Evolutionary Product Line Modelling

Serguei Roubtsov[*]
VTT Electronics
VTT Electronics, Kaitovayla 1,
P.O.Box 1100
FIN-90571 Oulu, Finland
ext-Serguei.Roubtsov@vtt.fi

Ella Roubtsova
Eindhoven University of Technology
Den Dolech 2, P.O.Box 513
5600 MB The Netherlands
E.Roubtsova@tue.nl

Pekka Abrahamsson[†]
VTT Electronics
VTT Electronics, Kaitovayla 1,
P.O.Box 1100
FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

ABSTRACT

A traditional product line approach struggles 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 accompanied by correctness control of product model transformations. An industrial case study from the embedded systems domain demonstrating a modelling technique is provided. The approach is supported by an appropriate tool prototype.

1. INTRODUCTION

The product line approach is an approach to software reuse. In large-scale industrial systems it is used, for example, in embedded systems domain. Embedded software product lines such as consumers electronics applications are usually characterized by a huge variety of slightly different product line members [18].

The mainstream of approaches to software product line (SPL) development [8, 4] applies different diversity management techniques to a generic SPL architecture. This allows a designer to produce new products reusing common SPL assets [10] within the boundaries of such a generic architecture. This approach is robust but also complicated and not flexible enough in terms of evolution support.

On the other hand, a component-based development approach has its own worth in the SPL area [17, 5]. This approach employs composition of reusable components as a basis for product population [19] development. However, in the absence of a reference SPL architecture, the main advantage of the product line approach, i.e. controlled variability, may be damaged.

We propose an SPL modelling method that provides inheritance of implemented product line members model specifications accompanied by correctness control of model transformations. The method considers inheritance of product behaviour specifications as inheritance of processes [2, 22]. The method combines the flexibility of component-based approaches with the rigorous correctness of architecture-based techniques. As a result, a designer obtains an instrument that allows him to model new product line members quickly introducing new required functionality and avoiding design bags.

The rest of the paper is organized as follows. Section 2 provides a brief discussion about existing SPL approaches and raises the relevant problems. Section 3 describes a case study from the domain of embedded systems. Section 4 explains our method and provides corresponding illustrations using the case study. Section 5 describes the tool prototype, which has been developed to support our method. The paper is concluded in Section 6.

2. SOFTWARE PRODUCT LINES: STATE-OF-THE-ART APPROACHES AND PROBLEMS

Software product lines traditionally employ a top-down architecture -based methodology of software system development [8, 10, 4, 14, 9]. 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 (variabilities). On the basis of requirements analysis a common product line architecture and a 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

---

[*] The work of S.A. Roubtsov is supported by The European Economic Interest Grouping ERCIM (European Research Consortium for Informatics and Mathematics.

[†] The research is carried on within VTT Electronics Agile Software Technologies project: http://agile.vtt.fi.

members development [6] (Figure 1).

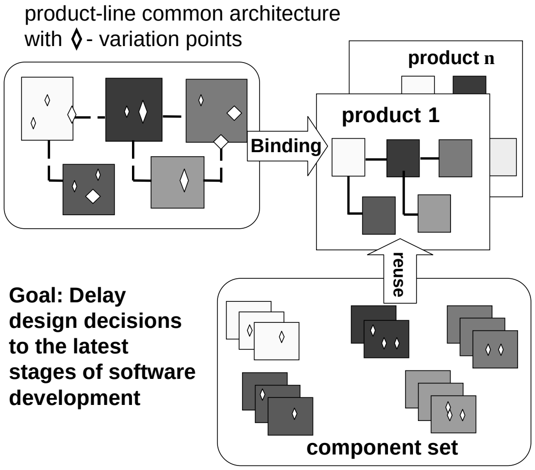


Figure 1: Traditional SPL modelling process.

So, a common SPL architecture with variability management fulfils a double role. Firstly, it provides the *reference of integrity* for SPL components 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 [19] 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 [14]. SPL development and maintenance give rise to a lot of related tasks, which have to be solved coherently [8, 4]. 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 [10, 4, 3].

The second problem is evolution support [25]. 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 [12] 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., [27]).

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

The alternative software reuse approach is an evolutionary component-based software development process [26]. In the SPL domain it is a product population approach [17, 19, 18, 5]. That approach uses lightweight [17] common architecture and implements software component modifications and component compositions instead of architecture-based variability management (e.q., [18]).

The benefits of evolutionary approaches are explicit. An SPL grows when new product line members appear. A design process is flexible and 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 line members design correctness rise sharply. Component modification and composition rules are static, they do not guarantee that the entire system behaviour comprises the behaviour of composition 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 line members design 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 should be proved (Figure 2).

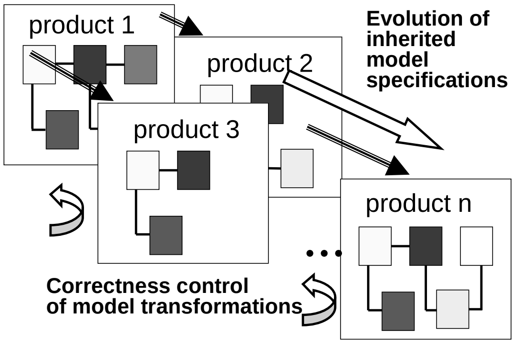


Figure 2: Evolutionary SPL modelling approach.

In our approach design specifications are implemented using UML (Unified Modeling Language) profile with defined inheritance relations on specifications [23]. The profile defines a special type of UML class diagrams, interface-role diagrams, similar to CATALYSIS approach [11]. Component system behaviour is specified in the profile using UML sequence diagrams as it was 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 inheritor's ones allows a designer to prove correctness of inheritance or to find the points of wrong design decisions.

In [21] the evolutionary SPL modelling technique is 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 previous theoretical results are extended by the notion of a product process graph. The notion of inheritance of product line members specifications is defined on the basis of a process graph definition. In this paper we also discuss the application of our method.

3. CASE STUDY: SCIENTIFIC SILICON ARRAY X-RAY SPECTROMETER

We intend to emphasize applicability of our method. Our case study is a product line representation of Scientific Silicon Array X-Ray Spectrometer (SIXA) Control Software [13, 9][1]. This is an onboard satellite system that provides scientific data in two measurement modes [13]: Energy Spectra (EGY) and Single Event Characterization (SEC).

