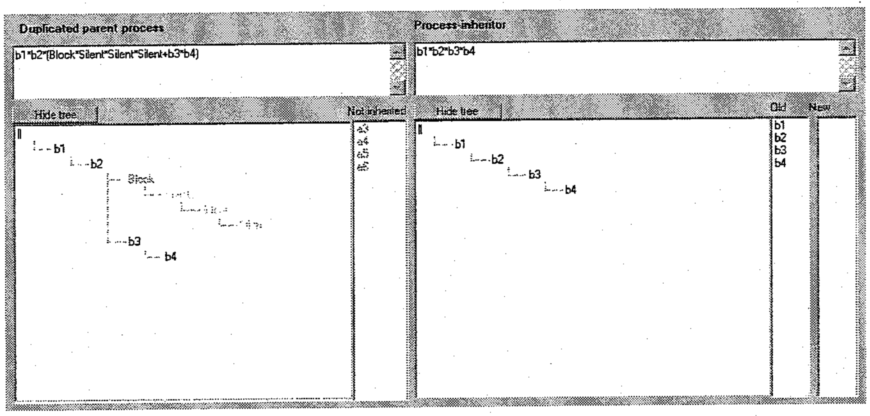


fig. 42: Child behaviour consisting of only one sequence.



b1*b2*(Block*Silent*Silent*Silent+b3*b4)

b1*b2*b3*b4

```
||
 '--b1
     '--b2
         |-- Block
         |
         '--b3
             '-- b4
```

```
||
 '--b1
     '--b2
         '--b3
             '--b4
```

b1
b2
b3
b4

fig. 43: Process tree including blocked and silenced actions.

We can see that the tool now also blocks and silences actions of our parent process tree, namely those actions that are not inherited. At first glance this seems like the right thing to do if we want the tool to support the possibility to inherit only part of the parent behaviour. We have to note that this functionality of the tool is only at the beginning of its development. It also does not comply with the notion of life-cycle inheritance we saw in [[**]].

5.2.5 Testing one sequence at a time

The previous example showed us that we can check inheritance of our child sequences individually. When we have a set of sequences and we test them all individually, we should derive correct inheritance with all of them to make sure we have correct inheritance of the set. This way we can use the tool to verify sequences without having any branches in our process trees. Therefore we have no new actions at the start of branches that become blocked. Sequences starting with a new action (the very first action of the sequence) will still encounter this blocking-problem.
We now use this method of testing on our first test, and try to capture the mistake we made.
We start by checking the second correct sequence:

