Behavioural inheritance in the UML to model software product lines

Ella Roubtsova\textsuperscript{a,\textdagger}, Serguei Roubtsov\textsuperscript{b}

\textsuperscript{a}Eindhoven University of Technology, Den Dolech 2, P.O. Box 513, 5600 MB, The Netherlands
\textsuperscript{b}VTT Electronics, Kaitovayla 1, P.O. Box 1100, FIN-90571 Oulu, Finland

Received 12 January 2003; received in revised form 10 April 2003; accepted 24 April 2003

Abstract

Traditional product line approaches struggle with complexity and weak evolution support. We propose an evolutionary software product line modelling approach based on controllable inheritance of product line members specifications. Instead of a predefined product line architecture we use hierarchies of implemented product specifications plus correctness control of product model transformations. The approach is supported by an appropriate tool prototype.

Keywords: Software product line; Architectural design; Behavioural inheritance; UML

1. Introduction

The concept of a Software Product Line (SPL) is a concept of software reuse in industry covering the design process and the implementation technology [4].

Software product lines traditionally employ a top-down architecture-based methodology of software system development [4,8–10,19]. It starts by choosing a set of products comprising a product line and then proceeds by identifying what requirements are common to all products (commonalities) and what product features make them different (variations). On the basis of requirement analysis, a common product line architecture and a

\textdagger\ Corresponding author.
\textit{E-mail addresses:} E.Roubtsova@tue.nl (E. Roubtsova), S.Roubtsov@tue.nl (S. Roubtsov).
set of reusable components are designed and implemented. Finally, actual products are derived from these shared assets [4]. Commonalities between SPL members are captured by a generic architecture. Variabilities are usually introduced into this architecture by means of so-called variation points [6], which imply unresolved diversity in the generic and component architectures that should be explicitly introduced and bound into a concrete product during possibly latest phases of product line members development [6].

So, a common SPL architecture with variability management fulfils a double role. Firstly, it provides the reference of integrity for SPL component reuse. Secondly, the diversity of all product line members, existent or future, should correspond to the variability already implicit in such a generic architecture. The SPL architecture should provide correctness of product modifications.

However, there are some disadvantages of such an architecture-driven [31] approach. The first problem is complexity. The entire development process is divided into two concurrent parts—domain engineering for reusable SPL assets and application engineering for product line members [19]. SPL development and maintenance give rise to a lot of related tasks, which have to be solved coherently [4,8]. Among others design of a reusable architecture is an especially complicated problem. How much commonality and variability should be introduced into a common SPL architecture? It has to be somewhat between minimal reuse (common requirements only) and maximal reuse (all requirements, both common and different). The more variability is introduced into the architecture, the more benefits of reuse should be expected. However, design of such a flexible architecture meets a truly challenge [3,4,10].

The second problem is evolution support [27]. Requirements are changed, technology is improved. How can we predict the features and, therefore, the architectures of future product line members? Even architecture itself suffers from erosion during a software product evolution process. Research [29] shows how seemingly robust design decisions taken early in the evolution of a single product may conflict with requirements that need to be implemented later in the evolution. For product lines the problem increases immensely (e.g., [12]).

The impact of the above mentioned problems is the high cost of wrong architectural design decisions.

The alternative software reuse approach is an evolutionary component-based software development process [28]. In the SPL domain it is a product population approach [5,30–32]. That approach uses lightweight [30] common architecture and implements software component modifications and component compositions instead of architecture-based variability management (e.g., [32]).

The benefits of evolutionary approaches are explicit. An SPL grows when new product line members appear. A design process is incremental. Similar already implemented products are reused to introduce the extensions which are required by a new product. However, in the absence of a fixed common architecture the problems of SPL integrity and product design correctness rise sharply. Component modification and composition rules are static; they do not guarantee that the entire system behaviour comprises the behaviour of composed parts in a correct manner. The evolutionary approach needs a design methodology that can help designers collect useful features of already implemented SPL members and avoid incorrect design decisions while they introduce new product
functionality. In addition, SPLs are rather long-lived software projects and need to be supported not only by a reusable component set but also by some joint model to be a reference of integrity.

In order to overcome outlined challenges we propose an evolutionary software product line modelling method based on the inheritance of product specifications and correctness control of model transformations. Each implemented specification can become a predecessor of a new product specification. At the same time, correctness of behavioural inheritance with new extensions has to be checked.

In our approach design specifications are implemented using UML (Unified Modeling Language) profile with defined inheritance relations on specifications [22]. The profile defines a special type of UML class diagrams, interface-role diagrams, similar to CATALYSIS [11] and ISpec approaches [18]. Component system behaviour is specified in the profile using UML sequence diagrams as first introduced in [7]. Process semantics is used as a basis for inheritance relations on component behavioural specifications [2,22].

Correctness control is provided by product model transformation checks using inheritance of processes. Applying of backward derivation rules to produce parent product process specifications from the inheritor’s ones allows a designer to prove correctness of inheritance or to find the points of wrong design decisions.

In [25] the evolutionary SPL modelling technique has been used within the traditional architecture-centric SPL development process. Now we advocate our modelling method as a self-sufficient and robust alternative to the traditional one.

The remainder of the paper is the following. Section 2 introduces our design method for specification of products of software product lines. Section 3 gives the notion of inheritance of product specifications and provides corresponding illustrations using a simple case study. Section 4 describes the tool prototype which has been developed to support our method. The issues of complexity of underlying algorithms are also discussed in this section. Section 5 shows the place of our method in the software product line development. Section 6 concludes the paper and discusses yet unresolved problems.

2. A design method

To design products of software product lines we use a special kind of role approach, interface-role modelling [18,22,24]. The interface-modelling approach introduces an interface suite, which is represented by a finite set of roles communicating via interfaces provided by these roles.

First, we consider interface suites as product requirements models. An interface suite is specified in a UML profile which contains an interface-role diagram and sequence diagrams. This form of specification in terms of roles and interfaces allows us to collect requirements from customers and represents desired features of products.