Despite some differences between EGY and SEC measurement realizations there are also a lot of common requirements that makes it possible to regard this case study as an example of an SPL. Following [9] we intend to model three members of SIXA software product line:

○ stand alone EGY Controller
○ stand alone SEC Controller
○ combined EGY and SEC Controller

The key aspects of SPL modelling have to be found in the requirements, both functional and behavioural, to product line members. Let us consider them subsequently.

3.1 Product line members functionality

The SIXA Controller fulfils the following functional requirements [13]:
- it receives measurement programmes from the ground via a satellite computer,
- provides data measurement,
- collects and sends data back.

These requirements to the product line software can be described in terms of four interconnected subsystems [13] realizing main product features:

- *Measurement Control* subsystem. This subsystem provides *Controller Commands* interface with an onboard satellite computer. External control commands and measurement programmes come via this interface.
- *Data Acquisition* subsystem. It executes measurement programmes received via its interface *Control Data Acquisition* from *Measurement Control* subsystem.
- *Data Management* subsystem. It
  - fills its internal buffer with data received from *Data Acquisition* subsystem via interface *Save Data*.
  - sends scientific data back to the ground via *Satellite Computer* interface *Controller Data Response* following commands from *Measurement Control* subsystem via interface *Control File Management*.
- *Satellite Computer* that is regarded as an external system. It uses Spectrometer interface *Controller Commands* and receives scientific data via its own interface *Controller Data Response*.

The described above SIXA spectrometer functionality is common for the entire SPL.

The variability is defined by the different measurement modes that have to be implemented. EGY and SEC modes are realized by different specific *Data Acquisition* subsystems and corresponding interfaces *Control Data Acquisition* and *Save Data*. There is also slightly different organization of a data exchange process with the satellite computer: EGY Controller *Data Management* subsystem sends data to the satellite computer after measurement programme has been fulfilled completely, whereas SEC Controller *Data Management* subsystem can initialize data exchange when its internal buffer is full. So, this subsystem should be able to send such a request to *Satellite Computer*.

EGY and SEC Controller has to provide functionality of each stand alone mode whatever has been chosen by the ground measurement programme.

3.2 Product line members behaviour

The behavioural requirements to the SIXA Spectrometer software are defined by two data observation processes, one process for each observation mode [13]. Both processes comprise two sequential sub-processes: data measurement and data exchange. Using usual algorithmic notation the processes can be described as it is shown in Fig. 3. (We omit a few not significant technical details in order to draw a more clear picture.) Each block in Fig. 3 corresponds to an operation call that is performed by interacting SIXA Controller software subsystems and supported by hardware signals. The blocks above the dashed line (Fig. 3) perform the data measurement sub-processes, the blocks below this line correspond to the data exchange sub-process.

The data exchange sub-process is common for EGY and SEC modes: after sending to the ground the number of blocks with scientific data to be transmitted it performs a cycle of data blocks transmission.

The data measurement sub-processes are partially different. The dark blocks in Fig. 3 depict the steps of the measurement sub-processes which are different for EGY and SEC modes. The EGY measurement sub-process is performed subsequently for each of the predefined observation targets.

---

[1] We thank Prof. Eila Niemela and Tuomas Ihme from VTT Electronics for sharing the insights into this case study

This corresponds to the external cycle of the algorithm on the left hand side in Fig. 3. The algorithm on the right hand side does not contain this cycle because in SEC measurement mode a single target is observed continuously. For both modes a single target observation cycle lasts until an observation time is expired. However, in SEC mode the observation process can be interrupted when *Buffer Full* message is raised in the system.

The real SIXA spectrometer has more features to be modelled [9], support of a hard disk in SEC mode, for example. However, additional features can become part of future SPL members generations. The case study is enough to give a demonstration of how our method works.

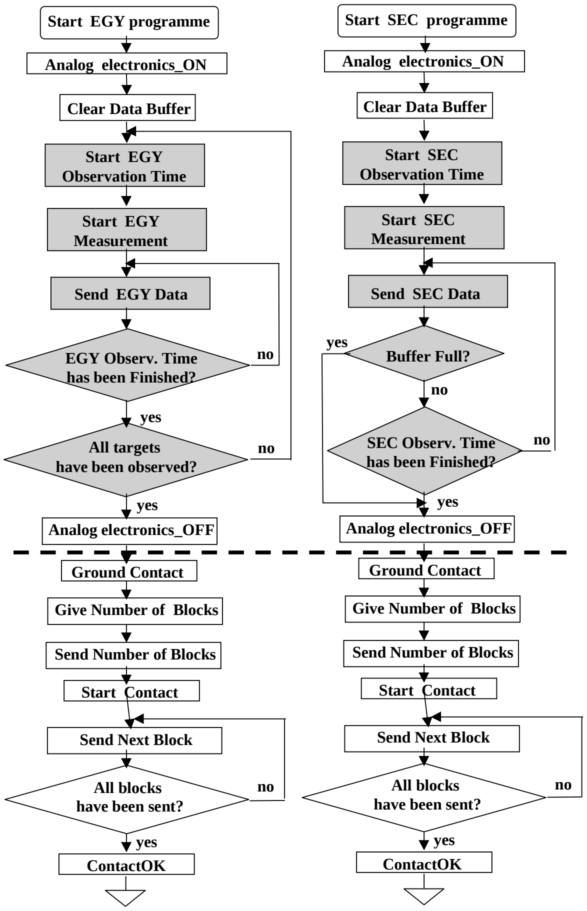


Figure 3: Observation algorithms for SIXA Spectrometer. On the left hand side: EGY mode; on the right hand side: SEC mode; measurement subprocess is above $- - --$ line; data exchange subprocess is below.

4. EVOLUTIONARY PRODUCT LINE MODELLING METHOD

The method includes two parts: a product model specification and the definition of inheritance of product line members specifications with the derivations rules providing correctness of model transformations.

4.1 Product Model Specification

The *product line member specification* is a pair

$$PrSp = (IR, BS)$$

where IR is an interface-role specification and BS is a behavioural specification.

4.1.1 Interface-role specification

The interface-role specification describes static aspects of product functionality. Roles can *provide* interfaces, which the 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.

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 during role modelling [28]. However, 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 provided by roles [28, 21].