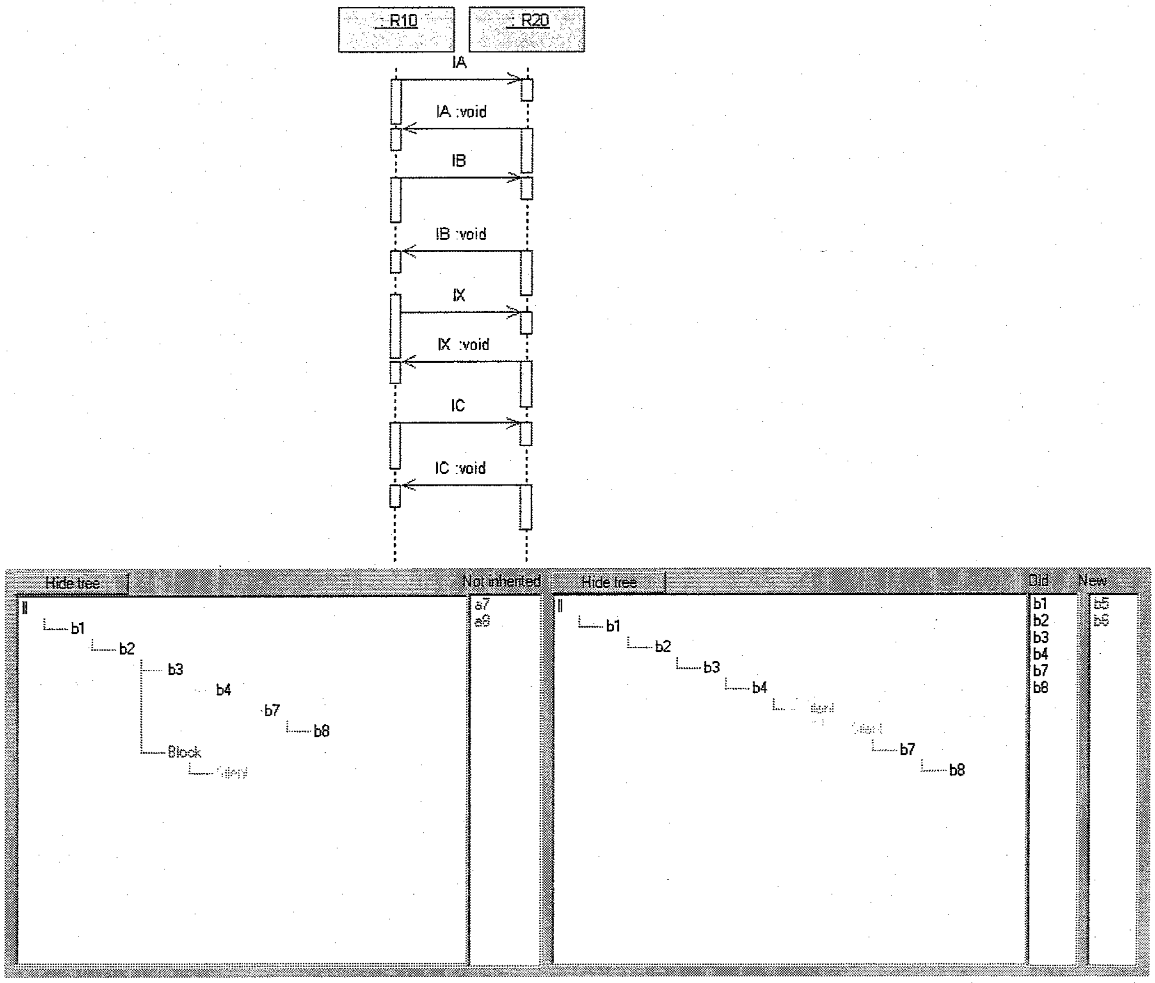


fig. 44: The second correct sequence and its resulting process tree.

The child leaves out the actions involving new interface X, and the parent disregards sequence [*call.A*, *return.A*, *call.D*, *return.D*], which was not inherited.

So far so good.
The third (and final) sequence we have to verify is the one that did not include the call- and return-actions of inherited interface C. This is a mistake in our point of view and should not be correctly inheriting a parent sequence that does include these actions.