Second, roles and interfaces in the interface-role approach can be seen as abstractions from different components [24] as well as the interface suite itself representing a software component system. So, a product model in the form of an interface suite is related to the standard component architecture model [28].
Third, our UML profile has a process semantics and an inheritance relations defined on specifications. We use the inheritance of interface suites [22] as an instrument of the evolution of SPL products.

2.1. Interface-role specification of SPL products

We specify SPL products as interface suites (IS). An interface suite is a set of roles communicating via interfaces [24]. Roles and interfaces are abstractions both from desired product features and from the implementation.

Roles can provide interfaces, which other roles can require [11]. Each such a pair of roles interacting via the interface can model a piece of product functionality, i.e. a product feature [4]. So, product functional requirements can be mapped directly to interface-role specifications [25].

On the other hand, roles with interfaces are quite similar in nature to product components. Components interact by playing roles. A designer is free to abstract from a concrete component implementation choosing roles that are closer to requirements [33] than software components. At the same time, one or several interacting roles can be mapped onto a product component architecture in such a way that component boundaries should come across the interfaces provided by roles [25,33].

To show the application of the role approach, let us consider a simple software product line Graph Designer.

The first product of this software product line accepts data series for constructing a graph from a user. The user chooses the graph properties and starts drawing. Three roles, User, Graph Maker and Graph Drawer (boxes in Fig. 1), interacting between each other via three interfaces (circles in Fig. 1), represent product functional requirements. The role to which a dashed arrow in Fig. 1 is directed provides the interface that the role from the other side of the arrow requires.

The second product of SPL Graph Designer can take data series both from a user and from a database (Fig. 2).

The third product draws a real-time graph periodically updating data series from a database (Fig. 3). A user starts and stops drawing. Roles and interfaces as well as relations between them are described in our UML profile.

2.2. Definition of the UML profile

We specify a product of a software product line in our UML profile as an interface suite

\[ IS = \{IR, S_1, \ldots, S_k\}, \]

which contains

– interface-role diagram \( IR \);
– set of sequence diagrams \( \{S_1, \ldots, S_k\} \);
– a process semantics applied to the combination of those diagrams.

Let us consider these three parts consecutively.
2.2.1. Interface-role diagram

Interface-role diagram $\mathcal{IR}$ is presented in our profile by a UML class diagram where roles are represented by classes with stereotype $\ll$Role$\gg$. Interfaces of those diagrams specify sets of operations, provided by roles.

A role is an abstraction from an implementation class or an environment. Implementation classes and the environment can play different roles in a pattern of interaction. So, roles are used to capture the interaction pattern and reuse this pattern in different implementation and environment.

An interface-role diagram (Fig. 1) is a graph

$$\mathcal{IR} = (\mathcal{R}, \mathcal{I}, \mathcal{P}_I, \mathcal{R}_I, \mathcal{R}_R)$$

with two kinds of node and three kinds of relation:

- $\mathcal{R}$ is a finite set of roles. Each role $r \in \mathcal{R}$ is depicted by a box.

For example, there are three roles in Fig. 1: User, Graph Maker and Graph Drawer. In general, a role can have several players (instances), but we do not refer to players in this paper.

- $\mathcal{I}$ is a finite set of interfaces depicted by circles. Each interface $i \in \mathcal{I}$ has a finite set of operations $\mathcal{OP}_i$. Each operation has a finite set of results $\mathcal{Res}_i$.
Fig. 2. Interface suite for the second product of SPL named Graph Designer which receives data from a database.
In Fig. 1 there are three interfaces, \texttt{IGetGraph}, \texttt{IDataSeries} and \texttt{IDraw}. To simplify the case study we assume that each interface has only one operation with different results. So, we can use names of interfaces to identify operations. Results are shown as sets of values near the interface. For example, the operation of interface \texttt{IGetGraph} has the set of results \{true, false\}.

\begin{itemize}
  \item \( PI \subseteq \{(r, i) \mid r \in R, i \in I\} \) is a provide relation on roles and interfaces. Each role provides a finite set of interfaces.
  \item \( RI \subseteq \{(r', (r, i)) \mid r', r \in R, i \in I, (r, i) \in PI\} \) is a require relation on roles and interfaces. Each role requires a finite set of provided interfaces.
  \item \( RR \subseteq \{(r, r') \mid r, r' \in R\} \) is a relation of inheritance on the set of roles. The relation is shown by a solid line with the triangle end \( r' \mapsto r \) directed from role-child \( r' \) to role-parent \( r \).
\end{itemize}

In Fig. 1 the relation of inheritance is empty. In Fig. 2 role \texttt{New Graph Designer} inherits roles \texttt{User}, \texttt{Graph Maker} and \texttt{Graph Drawer}.

\subsection{Sequence diagram}

Our sequence diagram (Fig. 1) is a UML sequence diagram [20] \( S = (B, A_s) \), where

\begin{itemize}
  \item \( B \) is a set of boxes with a dashed line drawn down from the box.
  \item \( A_s \) is a set of labelled arcs. An arc \((v, w, l) \in A_s\) is depicted as an arrow that connects the dashed line running from box \( v \) to the dashed line running from box \( w \), where \( v, w \in B \). An arc has a label \( l \).
\end{itemize}

In our profile, boxes \( v, w \) represent players (instances) of roles from the interface-role diagram. For the sake of simplicity we assume that each role has only one player, so a box represents a role, \( v, w \in R \), for example, roles \texttt{User}, \texttt{Graph Maker} and \texttt{Graph Drawer} (Fig. 1).

A label \( l \) is a tuple \( l = (n, rp, ior) \), where

\begin{itemize}
  \item \( ior \) is an operation call or return from the interface-role diagram. In general, \( ior = interface\_operation \) for an operation call and \( ior = interface\_operation : result \) for an
Fig. 3. Interface suite for Real-Time Graph Designer with database.
operation return. In our case, each interface has only one operation, so \( \text{ior} = \text{interface} \) or \( \text{ior} = \text{interface : result} \). For example, \( \text{IGetGraph: true} \) in Fig. 1.