Interface-role specification is a tuple

$$IR = (R, I, PI, RI, RR), where :$$

- R is a finite set of roles. $R = R_p \cup R_d$, R_p is a subset of roles that provide interfaces; R_d is a subset of roles that require interfaces. The same role can belong to both subsets R_p and R_d.
- I is a finite set of interfaces provided by roles from R_p. Each interface $i \in I$ has finite set of operations OP_i. Each operation $op \in OP_i$ has finite set of result values Res_{op}.
- $PI \subseteq \{(r, i)|\ r \in R_p, i \in I\}$ defines provided relations between roles and interfaces.
- $RI \subseteq \{(r', pi)|\ r' \in R_d, pi \in PI\}$ defines required relations between roles and interfaces. Each role requires a finite set of provided interfaces.
- $RR \subseteq \{(r, r')|\ r, r' \in R\}$ is a set of inheritance relations on the set of roles. These relations are part of inheritance relations between product line members specifications and will be considered later (see section 4.2.1).

The interface-role specification of EGY Controller is shown in Fig. 4. In all specification parts, where EGY Controller specifics has to be introduced, the names have prefix "EGY".

Four roles-providers correspond to four subsystems in the product requirements specification as well as five provided interfaces represent specified earlier (section 3.1) system interfaces.

Provided relations are presented by pairs (*role-provider*, *interface*), for example, (*Satellite Computer*, *IController Data*

Responce). For each such a pair each possible triple (*role-requirer*, *role-provider*, *interface*) represents a required relation, for example, (*EGYData Acquisition*, *EGYData Management*, *ISaved EGYData*) (Fig. 4).

Operation names in Fig. 4 are the same as the names of operations presented by blocks in Fig. 3. We only use a few abbreviations.

We have chosen EGY Controller to be the first product in the product line; hence its specification does not contain inheritance relations.

Roles-requirers (Rd)	Roles-providers (Rp)	Interfaces (I)		
		Names of interfaces	Operations (Op_i)	Result values (Res_{op})
EGY Data Management	Satellite Computer	IController Data Responce	SendNoOf Blocks(integer)	void
			SendNext Block(structure)	void
Satellite Computer	EGY Measurement Control	IController Commands	Analog_ON	void
			Start EGY Observation Time	void
			Finish EGY Observation Time	void
			Analog_OFF	true
			GroundContact	void
			ContactOK	void
EGY Measurement Control	EGY Data Acquisition	IControl EGYData Acquisition	StartEGY Measurement	true
EGY Measurement Control	EGY Data Management	IControl File Management	ClearData	void
			GiveNoOf Blocks	void
			StartContact	void
EGYData Acquisition		ISaved EGYData	SendEGYData (structure)	void

Figure 4: Interface-role specification IR_{EGY} of EGY Controller

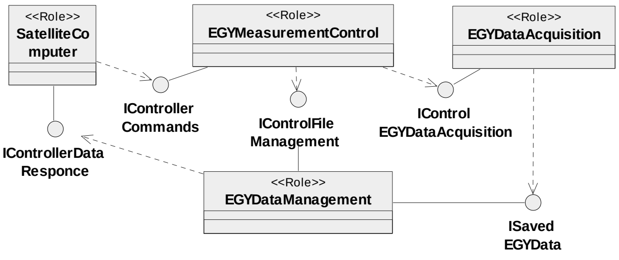


Figure 5: Interface-role diagram for EGY Controller

The interface-role specification is realized in the UML profile [23] and presented by a UML class diagram [16], where roles are UML classes with stereotype ≪Role≫ and interfaces are classes with stereotype ≪Interface≫. Interfaces are depicted by cycles. Provided relations are presented by UML realize-relations between roles and provided interfaces and depicted by solid lines [16]. Required relations are the same as UML dependency relations between roles and required interfaces. A required relation is depicted by a dashed arrow directed from a role to a required interface [16].

The interface-role diagram of EGY Controller is shown in Fig. 5.

4.1.2 Behavioural specification

The behavioural specification describes dynamic aspects of product functionality, i.e. product behaviour. A grain of product behaviour is presented by a *pair of actions* [22]. The first action of the pair is an *operation call*, the second one is an *operation return*. It has to be noticed here that operation calls and returns in the model specification are not the same as ones in the implementation phase: each modelled call and/or return can be implemented by one or several methods (procedures and functions).

An action name for the operation call is $a = r'.r.i.op$, which means "role r' calls operation op of interface i provided by role r ".

An action name for the operation return is $a = r'.r.i.op : res_{op}$, which means "role r returns result res_{op} responding to operation call $a = r'.r.i.op$".

As a result of product IR specification, action set A_{PrSp} is introduced for the entire product specification. To refer to the concrete actions of this set we apply on it a numeric order relation giving natural numbers to all actions:

$$A_{PrSp} = \{a_1, a_2, ...\}$$

The quantity of actions $a_i \in A_{PrSp}$ is defined completely by the quantity of operation calls and returns via required relations $ri \in RI$ between roles $r' \in R_d$ and $r \in R_p$.

Fig. 6 shows the action set for the EGY Controller specification. We omit interface names in action names for convenience. This is possible if operation names are unique for each pair of interacting roles. There are thirteen operation calls and same number of operation returns in this set.

Using action set A_{PrSp} we construct behavioural specification BS of a product line member as a finite *set of sequences* representing product behavioural patterns [22]:

$$BS = \{S_1, S_2, ..., S_n\},$$

where $S_i, \forall i = 1, 2, ..., n$ is a *sequence of actions* $a_j, a_k \in A_{PrSp}, \forall j, k = 1, ..., |A_{PrSp}|$:

$$S_i = \{a_j, a_k, ...\}$$

The last definition means that we can construct behavioural pattern S_i using any action from action set A_{PrSp} any number of times. We apply the restriction that one and only one action representing operation return must appear after (but not necessarily just after) the action that represents the corresponding operation call.