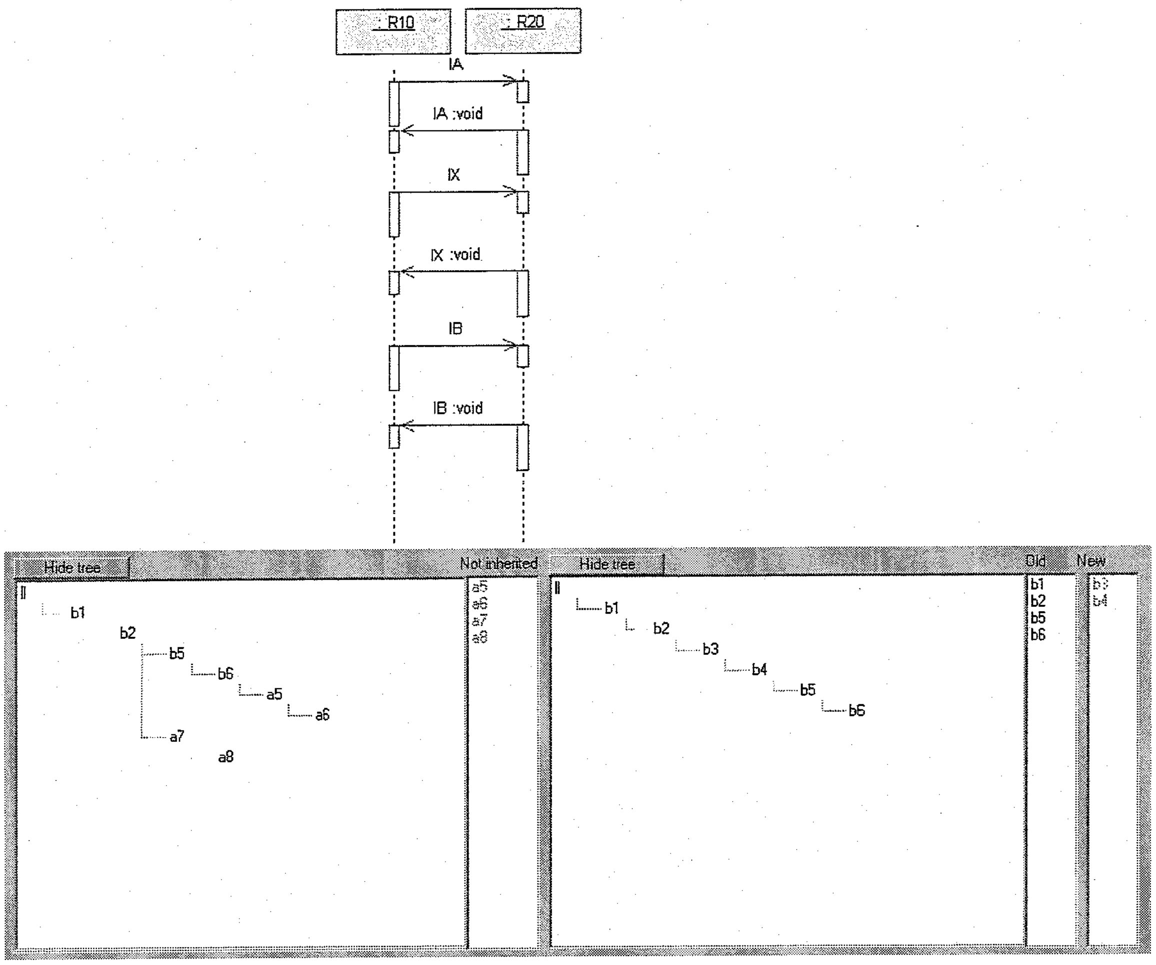


fig. 45: The erroneous sequence and its resulting process tree.

We can see that $b3$ and $b4$ represent *call.X* and *return.X* respectively. $a5, a6$ represents interface C and $a7, a8$ represents interface D. The omission of an action in a child sequence will lead to the projection of that action in our parent process tree, because the tool "thinks" its an uninherited action. Again we do not agree with this.
In our opinion a child is allowed to inherit part of the parent behaviour by not inheriting all of its parent sequences. We do not allow a child to inherit only part of one particular parent sequence. We feel that would be too unrestrictive.

5.3 Concluding

We can conclude that intuition plays a big role, when discussing inheritance of behaviour. There are different notions of inheritance, as there are different intuitions. Creating a tool to automatically verify inheritance can only adress one notion of inheritance at a time.
It will be very useful to implement different notions of inheritance in one single tool. We could for instance implement both projection inheritance and protocol inheritance quite easily: hiding all new actions, blocking none (projection) or blocking all new actions, hiding none (protocol).
We could also try to implement a different approach on constructing the process trees from a set of sequence diagrams. We believe that, since we cannot indicate moments of choice in sequence diagrams, we should move this moment of choice back to the root of the process tree.

Chapter 6

Conclusion

When we started working on this thesis our main goal was to investigate and formalise inheritance of behaviour using sequence diagrams. To formalise these sequence diagrams, that describe the dynamic part of an ISpec model, we first need a decent formal representation of the static part of the model, described by an interface role diagram. We introduce sets and relations representing the suites, roles, interfaces and methods of an interface role diagram. We use these sets and relations to formally define the restrictions an interface role diagram has to meet. We also introduce formal notions of both composition and extension of our interface role diagrams to be able to assess inheritance on the structural level. Behavioural inheritance needs to be consistent with the inheritance defined on the structural level.