- \( n = 1, 2, \ldots, N \). \( n \) gives natural numbers to labels. A natural number at an arrow allows distinguishing several occurrences of label \( l \) in a sequence diagram (1 : \( \text{ior} \)), (2 : \( \text{ior} \)) etc.
- \( \text{rp} \) is a repetition symbol, \( \text{rp} \in \{\omega, \text{st}_j, \text{f}_j\} \). Repetition symbol value \( \text{st}_j \) (Fig. 3) is used to indicate the start of a repeated subsequence, where \( j \) is a number of a repeated subsequence within sequence diagram \( S \). Repetition symbol value \( \text{f}_j \) indicates the end of the repeated subsequence \( j \). A sequence diagram can have several repeated subsequences \( j = 1..m \). By convention we omit the empty value \( \text{rp} = \omega \) for all elements of \( A \) which do not start or end any repeated subsequence.

2.2.3. Process semantics

A component specification in our profile is a process of type

\[
P = (A, z, z_1, \ldots, z_F, T),
\]

where

- \( A \) is a finite set of actions \( a \in A \).

An action \( a \) has the following compound name for an operation call

\[
a = \text{role}^1.\text{role}^2.\text{interface}
\]

and

\[
a = \text{role}^1.\text{role}^2.\text{interface : result}
\]

for an operation return. The set of actions \( A \) of the process is exactly defined by the set of arcs in all sequence diagrams \( A_{s_1} \cup \ldots \cup A_{s_k} \). In turn, the set of arcs at all sequence
diagrams of a product specification $IS$ is a multiset on the require relation set $RI$ from the interface-role diagram of this product (some of operations can be called several times).

- $z, z_1, z_2, \ldots, z_F$ is the finite set of abstract states from the initial state $z$ to the final $z_F$.
- $T$ is a set of transitions. A transition $t \in T$ defines a pair of states $(z', z'')$, such that $z''$ is reachable from $z'$ as a result of the action $a: z' \xrightarrow{a} z''$.

For example, action $a_1 = User.GraphMaker.IGetGraph$ in Fig. 1 means that role $User$ calls interface $IGetGraph$ provided by role $GraphMaker$, how it has been specified by the interface-role diagram. Fig. 1 also shows the complete set of actions $A = \{a_1, a_2, \ldots, a_l\}$ defined by the UML specification of product $Graph Designer$.

We have developed an algorithm for constructing a process term from the set of processes corresponding to sequence diagrams $[22, 23]$. To construct the process term corresponding to a set of sequence diagrams:

- We construct set of processes $SP = \{p_1, \ldots, p_m, \ldots, p_M\}$ corresponding to sequence diagrams $S_1, \ldots, S_m, \ldots, S_M$:

1. If sequence $S_m$ does not contain repeated subsequences ($rp = \omega$), we construct a process term of type

   $$p = \text{createPlayers} \cdot a_1 \cdot \ldots \cdot a_j \cdot \ldots \cdot a_m$$

   where $a_j \in A_{S_m}$ and $\text{createPlayers}$ is a set of create-actions. The create-actions are calls and returns of create-interfaces provided by all roles of a component. Those create-interfaces are required by some hidden role $Factory$.

2. If sequence $S_m$ contains a repeated subsequence then we construct a process term of type

   $$p = \text{createPlayers} \cdot a_1 \cdot \ldots \cdot a_{j-1} \cdot (st, a_j) \cdot \ldots \cdot (f, a_{j+l}) \cdot W = \text{createPlayers} \cdot a_1 \cdot \ldots \cdot a_{j-1} \cdot (st, a_j) \cdot \ldots \cdot (f, a_{j+l}) \cdot \omega + W$$

   Symbol $r$ indicates a cycle in the process. $W$ is a process that follows the cycle. $(st, a_j) \cdot \ldots \cdot (f, a_{j+l})$ is the body of the cycle from start action $(st, a_j)$ to final action $(f, a_{j+l})$.

   This form of a process term is suitable for representing nested cycles.

   - In set $SP$, we find subsets $SP_1, \ldots, SP_k, \ldots, SP_K$ of parallel processes. Each pair of these subsets has processes which are started by different players and have disjoint sets of actions.
   - For each subset $SP_k$ in which processes have joint sets of actions or/and are started by the same players we compose a single process term $Z_k$. We apply a special tree-constructing algorithm, which keeps alternative branches with possible cycles, nested and/or sequential.
   - Finally, we construct a process term of type $P := Z_1 \parallel \ldots \parallel Z_K$ and the process graph corresponding to this process term $[1]$.

The resulting process term can be easily transformed back to sequence diagrams: each path of the process graph is mapped onto a sequential process corresponding to a sequence diagram.
The process term $p_1$ constructed by our algorithm using the UML specification of product Graph Designer is shown in Fig. 1.

2.2.4. A product specification in the UML profile with the process semantics

First product of SPL Graph Designer. The interface-role diagram of the first product is presented in Fig. 1. Role Graph Maker provides interface $IGetGraph$, which is required by role User.

The behavioural pattern of Graph Designer is presented by the set of sequence diagrams (Fig. 1). To simplify the picture we assume that each role has only one player, so, it is possible to talk about an interaction between roles.

The behavioural pattern for the first product of SPL Graph Designer is the following: role User asks role Graph Maker via interface $IGetGraph$ to draw a graph of a predefined type; role Graph Maker demands data series from role User via interface $IDataSeries$; User sends data series to Graph Maker by means of action $IDataSeries:structure$. Next steps correspond to the pair of actions, which Graph Maker and Graph Drawer perform before the visualization of the graph. Graph Maker commands Graph Drawer to draw the graph using interface $IDraw$. Graph Drawer prepares data structures to be drawn and returns them as a result via the same interface. The last action is response $IGetGraph: true$ from Graph Maker to User on the user’s request from the first step. This successful visualization of a graph is presented by the left sequence diagram (Fig. 1). The second sequence diagram in Fig. 1 corresponds to the case when the user’s data is not complete or correct to be drawn. In this case, Graph Maker returns result $IGetGraph: false$ to User. The set of actions and process $p_1$ corresponding to the UML specification are shown in Fig. 1.