Any sequence S_i can contain any number nested in any depth repeated subsequences or *cycles* [21]. For example,

A_{EGY}={a1,...a26}

a1 - SatelliteComputer.EGYMeasurementControl.Analog_ON
a2 - EGYMeasurementControl.EGYDataManagement.ClearData
a3 - EGYMeasurementControl.EGYDataManagement.ClearData:void
a4 - SatelliteComputer.EGYMeasurementControl.Analog_ON:void
a5 - SatelliteComputer.EGYMeasurementControl.StartEGYObservationTime
a6 - SatelliteComputer.EGYMeasurementControl.StartEGYObservationTime:void
a7 - EGYMeasurementControl.EGYDataAcquisition.StartEGYMeasurement
a8 - EGYDataAcquisition.EGYDataManagement.SendEGYData(structure)
a9 - EGYDataAcquisition.EGYDataManagement.SendEGYData:void
a10 - EGYMeasurementControl.EGYDataAcquisition.StartEGYMeasurement:true
a11 - SatelliteComputer.EGYMeasurementControl.FinishEGYObservationTime
a12 - SatelliteComputer.EGYMeasurementControl.FinishEGYObservationTime:void
a13 - SatelliteComputer.EGYMeasurementControl.Analog_OFF
a14 - SatelliteComputer.EGYMeasurementControl.Analog_OFF:void
a15 - SatelliteComputer.EGYMeasurementControl.GroundContact
a16 - SatelliteComputer.EGYMeasurementControl.GroundContact:void
a17 - EGYMeasurementControl.EGYDataManagement.GiveNoOfBlocks
a18 - EGYDataManagement.SatelliteComputer.SendNoOfBlocks(integer)
a19 - EGYDataManagement.SatelliteComputer.SendNoOfBlocks:void
a20 - EGYMeasurementControl.EGYDataManagement.GiveNoOfBlocks:void
a21 - EGYMeasurementControl.EGYDataManagement.StartContact
a22 - EGYMeasurementControl.EGYDataManagement.StartContact:void
a23 - EGYDataManagement.SatelliteComputer.SendNextBlock(structure)
a24 - EGYDataManagement.SatelliteComputer.SendNextBlock:void
a25 - SatelliteComputer.EGYMeasurementControl.ContactOK
a26 - SatelliteComputer.EGYMeasurementControl.CoontactOK:void

Figure 6: Set of actions A_{EGY} for EGY Controller

sequence:

$$S_i = \{st_1, a_j, ...f_1, a_k, ...st_2, a_m, ...st_3, a_p, ...f_3, a_q, ...f_2, a_n\}$$

contains three cycles, the first cycle goes form a_j to a_k, the second one lasts from a_m to a_n. The third cycle $a_p, ...a_q$ is nested in the second one. Prefix "st," with the number of a cycle denotes the action starting repetition and prefix "f," with the same number denotes the action finishing repetition.

{Si}	Sequence of actions $a_j \in A_{EGY}$
EGYObservation	a1, a2, a3, a4, st_1,a5, a6, st_2,a7, a8, a9, f_2,a10, a11, f_1,a12, a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, st_3,a23, f_3,a24, a25, a26

Figure 7: Behavioural specification BS_{EGY} of EGY Controller

Behaviour of EGY Controller is specified by requirements to the EGY observation process which is described in section 3.2. Using this specification we have designed behavioural specification

$$BS_{EGY} = \{EGYObservation\}$$

containing single sequence $EGYObservation$ (Fig. 7).

The behavioural specification is realized in the UML profile [22] and presented by a set of UML sequence diagrams [16], one diagram for each sequence S_i. The precise definition of a sequence diagram for this UML profile is given in [21].

The sequence diagram for EGY Controller is shown in Fig. 8. This diagram corresponds to the algorithm on the left hand side in Fig. 3.

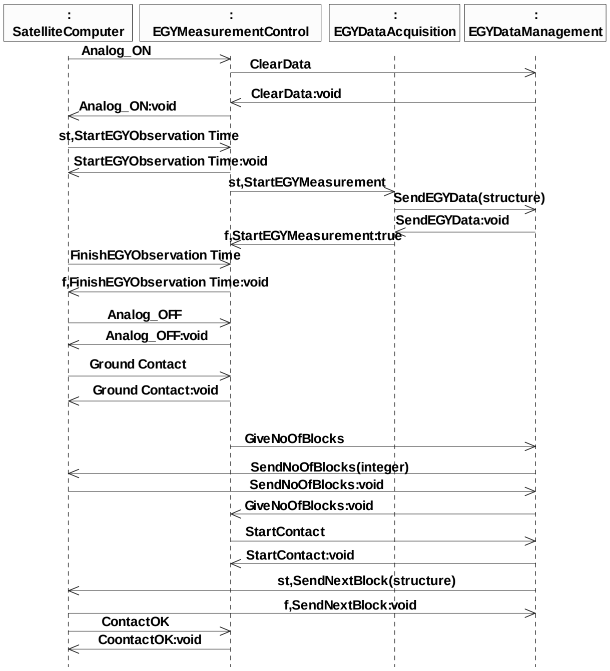


Figure 8: Sequence diagram EGYObservation for EGY Controller

4.2 Inheritance of Product Specifications

We regard inheritance of product line members as inheritance of product behaviour. If, for example, product EGY and SEC Controller inherits product EGY Controller, then it inherits the possibility to observe energy spectra and extends it by the SEC spectra observation facility.

Let us use notation $PrSp_q \longrightarrow\!\!\triangleright PrSp_p$ to depict *inheritance* of product $PrSp_q$ from product $PrSp_p$.

In our approach behaviour is presented by product BS specification. So, *product specification $PrSp_q$ inherits product specification $PrSp_p$ if behavioural specification BS_q inherits behavioural specification BS_p.*

Behaviour specification $BS_q = \{S_{1_q}, S_{2_q}, ..., S_{n_q}\}$ *completely inherits* $BS_p = \{S_{1_p}, S_{2_p}, ..., S_{m_p}\}$ *if* $n \geq m$ *and each sequence* S_{i_q} *inherits corresponding sequence* S_{i_p}.

If BS_q inherits a subset of sequences of BS_p we have the case of *partial inheritance.*

Hence, to define the inheritance of product specifications we need to define the inheritance of sequences presenting product behaviour patterns.

Each sequence S_i is defined by set of actions A_{PrSp} and this set is defined by set RI of required relations on product interface-role specification IR. So, first we need to define inheritance at the level of interface-role specifications.