One sequence diagram describes one possible behavioural pattern of a model. The behaviour described by a sequence diagram has to be conform the structure of the model, which means that the right methods are called on the right interfaces of the right roles.
A set of sequence diagrams describes several possible behavioural patterns and therefore a (part of) the behaviour of a model. We could even describe the total behaviour of a model by a set of sequence diagrams. It is however not realistic to presume this, since even a simple model can have a very large, possibly infinite, set of sequence diagrams.

To investigate behavioural inheritance we need to compare a set of sequence diagrams of one model, which we call the parent model, to a set of sequence diagrams of another model, the child model. The child model has to inherit correctly from the parent model using our definition of structural inheritance and all sequence diagrams have to be conform their accompanying model.
Furthermore we need a notion of inheritance of behaviour to be able to formalise our demands. Several notions already exist in scientific literature, from which we deducted our notion of inheritance of behaviour. Intuition plays a role here, but we also showed that our representation has the flexibility to formalise other notions of inheritance as well.

We showed some examples to clarify our notion of inheritance and we used the same examples on a tool under development that uses a different notion of inheritance. From this we can conclude that automatically verifying inheritance can only adress one notion of inheritance at a time. Having different notions of inheritance could be the basis of constructing a tool that will be able to verify these different notions of inheritance seperately. Then we would be able to compare the different notions of inheritance more easily to guide our intuitions to possibly defining one, definitive, notion of inheritance of behaviour.

Bibliography

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

[2] E.E. Roubtsova, W.M.P. van der Aalst, S.A. Roubtsov & R. Kuiper. *Inheritance of UML Behavioral Specification in Component System Development.*

[3] E.E. Roubtsova & R. Kuiper. *Process semantics for UML component specifications to assess inheritance.* Proceedings of the International Workshop on Graph Transformation and Visual Modeling Techniques, October 2002.

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

[5] H.B.M. Jonkers. *Towards Practical and Sound Interface Specifications.* Integrated Formal Methods 2000, pp. 116-135.

[6] E.e.Roubtsova, L.C.M. van Gool, R.Kuiper & H.B.M. Jonkers. *A Specification Model for Interface Suites.* In Proceedings of 4th International Conference on the Unified Modeling Language,UML'01, "Modeling Languages, Concepts and Tools", LNCS 2185, Springer Verlag, 2001, Toronto, Canada, pp.457-471.

[7] M. Schrefl & M. Stumptner. *Behavior-Consistent Specialization of Object Life Cycles.* Transactions on Software Engineering and Methodology (TOSEM), Volume 11, 2002, pp. 92-148.

[8] E.E. Roubtsova, R. Kuiper & H.B.M. Jonkers. *Interface Suites as Contracts. Composition of Contracts in UML.* Proceedings of 3rd Workshop on Embedded Systems, PROGRESS 2002. October 2002, The Netherlands. STW, pp.203-210.

[9] G. Saake, P. Hartel, R. Jungklaus, R. Wieringa & R. Feenstra. *Inheritance Conditions for Object Life Cycle Diagrams.* EMISA Workshop, 1994.

[10] T. Basten & W.M.P. van der Aalst. *A process-Algebraic Approach to Life-Cycle Inheritance = Encapsulation and Abstraction.* Computing Science Reports 96/05, Eindhoven University of Technology, Eindhoven, 1996.

[11] G. Kappel & M. Schrefl. *Inheritance of Object Behavior - Consistent Extension of Object Life Cycles.* East/West Database Workshop 1994: pp. 289-300.

[12] E.E. Roubtsova & S.A. Roubtsov. *UML-base Tool for Constructing Component Systems via Component Behaviour Inheritance.* Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03),Trondheim, Norway, 4-7 June 2003, pages 139-154. To appear in Elsevier Electronic Notes in Theoretical Computer Science, 80 (2003).

[13] J.C.M. Baeten & W.P. Weijland. *Process Algebra.* Cambridge University Press.

[14] P. Stevens & R Pooley. *Using UML. Software Engineering with Objects and Components.* Addison-Wesley, an imprint of Pearson Education.