3. Inheritance of product specifications

Inheritance relations defined on the set of UML specifications of products are a key element for modelling software product lines.

A product specification is an abstraction from implemented components. The specification describes interfaces of some future components and explicit dependencies of those components from an environment. These dependencies are modelled as behaviour of roles in a predefined environment including other system parts. It is this behaviour that should be reused when we replace an implemented component by another implemented component or when we replace an environment by a different environment. There will be always other components, other elements of environment that cause the same behaviour. That is why we consider a product specification as an interaction pattern. Because of the abstraction from implementation, the pattern is explained in terms of roles, communicating via interfaces provided by these roles. This gives us a freedom to implement several roles as one implemented component or represent a role by several implemented components. This also allows us to represent environment and future components in a uniform and platform independent way [21].

This behavioural pattern is captured in our design method as a process-term constructed from the UML specification.
The process semantics of our UML profile allows us to transform the inheritance relation on processes [2] to inheritance relations on product specifications both at the interface-role diagram and the sequence diagram levels.

In work [2] it has been defined that for any processes $p$ and $q$ being closed terms in a process algebra $(PA^+_\delta + RN)(A)$ [2], process $q$ inherits process $p$ if and only if there are sets

$$H \subseteq A$$

and

$$I \subseteq A, \ I \cap H = \emptyset,$$

such that process $p$ is derived from process $q$ in the process algebra $(PA^+_\delta + RN)(A)$ using hiding function $\delta_H$ and abstracting function $\tau_I$:

$$(PA^+_\delta + RN)(A) \vdash \tau_I(\delta_H(q)) = p.$$  

The left hand side of the derivation rule $\tau_I(\delta_H(q))$ defines the rewriting rules for process $q$ to derive process $p$.

The signature and axioms of process algebra $(PA^+_\delta + RN)(A)$, which is an abbreviation for Process Algebra with blocking action $\delta$, silent action $\tau$ and renaming $RN$, are given in Table 1. Axioms $A_1$–$A_7$ formalize alternative and sequential composition of processes, $M_1$–$M_4$ behaviour of concurrent processes, constant $a \in A \cup \{\delta\}$. Axioms $B_1, B_2$ allow us to remove silent action $\tau$ which does not enforce a choice. Axioms $D_1$–$D_4, \ T_1$–$T_4$ introduce renaming operators. The blocking operator $\delta_H$ renames actions from $H \subseteq A$ in a process term to blocking action $\delta$. The hiding operator $\tau_I$ renames actions from $I \subseteq A$ in a process term to silent action $\tau$ [2].

We have found a practical use of this theoretical result, namely, we have found

* how to define this process algebra $(PA^+_\delta + RN)(A)$ using UML specifications of SPL products. 

---

Table 1

Process algebra $(PA^+_\delta + RN)(A)$

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$x + y = y + x$</th>
<th>$A_2$</th>
<th>$(x + y) + z = x + (y + z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_3$</td>
<td>$x + x = x$</td>
<td>$A_4$</td>
<td>$(x + y) \cdot z = x \cdot (y + z)$</td>
</tr>
<tr>
<td>$A_5$</td>
<td>$(x \cdot y) \cdot z = x \cdot (y \cdot z)$</td>
<td>$A_6$</td>
<td>$x + \delta = x$</td>
</tr>
</tbody>
</table>

$B_1$: $x \cdot \tau = x$

$B_2$: $x \cdot (\tau \cdot (y + z)) = x \cdot (y + z)$

$D_1$: $a \not\in I \Rightarrow \tau_I(a) = a$

$D_2$: $a \in I \Rightarrow \tau_I(a) = \tau$

$D_3$: $\delta_H(x + y) = \delta_H(x) + \delta_H(y)$

$D_4$: $\delta_H(x \cdot y) = \delta_H(x) \cdot \delta_H(y)$

$M_1$: $x || y = x || y + \parallel x$

$M_2$: $a || x = a \cdot x$

$M_3$: $a \cdot x || y = a \cdot (x || y)$

$M_4$: $(x + y) || z = x || z + y || z$
how to choose the set of all actions $A$, the set of actions $I$ that should be hidden and the set of actions $H$ that should be blocked to derive parent process $p$ from child process $q$.

All our findings we have implemented in a tool for designers of product lines [23].

3.1. Inheritance at the interface-role diagram level

Let us see how the inheritance of processes is specified by interface-role diagrams. Let interface-role diagram $IR_p$ of a parent product be given. We should specify a new product which inherits behaviour of this parent. To prevent the erosion of the parent behaviour the roles of the new product are not allowed to require interfaces of parent roles from the interface-role diagram $IR_p$, and roles from parent $IR_p$ are not allowed to require interfaces from $IR_q$ of the new product. To be reused the parent interfaces should be provided by role-inheritors of parent roles. This way the interface-role diagram of the new product contains the action space for both the parent product and the new product.

To define inheritance between interface-role diagrams, we use inheritance relation on classes, which is defined in the UML and represented by an arrow with a triangle end. In our profile, classes represent roles. If role $r_1$ inherits role $r_2$, then we note this as follows: $r_1 \leftarrow r_2$.

Let interface-role diagrams be given:

\[
IR_p = (R_p, I_p, PI_p, RI_p, RR_p),
\]

\[
IR_q = (R_q, I_q, PI_q, RI_q, RR_q).
\]

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

1. Role sets $R_q$ and $R_{new}$ are disjoint. $R_q = R_p \cup R_{new}$.
2. Interface sets $I_p$ and $I_{new}$ are disjoint. $I_q = I_p \cup I_{new}$.

\[
RR_{new} = RR_p \cup RR_{eq}, \quad \text{where}
\]

\[
RR_{eq} = \{ (r_p, r_{new}) | r_p \in R_p \land r_{new} \in R_{new}, \land r_{new} \leftarrow r_p \}.
\]

So, the relation $RR_{eq}$ defines subset of roles $R_d \subseteq R_{new}$, which have parents in set $R_p$.