4.2.1 Inheritance of interface-role specifications

Interface-role specification

$$IR_q = (R^q, I^q, PI^q, RI^q, RR^q)$$

inherits interface-role specification

$$IR_p = (R^p, I^p, PI^p, RI^p, RR^p)$$

if $\exists(r', r) \in RR^q | r' \in R^q, r \in R^p$ and $\neg\exists(r, r') \in RR^p | r' \in R^q, r \in R^p$

In other words, at least one role from IR_q inherits at least one role from IR_p and none of the roles from IR_p inherit roles from IR_q.

If role r' inherits role r: $r' -\!\!\triangleright r$, then [22]:

- role-parent r is included in specification IR^q;
- role-child r' inherits all interfaces, provided by role-parent and, hence, all its provided relations;
- role-child r' inherits required relation of role-parent r

  $ri = (r, pi) \in RI^p | pi = (r'', i) \in PI^p, r, r'' \in R^p, i \in I^p$,

  if role-provider r'' is also inherited by specification IR^q.

Inheritance of roles is defined in the UML profile [22] and corresponds to the specialize-relation between UML classes [16]. The relation is shown on the interface-role diagram by a solid line with the triangle end $-\!\!\triangleright$ directed from role-child to role-parent [16].

As a result of inheritance, the child interface-role specification comprises two parts:

$$IR_q = (IR_q^{Inh}, IR_q^{New}), where$$

IR_q^{Inh} contains inherited roles, their provided interfaces and provided relations, and, possibly, required relations; IR_q^{New} is a new part, which contains new roles, interacting via new interfaces; it realizes new product functionality and inherits the functionality of a parent product. The only possibility to utilize IR_q^{Inh} specification is to use its roles as parents in inheritance relations with roles from IR_q^{New} specification.

Dealing with our case study a designer should first decide how to order the chain of inheritance:

$$PrSp_{EGYandSEC} -\!\!\triangleright PrSp_{SEC} -\!\!\triangleright PrSp_{EGY}$$

or

$$PrSp_{SEC} -\!\!\triangleright PrSp_{EGYandSEC} -\!\!\triangleright PrSp_{EGY}.$$

In other words, what product should inherit EGY Controller first, SEC Controller or EGY and SEC Controller? Despite the fact that a usual composition way dictates the first variant, the second one is the right answer. If the first variant had been chosen, then role *EGYData Acquisition* from IR_{EGY} specification should have been replaced by a new role that fulfils another observation process and EGY data acquisition functionality would have been lost for further utilization.

The first inheritor EGY and SEC Controller has to utilize functionality of EGY Controller and extend it by new SEC Controller functionality. Fig. 9 a) shows inheritance relations between roles from IR_{EGY} and $IR_{EGYandSEC}$. Each role from parent specification IR_{EGY} has a child role. So, all provided interfaces and required relations are inherited by product EGY and SEC Controller. The part IR^{New} of interface-role specification $IR_{EGYandSEC}$ is shown in Fig. 9 b). New functionality is realized by three new interfaces of the child roles.

Child roles (R^q)	Parent roles (R^p)	Inherited interfaces I^p
EGY&SEC SatelComputer	Satellite Computer	IController Data Responce
EGY&SEC MeasureControl	EGY Measurement Control	IController Commands
EGY&SEC Data Acquisition	EGY Data Acquisition	IControl EGYData Acquisition
EGY&SEC Data Management	EGY Data Management	IControl File Management ISaved EGYData

a)

Roles-requirers (Rd)	Roles-providers (Rp)	Interfaces (I)		
		Names of interfaces	Operations (Op_i)	Result values (Res_{op})
EGY&SEC Data Manag.	EGY&SEC SatelComputer	IBufferFull	BufferFull	void
EGY&SEC Measurement Control	EGY&SEC Data Acquisition	IControl SECData Acquisition	StartSEC Measurement	true
EGY&SEC MeasureControl	EGY&SEC Data Manag.	ISaved SECData	SendSECData (structure)	void

b)

Figure 9: a) Inheritance of roles and b) IR^{New} part of EGY and SEC Controller specification

The interface-role diagram of EGY and SEC Controller is shown in Fig. 10.

Third product SEC Controller inherits the second one. The interface-role specification of EGY and SEC Controller already contains the functionality required for the third product. A designer is free not to utilized by SEC Controller part of this functionality dealing with EGY data acquisition.

Products-inheritors keep functionality of their predecessors within inherited required relations. However, how can a designer be aware that parent behaviour is not damaged by new design decisions widening or narrowing parent functionality? Such decisions should be supported by product behaviour inheritance modelling, which we consider next.

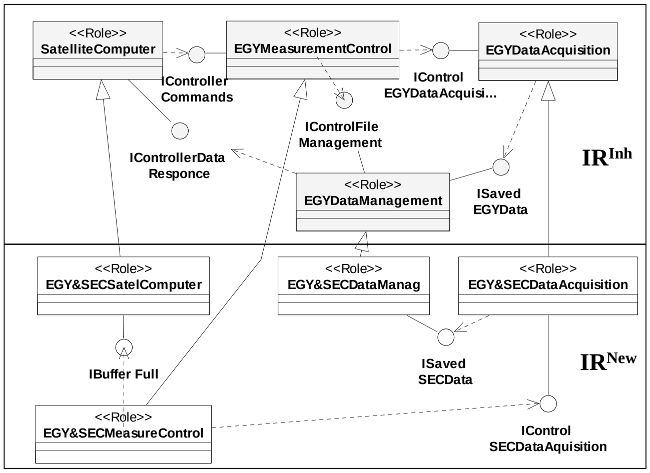


Figure 10: Interface-role diagram of EGY and SEC Controller

4.2.2 Inheritance of product behaviour

To define inheritance of product behaviour we apply process semantics on behaviour specifications BS. We use a process semantics of type

$$P = (A, \mathcal{P}, T) \text{ [2]}, where:$$

- A is a finite set of actions.

- $\mathcal{P} = \{p, p_1, p_2, ..., p_F\}$ is a finite set of abstract states from initial state p to final state p_F.

- T is a set of transitions. Transition $t \in T$ defines a pair of states (p', p''), such that p'' is reachable from p' as a result of action $a \in A$: $p' \overset{a}{\Longrightarrow} p''$.

Considering set of actions A as set A_{PrSp} from a product line member specification, we construct a single *process graph* for the entire product behaviour specification.