For example, role New Graph Designer (Fig. 2) has three parent roles. However, there is a new role Graph Data Source which has no parents.

4. Elements of the provide relation from role-parents are duplicated (inherited) in role-inheritors.

\[
PI_q = PI_p \cup PI_d \cup PI_{new} \cup PI_d,
\]

\[
PI_d = \{ (r_d, i) | r_d \in R_d \land i \in I_p \land (\exists r \in R_p [r_d \leftarrow r \land (r, i) \in PI_p]) \}.
\]

For example, role New Graph Designer (Fig. 2) provides the same interfaces as its parent: IDraw; IGetGraph; IDataSeries.
(5) Element \((x, (r, i))\) of the require relation \(RI_p\) is inherited if both role \(r\) that provides interface \(i\) and role \(x\) that requires interface \(i\) are inherited.

\[
RI_q = RI_p \cup RI_{\text{new}} \cup RI_{\text{d}}.
\]

The inheritance on interface-role diagrams specifies the duplicating function \(\rho_{RI_d}\).

\[
RI_d = \{(x_d, (r_d, i)) \mid r_d \in R_d \wedge i \in I_p \wedge (\exists x \in R_p \mid r_d \not\rightarrow r \wedge x \not\rightarrow x \wedge (r, i) \in PI_p \wedge (x, (r, i)) \in RI_p)\}.
\]

For example, role New Graph Designer (Fig. 2) requires interface IDraw because this role inherits both the parent-provider GraphDrawer and the parent-requirer Graph Designer.

3.2. Inheritance at the sequence diagram level

Let us see how the inheritance of processes is specified by the set of sequence diagrams.

Sequence diagram set \(S_q\) of a new product which inherits a parent sequence diagram set can assume different forms. It is impossible to predict all variants of such sets. So, we do not restrict designers in designing of sequences, but we check that the process constructed from the new sequence diagram set inherits the process constructed from the parent sequence diagram set.

We have extended the process algebra of type \((PA^+_3 + RN)\) by one axiom \(R\) to enable specification of nested cycles and sequences of cycles. So, we use process algebra \((PA^+_3 + RN + R)\). Axiom \(R\) means that process \(y\) which follows a cycle \(x \cdot (r \cdot x + w)\), where \(x\) is a process body of the cycle, \(r\) is an action indicating repetition, \(w\) is the output action, cannot be added to the body of the cycle \(r \cdot x\), but it is added to the output action \(w\) of the cycle.

\[
R : \ x \cdot (r \cdot x + w) \cdot y = x \cdot (r \cdot x + w \cdot y).
\]

The set of actions \(A\) for this process algebra is a multiset on set \(RI_p = RI_p \cup RI_{\text{new}} \cup RI_d\) (some elements of \(RI_q\) can be repeated several times) extended with repetition actions \([r_1, \ldots, r_G]\).

Let parent sets of sequence diagrams \(S_p\) and child set of sequence diagrams \(S_q\) be given; processes \(p\) and \(q\) have been constructed from set of sequences \(S_p\) and \(S_q\) correspondingly.

Process \(q\) inherits process \(p\) if and only if there exist such sets \(H \subseteq A\) and \(I \subseteq A\), \(I \cap H = \emptyset\) that process \(y = \rho_{RI_d}(p)\) duplicated from the parent process \(p\) is derived from the process \(q\).

\[
(PA^+_3 + RN + R) \vdash \tau_{RI_{\text{new}}} (\delta_H RI_{\text{new}}(q)) = y.
\]

The right hand side \(y = \rho_{RI_d}(p)\) of the derivation rule is used to define the process inherited from the parent. The left hand side of the derivation rule \(\tau_{I}(\delta_H(q))\) defines the rewriting rules for process \(q\).

We have automated the rewriting rules as graph transformation rules (Fig. 4) for process graphs [1] that correspond to process terms. The set of actions that should be blocked \(H \subset RI_{\text{new}}\) contains the new actions that start new alternatives. The set of actions that
Fig. 4. (a) $\delta H(q)$ and (b) $\tau_I(q)$ graph transformation rules.

3.2.1. Partial inheritance

If a new product specification inherits roles of an old specification, but the sequence diagrams of the new product do not use all the actions defined by the parent specification, then we deal with partial inheritance. In such a case, a partial behavioral pattern should be inherited. This partial behavioural pattern first should be derived from the parent process.

The duplicating function defined by the inheritance relation on the interface role diagram of the new product $\rho_{RI_d}(p)$ allows us to define $RI_d$, the complete set of actions that can be inherited.

From the set of sequence diagrams of the new product we can collect the set of action names. The behaviour of a product is specified by new sequence diagrams using action names from the interface-role diagram. The list of action names is constructed by our tool. The process corresponding to the set of sequence diagrams is constructed. Then using rewriting rules we check that the new product inherits processes of its parent products.

3.3. Software product line specification

We represent requirements for each new product as an interface-role diagram and collect the set of action names. The behaviour of a product is specified by new sequence diagrams using action names from the interface-role diagram. The list of action names is constructed by our tool. The process corresponding to the set of sequence diagrams is constructed. Then using rewriting rules we check that the new product inherits processes of its parent products.

**Second product of the SPL Graph Designer.** Graph Designer which receives data from a database is developed using inheritance at the interface-role diagram and the sequence diagram levels (Fig. 2). At the interface-role diagram we can see that IS Graph2 inherits
IS Graph1. Role New Graph Designer inherits all three roles of the parent product. So, according to the definition of inheritance, it also inherits all parent interfaces. To extend parent functionality we have added role Graph Database, which supplies data series to role New Graph Designer via new interface IDatabase Series.