Process graph $G_p = (N, E)$ is a directed (cyclic or acyclic) graph [1] in which

- each node n∈N corresponds to the state from $\mathcal{P}$; all nodes, except the root and the final nodes, are unnamed;
- each edge e∈E corresponds to the action from A_{PrSp} and is named as this action;
- the edges may carry the termination label ↓ to one **final** node. This node corresponds to states p_F.
- The process graph has one common root in **start** node that corresponds to initial states p. Each initial state p is considered as a result of *start* action that creates instances of interacting roles [22]. Action *start* is implicit but not shown in the process graph.

Process graph (Fig. 11) keeps parallel branches containing alternatives of sequential, probably cyclic, paths between **start** and **final** nodes. Each such a finite sequential path corresponds to sequence S_i from product behaviour specification BS. Two or several sequences beginning from the same action and containing the same subsequence of actions correspond to a single sequential sub-path in the process graph beginning from **start** node. First two actions that become different for two sequences running the same sub-path produce alternative edges in the process graph. Parallel branches model parallel processes. These branches are the alternatives, which begin from **start** node and, in addition, each pair of them corresponds to the subsets of sequences from BS, which have disjoint sets of actions and are not started by same roles [21].

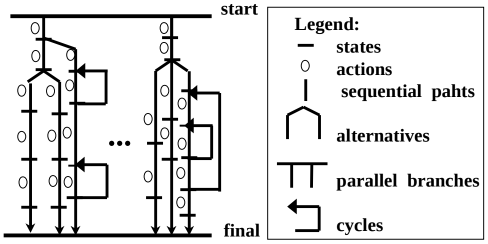


Figure 11: Process graph type

For process graph construction we apply our own algorithm. The algorithm provides control of crosscutting cycles which may be designed by mistake for a single sequence or produced during the process graph construction. The early alternative exit from a cycle body is not prohibited for the process of type P.

The process graph for EGY Controller is shown in Fig. 13 a). It contains the only sequential path that corresponds to single sequence $EGYObservation$ from BS_{EGY} specification.

Behaviour specification $BS_{EGYandSEC}$ for EGY and SEC Controller

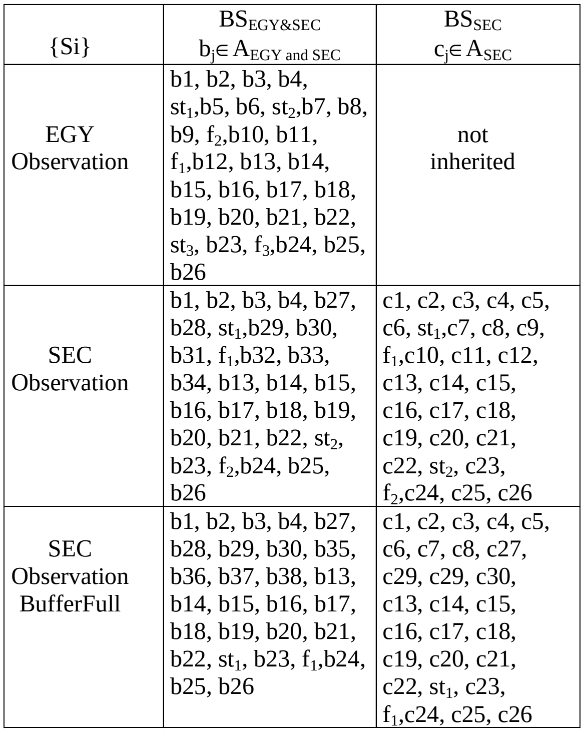
$BS_{EGYandSEC} = \{EGYObservation, SECObservation,$
$SECObservationBufferFull\}$

contains three sequences realizing the requirements to the behaviour of second product. These requirements have been described in section 3.2.

Sequence $EGYObservation$ fulfils the same behaviour pattern as the sequence from BS_{EGY} specification. However, inherited required relations are realized by new roles and, therefore, actions from the second product behaviour specification (Fig. 12) have different names, for example,
b1 = EGY&SECSatelComputer.EGY&SECMeasureControl.Analog_ON
instead of
a1 = SatelliteComputer.EGYMeasurementControl.Analog_ON
and so on to actions $b26$ and $a26$ correspondingly (compare Fig.7 and Fig.12).

Sequence $SECObservation$ models the conventional SEC mode measurement process, whereas sequence

SECObservation BufferFull corresponds to Buffer Full event in the system (section 3.2).



{Si}	$BS_{EGY\&SEC}$ $b_j \in A_{EGY\ and\ SEC}$	BS_{SEC} $c_j \in A_{SEC}$
EGY Observation	b1, b2, b3, b4, st_1,b5, b6, st_2,b7, b8, b9, f_2,b10, b11, f_1,b12, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_3, b23, f_3,b24, b25, b26	not inherited
SEC Observation	b1, b2, b3, b4, b27, b28, st_1,b29, b30, b31, f_1,b32, b33, b34, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_2, b23, f_2,b24, b25, b26	c1, c2, c3, c4, c5, c6, st_1,c7, c8, c9, f_1,c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, st_2, c23, f_2,c24, c25, c26
SEC Observation BufferFull	b1, b2, b3, b4, b27, b28, b29, b30, b35, b36, b37, b38, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_1, b23, f_1,b24, b25, b26	c1, c2, c3, c4, c5, c6, c7, c8, c27, c29, c29, c30, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, st_1, c23, f_1,c24, c25, c26

Figure 12: Behavioural specifications $BS_{EGYandSEC}$ for EGY and SEC Controller and BS_{SEC} for SEC Controller

The corresponding process graph for EGY and SEC Controller is shown in Fig. 13 b). It contains three possible sequential paths from **start** to **final** node. These three paths correspond to three sequences in $BS_{EGYandSEC}$ specification (Fig. 12).

Behaviour specification BS_{SEC} for SEC Controller

$$BS_{SEC} = \{SECObservation, SECObservationBufferFull\}$$

contains two sequences, which comprise exactly the same operations as ones for EGY and SEC Controller (Fig. 12). However, corresponding actions have different names. The process graph for SEC Controller is shown in Fig. 13 c). It contains two sequential paths corresponding two sequences from BS_{SEC}. Sequence *EGYObservation* is not utilized.