If a child IS inherits a set of parent roles with the specified behaviour, this behaviour is inherited by the child IS as a subprocess. For example, if role New Graph Designer inherits all roles of the first product IS, it inherits the behaviour pattern of the first product. So, the second product is able to draw graphs using data received from a user. Role NewGraphDesigner inherits provided interfaces IGetGraph and IDraw and can require these interfaces (Fig. 2).

The second product inherits the behavioural pattern of the first product (Fig. 1) and extends it by the set of two new diagrams (Fig. 2). The interface-role diagram of the second product defines renaming function \( \rho'_{R_{\text{IS}}} \) which duplicates actions \( a_1, \ldots, a_7 \) to \( b_1, b_9, b_{10}, b_4, b_5, b_6, b_8 \) (Fig. 2). The duplicated parent process \( p'_1 = \rho'_{R_{\text{IS}}} (p_1) = \text{start} \cdot \text{createPlayers} \cdot b_1 \cdot b_9 \cdot b_{10} \cdot (b_6 + b_4 \cdot b_5 \cdot b_6) \cdot \text{final} \) is derived from the process of the second product via blocking \( \delta \) of new actions \( H = \{b_2, b_3, b_7\} \).

Indeed, process \( p_2 \) inherits \( p'_1 \):

\[
\delta_H (p_2) = (\text{Axiom } D_3, D_4) \delta_H (\text{start} \cdot \text{createPlayers} \cdot b_1 \cdot (b_2 \cdot b_3 \cdot (b_4 \cdot b_5 \cdot b_6 + b_7 \cdot b_8) + b_9 \cdot b_{10} \cdot (b_8 + b_4 \cdot b_5 \cdot b_6)) \cdot \text{final}) = \\
(A\text{xiom } D_1, D_2) \delta_H (\text{start} \cdot \text{createPlayers} \cdot b_1 \cdot (\delta \cdot \delta (b_4 \cdot b_5 \cdot b_6 + \delta \cdot b_8) + b_9 \cdot b_{10} \cdot (b_8 + b_4 \cdot b_5 \cdot b_6)) \cdot \text{final}) = \\
(A\text{xiom } A_7) \text{start} \cdot \text{createPlayers} \cdot b_1 \cdot b_9 \cdot b_{10} \cdot (b_8 + b_4 \cdot b_5 \cdot b_6) \cdot \text{final} = p'_1.
\]

The third product of the SPL named Real-Time Graph Designer is presented by Fig. 3. We have created two new roles Timer and New Real-Time Graph Designer. Role New Real-Time Graph Designer inherits all roles of the previous IS. These two new roles realize real-time drawing via five new interfaces.

Role New Real-Time Graph Designer uses its own interface IGetRTGraph to initialize real-time graph drawing. Next, New Real-Time Graph Designer starts Timer via interface ISetTimer (Fig. 3). Timer repeatedly generates calls of interface IOnTime. New Real-Time Graph Designer performs all inherited actions required to get a snapshot graph. To stop the drawing of snapshot graphs role New Real-Time Graph Designer calls interface IStopTimer provided by role Timer. (Let us assume that all graph snapshots are successful in this case.)

Using the sequence diagram of the third product we have constructed the process term

\[ p_3 = \text{start} \cdot \text{createPlayers} \cdot c_1 \cdot c_2 \cdot c_3 \cdot c_4 \cdot (s_t, c_5) \cdot c_6 \cdot c_7 \cdot c_8 \cdot c_9 \cdot (f, c_{10}) \cdot (r \cdot (s_t, c_5) \cdot c_6 \cdot c_7 \cdot c_8 \cdot c_9 \cdot (f, c_{10}) + c_{11} \cdot c_{12} \cdot c_{13} \cdot c_{14}) \cdot \text{final}. \]

This process inherits from the second product only actions \( b_2, b_3, b_4, b_5 \) renamed to \( c_1, c_8, c_9, c_{10} \). The set of actions that have not been inherited is \( X = b_1, b_6, \ldots, b_{10} \).

To define the partial process of product 2 which is inherited by product 3, we block actions from set \( H_X \subseteq X \), \( H_X = \{b_7, \ldots, b_{10}\} \) starting alternatives without reused actions. We also hide actions from set \( I_X \subseteq X \), \( I_X = \{b_1\} \) in the process \( p_2 \). This way we derive parent partial process \( p'_2 \) that should be inherited \( p'_2 = \text{start} \cdot \text{createPlayers} \cdot b_2 \cdot b_3 \cdot b_4 \cdot \ldots \)
**4. Tool support for product line design**

The described method comprises several formal techniques and algorithms to be used during a modelling process. The successful usage of the method requires appropriate tool support. We have developed a tool that provides an environment for design and reuse of component specifications in the UML [23]. The tool is implemented as a Rational Rose ADD-IN [15].

Using the tool a designer performs the following sequential steps (Fig. 5):

1. He/she chooses a parent product to inherit from. The interface-role diagram of this product is drawn by the tool in a Rational Rose class diagram window.
2. The designer extends the parent interface-role diagram by new roles and interfaces using dialogs provided by the tool. The interface-role diagram of the new product is produced.
3. The designer draws a set of sequence diagrams using the set of actions derived by the tool from the interface-role diagram of the new product.
4. The tool constructs the process graph corresponding to the UML specification of the new product.
5. The tool defines action sets that should be hidden and blocked in the process graph of the new product to derive the parent process graph, hides and blocks those actions and compares the parent process graph with the process graph-result of hiding and blocking.
6. If the process graph-result is not equal to the parent process graph, then the sequence diagrams that represent unreachable behaviour patterns are indicated by the tool. The designer should correct the design of the new product.
7. If the process graph-result is equal to the parent process graph, then the new product specification is correct and it can be used in further product development phases.
Fig. 5. Tool support.

A screen shot of a derivation dialog for *Graph Designer which receives data from a database* is shown in Fig. 6.

4.1. Complexity issues

The tool implements four main algorithms, the complexity of which should be considered:
Fig. 6. Parent process derivation dialog in the tool.