As a result of inheritance of interface-role specifications action set A_{PrSp_q} of the inheritor contains two subsets:

$$A_{PrSp_q} = A^{New}_{PrSp_q} \cup A^{Old}_{PrSp_q}; A^{New}_{PrSp_q} \cap A^{Old}_{PrSp_q} = \emptyset, where$$

- $A^{Old}_{PrSp_q}$ is a subset of actions, which are realized by *inherited required relations from* IR^{Inh}_q;
- $A^{New}_{PrSp_q}$ is a subset of actions, which are realized by *newly designed required relations from* IR^{New}_q.

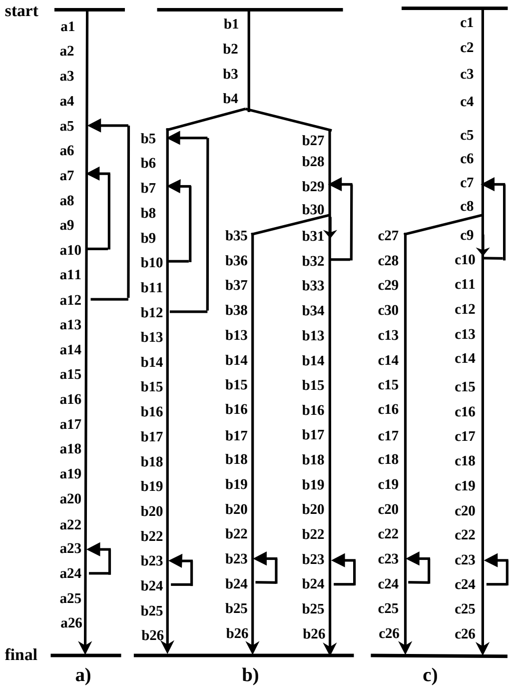
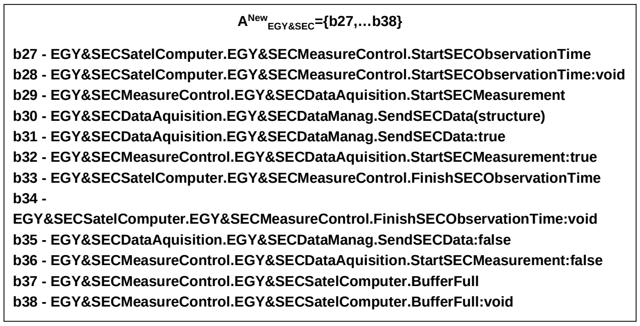


Figure 13: Process graphs for a) EGY Controller; b) EGY and SEC Controller; c) SEC Controller

For example, EGY and SEC Controller has subset $A^{Old}_{EGY\&SEC}$ $= \{b1, b2, ...b26\}$ and subset $A^{New}_{EGY\&SEC}$ of new actions presented in Fig. 14.



$A^{New}_{EGY\&SEC}$={b27,...b38}

b27 - EGY&SECSatelComputer.EGY&SECMeasureControl.StartSECObservationTime
b28 - EGY&SECSatelComputer.EGY&SECMeasureControl.StartSECObservationTime:void
b29 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement
b30 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData(structure)
b31 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData:true
b32 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement:true
b33 - EGY&SECSatelComputer.EGY&SECMeasureControl.FinishSECObservationTime
b34 - EGY&SECSatelComputer.EGY&SECMeasureControl.FinishSECObservationTime:void
b35 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData:false
b36 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement:false
b37 - EGY&SECMeasureControl.EGY&SECSatelComputer.BufferFull
b38 - EGY&SECMeasureControl.EGY&SECSatelComputer.BufferFull:void

Figure 14: Subset of new actions for EGY and SEC Controller

Now let us give the definition of correct product behaviour inheritance.

Firstly, we define *renaming function RN*, which we apply on parent set of actions A_{PrSp_p} producing subsets of inherited $A^{Inh}_{PrSp_p}$ and not inherited $A^{not\_Inh}_{PrSp_p}$ parent actions:

$$A^{Inh}_{PrSp_p} \cup A^{not\_Inh}_{PrSp_p} = RN(A_{PrSp_p}); A^{Inh}_{PrSp_p} \cap A^{not\_Inh}_{PrSp_p} = \emptyset$$

such that $A^{Inh}_{PrSp_p} = A^{Old}_{PrSp_q}$.

For example, $RN(A_{EGY}) = A^{Inh}_{EGY} = A^{Old}_{EGY\&SEC} =$

$\{b1, b2, ..., b26\}$; $A^{not\_Inh}_{EGY} = \emptyset$.

SEC Controller does not inherit from EGY and SEC Controller subset of actions $A^{not\_Inh}_{EGY\&SEC}=$ $\{b5, b6, b7, b8, b9, b10, b11, b12\}$, which corresponds to the specific EGY measurement subsequence from *EGYObservation* sequence (Fig. 12).

Secondly, let us define on graph of type G_p a pair of *graph transformation rules* $\delta(G_p)$ and $\tau(G_p)$.

- *Blocking rule* $\delta(G_p)$. If subset $B \in A_{PrSp}$ is defined and action $x \in B$, action $a \notin B$ and δ is *blocking* action, then process graph G_p is transformed as it follows from Fig. 15 a). This rule allows cutting down alternative branches starting from actions $x \in B$. Applied to a sequential path this rule cuts it down starting from action x but blocking action is not removed [2].
- *Hiding rule* $\tau(G_p)$. If subset $H \in A_{PrSp}$ is defined and action $y \in H$, action $a \notin H$ and τ is *silent* action, then process graph G_p is transformed as it follows from Fig. 15 b). This rule allows shortening sequential branches by means of deleting actions $y \in H$ [2].

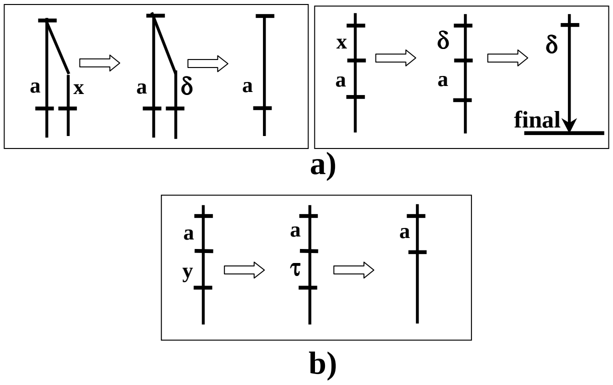


Figure 15: a) $\delta(G_p)$ and b) $\tau(G_p)$ graph transformation rules

Applying process algebra for process of type P [2] on process graph representation we define conditions of *complete* and *partial* inheritance of product specifications.