1. the algorithm for process graph constructing (step 4). The algorithm consists of two subtasks:
   (a) finding subsets of sequences forming parallel processes;
   (b) process tree constructing for each parallel process;
2. the algorithm of searching new actions on the interface-role diagram of a child product (step 5);
3. the algorithms for hiding and blocking of new actions (step 5);
4. the algorithm for comparing parent and modified child process trees (step 6).

The complexity of algorithms depends on three design parameters:
- \( M \) — the number of sequence diagrams;
- \( L \) — the length (in actions) of a sequence;
- \( N \) — the number of new actions to be hidden or/and blocked during derivation of a parent process graph.

All three parameters are independent, i.e. they can be set by a designer autonomously.

1(a). The first subtask of the process graph constructing algorithm, i.e. finding subsets of sequences forming parallel processes (the definition is given in Section 2.2.3), is solved as a linear search of matching actions in a set of \( M \) sequences, each of them containing, say, \( L \) actions. The worst case is, when deciding whether or not two sequences contain the same actions, we need to compare all \( L \) actions. This worst case corresponds to the
situation when we have $M$ parallel processes and we should perform pair by pair comparing of all $M$ sequences. So, the complexity linearly depends on number of comparisons $n = (M \cdot (M - 1)/2) \cdot L$ and can be evaluated as $\Theta(n)$ ("asymptotically equal" [14]). For design parameters the complexity should be $\Theta(M^2)$ and $\Theta(L)$, correspondingly. Keeping in mind that each parallel process tree may contain more than one sequence and, therefore, the real number of compared sequences should be less than $M \cdot (M - 1)/2$, it is feasible to conclude that the average case lies far below $O(M^2)$ ("less than or equal" [14]).

1(b). The problem of process tree constructing. Our algorithm is simpler than traditional tree building algorithms because all sequences running tree paths are already constructed. The primitive operation of this algorithm is comparing two actions, one from an already built process tree and one from a sequence to be fused with it. The worst (but, fortunately, non-idealistic) case is when all $M$ sequences are identical and should be fused completely. Indeed, then we have to compare each action from a current fused sequence with the corresponding action from the already built tree. In such a case the complexity, which depends on the number of compared pairs of actions $n = M \cdot L$, is linear and it is $\Theta(n)$.

As well, the algorithm input parameter $n$ linearly depends on any design parameter from $\{M, L\}$. Each other possible case, when not all sequences are identical, decreases the number of actions to be compared. Therefore, we can conclude that the complexity of that algorithm is $O(n)$ for any design parameter.

2. The algorithm of searching new actions on an interface-role diagram of a child product (step 5) consists of the (shortening each time) walks through the set of child’s actions repeated for each parent action. It linearly $O(N)$ depends on the number of actions.

3. The algorithm for hiding and blocking of new actions (step 5) is traversing the process tree for each new action to be blocked or hidden. The upper evaluation of complexity comes from the assumption that a new action should be hidden. In such a case we should traverse the entire process tree. The complexity is lower when a new action should be blocked. In that case the running branch is cut down and actions that follow the blocked action need not to be traversed already. For traversing the process tree we use the pre-order algorithm. Its complexity is $\Theta(n)$ [14]. Accompanying pointer operations of shortening or cutting affect only one (traversed) action each time and, therefore, do not change the class of complexity. Assuming a possible case when all $N$ new actions should be hidden the maximal nodes on the tree (number of actions to be traversed) is $n = N \cdot M \cdot L$ and we can consider complexity as $\Theta(n)$ against any design parameter from $\{N, M, L\}$. In practice (the average case) the complexity should be $O(n)$.

4. The tree comparison algorithm (step 6) performs comparing of each parent’s sequence with each path on the entire modified child process tree in order not only to conclude about matching or difference between them but also to find not eliminated blocking and hiding actions pointing on the wrong design. So, the worst case is when both trees are identical. In such a case we should compare $n = M^2 \cdot L$ pairs of actions and complexity is, therefore, $\Theta(M^2)$ and $\Theta(L)$ respectively.

So, we can conclude that the complexity of used algorithms lies under the boundary of practicability, i.e., polynomial-time efficiency [14].
5. Evolutionary software development process

Let us discuss the application of our evolutionary SPL modelling approach to an SPL development process (Fig. 7).

A designer compares requirements to a new product with the requirements specifications of already implemented components and finds a parent to inherit from. The required variations are determined and the IS specification of the new product is created on the basis of the parent specification in such a manner that new extensions (or reductions) do not damage inherited behaviour.

New functionality and behaviour are mapped onto the parent component architecture and required modifications of the architecture are specified. The modifications have to be made using (or, if necessary, modifying) the SPL component set. After implementation of the new product its IS specification becomes a part of SPL reusable assets.

So, we do not use domain engineering in the SPL development process. Instead, we propose including an additional phase into a traditional software development process between requirements engineering and component architecture design.

Requirements are very often specified in UML notation [13]. In such a case they can be mapped onto interface-role specifications directly: actors iterating via use cases can be mapped on roles; use cases itself can be realized as sets of required relations between roles; scenario diagrams can be considered as prototypes of sequence diagrams.

In design environments, such as Rational Rose, it may mean a model transformation from the Use Case View to the Logical View [15]. In any case, the role approach helps a designer to overcome a semantic gap between software requirements and a component architecture [33]. Roles are designed to abstract from a concrete component implementation. A pair of roles interacting via an interface can model a set of product related requirements, i.e. a product feature [4]. So, product functional requirements can be mapped onto interface-role specifications much easier than onto a component architecture.
The mapping IS specifications to a product component architecture can be done using the similarity between roles with interfaces and components. Components interact by playing roles. One or several interacting roles can be mapped to a product component architecture in such a way that component boundaries should come across the interfaces required by roles [33]. To illustrate such successful mapping, we have mapped our “toy” product line Graph Designer onto components from the repository of Borland Delphi 4 [17].

In Fig. 8:

- Delphi GUI Controls, Delphi BDE Controls and Access, Delphi Timer are Borland Delphi repository sets of implementation components (BDE—Borland Database Engine of Imprise Corp.);
- VtChart is a third party ActiveX component of Visual Components Corp.

Fig. 8 shows how boundaries between Delphi components come through IS require relations. In coding phase we needed only some tiny pieces of “glue” code to materialize these relations.

The process semantics of product specifications and the inheritance relation on product specifications that we have discussed in this paper can be used in other evolutionary approaches to SPL design. For example, such approach as Koala can be extended by the behavioural inheritance checks. Our interface suites can be mapped onto Koala’s configurations in such a manner that roles would correspond to components. Provide and require relations can be presented by connected Koala provide and require interfaces. The compositional capacity of a Koala component (combinations of components are components again [32]) provides appropriate support for inheritance of roles. Inheritance of interface-role specifications is supported by the ability of Koala’s configurations to
comprise other configurations. Successful mapping of our inherited product specifications onto Koala configurations can guarantee that those configurations are built in a correct manner from the system behaviour point of view.

5.1. Practical application and scalability

To illustrate our ideas we deliberately used a simplified case study. Meanwhile the method has already been successfully applied to a real system. It is a product line of Scientific Silicon Array X-Ray Spectrometer (SIXA) [26]. This is real-time embedded control software that comprises several subsystems fulfilling the roles of data acquisition and measurement control, data management and exchange. SIXA SPL members can work in two different measurement modes as well as the combined mode. It requires about 15 interfaces and 60 different action calls and returns corresponding to different hardware signals. In paper [26] we emphasize practical issues of the approach modelling three members of this product line. Recently we extended this SPL by another two members which support two processors and a hard disk. The method has shown good robustness and can be compared to other techniques because the case study is well documented and modelled also outside [9,16].

Of course, large scale industrial product lines could comprise far more than 15 interfaces, more likely hundreds of them (see e.g., [32]). The problem of scalability rises more sharply if designers have to deal with product lines rather than a single software product. However, our method provides some benefits in resolving scalability problems because our SPL design is incremental. At each step product modifications are usually observable. In different model views required refinements are introduced by means of new roles and/or interfaces and new sequences.

Inherited product functionality can usually be rolled up into a much smaller number of roles with inherited roles and interfaces hidden in them. For example, roles New Graph Designer and New Real-Time Graph Designer in Fig. 8 inherit all interfaces of the parent roles. Moreover, there is no need to include the entire hierarchy of parent IR diagrams into the inheritor’s one, although they are still part of the inheritor’s specification [18,24].

New behaviour is captured by new or modified sequence diagrams. In principle, each parent sequence diagram can be mapped automatically onto the inheritor’s specification to be modified by a designer. Explicitly declared inheritance relations between parent and child roles allow us to do this, although this is not implemented yet in the current version of the tool.

As far as process graph inheritance checks are concerned, we have shown earlier (see Section 4.1) that the complexity of inheritance check algorithms is practical with respect to each design parameter. So, the scalability of process modelling is not a problem because it can be done automatically.

6. Conclusion and future work

The approach presented in this paper provides evolutionary incremental SPL modelling based on inheritance of behaviour specifications of SPL members. The correctness of the evolutionary steps is proved by deriving a product-predecessor process from a
product-inheritor process. An appropriate tool prototype has been developed to support
the modelling. The tool applies techniques and algorithms which accompany the method.

The approach we propose is quite different from traditional architectural approaches. In
our approach, an SPL architecture should not be developed in advance. The architecture
grows evolutionarily as a hierarchy of already implemented products. We control the
growth by allowing only such new products that correctly inherit behaviour of old products.
So, the constraints on SPL growing come both from the rules of specification extensions
and from behaviour inheritance constraints given by designers.

It is much easier to tackle unanticipated changes of requirements in our approach than in
the traditional ones. In the traditional approaches the designer is restricted by the variability
points that have already been introduced into the SPL architecture. So not all the changes in
requirements can be made without changing the architecture. In our approach, to include a
new requirement a designer should find an implemented product line member with similar
behaviour and refine its specification using the technique, which the approach provides.
The designer should prove that the new product inherits behaviour of the chosen one and
include the new product to the software product line.

If an SPL already exists as an open for growth hierarchy of already specified products
each of which inherits behaviour of its predecessors, then internal members of the hierarchy
can also be changed. In future work we intend to investigate how changes in parent
specifications could affect inheritors’ ones (by analogy with time-line SPL variability).
There may be changes in parent specifications that do not affect child’s behaviour and,
therefore, can be accepted for the entire hierarchy. In contrast, some changes cannot be
accepted because they damage the SPL integrity. Perhaps, some changes may be applied
only locally or it may be feasible to maintain several hierarchies based on different versions
of some product line members. All these variants require behavioural inheritance checks
using our derivation technique.

Another future task (similar to SPL space variability) is to reveal how exactly a
growing SPL hierarchy should be saved and maintained among other SPL common assets.
Recently we consider the possibilities to use XML schemas and documents for this
purpose. Of course, an SPL hierarchy has to be maintained using appropriate configuration
management tools.

Behavioural inheritance is a helpful technique for software product line modelling.
This technique has a good computational basis and it is easily mapped onto requirements.
Studying human factors of design we have understood that there is an infinite set of possible
inheritance and reuse relations. We are developing now a logic of behavioural inheritance
which will allow designers formulating inheritance constraints as a sort of LEGO-game
combination. This logic of behavioural inheritance allows us to classify different cases of
inheritance constraints and implement corresponding algorithms for proving correctness
of reuse. The algorithm recently implemented in our tool allows designers to automatically
prove all the cases when parent processes are required to be extended by sequences and
alternatives built from new actions. Other cases use other algorithms the implementation
of which is work for the near future. We are also concentrating on different algorithms
for rolling up process trees and sub-page visualizing. These tasks are closely related to
specification of requirements to behaviour inheritance.
Acknowledgements

The work of Serguei Roubtsov has been supported by ERCIM (European Research Consortium for Informatics and Mathematics) and STW PROGRESS project EES5141 “Specification Tooling for Embedded Software Components” (The Netherlands).

References