- Child $PrSp_q$ *completely inherits* parent $PrSp_p$ if and only if $RN(A_{PrSp_p}) = A^{Inh}_{PrSp_p}$ and $A^{not\_Inh}_{PrSp_p} = \emptyset$ and
$$\tau(\delta(G_p^{PrSp_q})) = G_p^{PrSp_p}$$
on condition that
  - the action set of $G_p^{PrSp_p}$ is renamed using function $RN(A_{PrSp_p})$;
  - for the transformation of the child process graph subset B=$A^{New\_Alt}_{PrSp_q}$ and subset H=$A^{New\_Seq}_{PrSp_q}$, where $A^{New\_Alt}_{PrSp_q}$ is a subset of $A^{New}_{PrSp_q}$ containing actions, which *start alternative branches* and $A^{New\_Seq}_{PrSp_q}$ is the rest of $A^{New}_{PrSp_q}$.

In other words, if the parent action set contains only inherited actions we apply the renaming function on the parent set of actions and using the blocking rule eliminate from the child process graph all alternative branches that are started by new actions. Next, we apply the hiding rule and eliminate the rest of new child actions. If the resulting transformed graph is equal to the parent graph with renamed actions, then the child specification is a correct inheritor of the parent specification.

In spite of seemingly tricky notation this definition has clear rationale: alternatives started by new actions will run their own branches to the **final** state (Fig. 11); they will never return to parent behaviour and, therefore, have to be eliminated during parent process graph derivation. New actions running a sequential branch may be hidden to return to parent behaviour within the same branch (sequence).

- Child $PrSp_q$ *partially inherits* parent $PrSp_q$ if and only if $A^{Inh}_{PrSp_p} \neq \emptyset$ and $A^{not\_Inh}_{PrSp_p} \neq \emptyset$ and
$$\tau(\delta(G_p^{PrSp_q})) = \delta(G_p^{PrSp_p})$$
on condition that
  - action set from $PrSp_p$ is renamed using function $RN(A_{PrSp_p})$;
  - for the transformation of the child process graph subset B=$A^{New\_Alt}_{PrSp_q}$ and subset H=$A^{New\_Seq}_{PrSp_q}$, where $A^{New\_Alt}_{PrSp_q}$ is a subset of $A^{New}_{PrSp_q}$ containing actions, which *start alternative branches* and $A^{New\_Seq}_{PrSp_q}$ is the rest of $A^{New}_{PrSp_q}$;
  - for the transformation of the parent process graph subset B=$A^{not\_Inh}_{PrSp_p}$.

In other words, child process graph transformation is the same as that in the case of complete inheritance, but before comparing, the parent process graph is transformed using the blocking rule to eliminate not inherited parent actions and, therefore, corresponding sequences. The hiding rule is not applicable to the parent process graph because hiding means shortening sequences from parent specification BS_p each of those must be inherited completely or not inherited at all.

In our case study EGY and SEC Controller is a correct complete inheritor of EGY Controller. Indeed, if we rename parent actions $\{a1, a2, ..., a26\}$ to $\{b1, b2, ..., b26\}$ and hide and block the new actions from the child set, the child process graph is transformed to the parent one (actually, for such transformation blocking of action b27 in Fig. 13 b) is enough).

SEC Controller is a correct partial inheritor of EGY and SEC Controller. To prove this we need to block not inherited action b5 in Fig. 13 b) and rename the parent inherited actions: b1 to c1 , b2 to c2 and so on (compare graphs in Fig. 13 b) and c)). Graph transformation of the child graph is not required because the specification of the inheritor does not contain new actions.

If a child specification is not a correct inheritor of a parent specification, then transformed child or/and parent process graphs contain not eliminated τ and δ actions. The rest of a sequence (or sequences) starting by such an action becomes unreachable [2]. All these sequences are easily transformed back from the process graph and the positions of τ or/and δ actions show the points of design errors. These errors are actions, which cannot be realized within a given specification. So, the roles performing such impossible actions can be indicated. As a result, the method allows a designer not only to prove correctness of inherited specifications but also to find design bags.

5. TOOL SUPPORT

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 [24]. The tool is implemented as a Rational Rose Add-In [20].

A familiar with Rational Rose designer performs with the help of the tool the following sequential steps:

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 have to 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.

The screen shot of a derivation dialog for EGY and SEC Controller is shown in Fig 16. More details about the tool are contained in [24].

6. CONCLUSION AND FUTURE WORK

The presented method provides evolutionary incremental modelling of software product line members using inheritance of their behaviour specifications. Correctness of model transformations is proved by using a derivation technique that allows a designer to produce the process graph of a product-predecessor from the inheritor's one or to find the points of incorrect design.

An appropriate tool prototype has been developed to support the modelling. The tool applies techniques and algorithms which accompany the method. Robustness of the method and the tool is proved by the modelling of an industrial case study.

In future work we intend to find out how our method applicable to large-scale industrial systems. In this context the problem of product requirements mapping to our specifications needs to be investigated. In large-scale applications such successful direct mapping that we have shown in our case study is not so apparent. A kind of a specifications mapping technique is required. Recent researches (e.g., see in [14]) apply UML use case and scenario diagrams to SPL requirements engineering. In such a case, requirements can be mapped to interface-role specifications directly: actors iterating via use cases can be mapped to roles; use cases itself can be realized as sets of required relations between roles; scenario diagrams can be considered as prototypes of sequence diagrams.

Mapping between our specifications and product component architectures is also a significant problem. Component systems are usually described in Architecture Description Languages (ADLs) (see good overview [15]). Most of them allow representing roles and interfaces as components and connectors. Among others, ADLs with strong component evolution support, such as Koala [18], are more close to our approach. Moreover, Koala is a good practical example of an ADL for component-based product population development. Our specifications can be mapped to Koala's configurations in such a manner that roles would correspond to components. Provided and required relations can be presented by Koala's provides and requires interfaces. Compositional capacity of a Koala component (combinations of components are components again [18]) 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.

7. REFERENCES

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

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

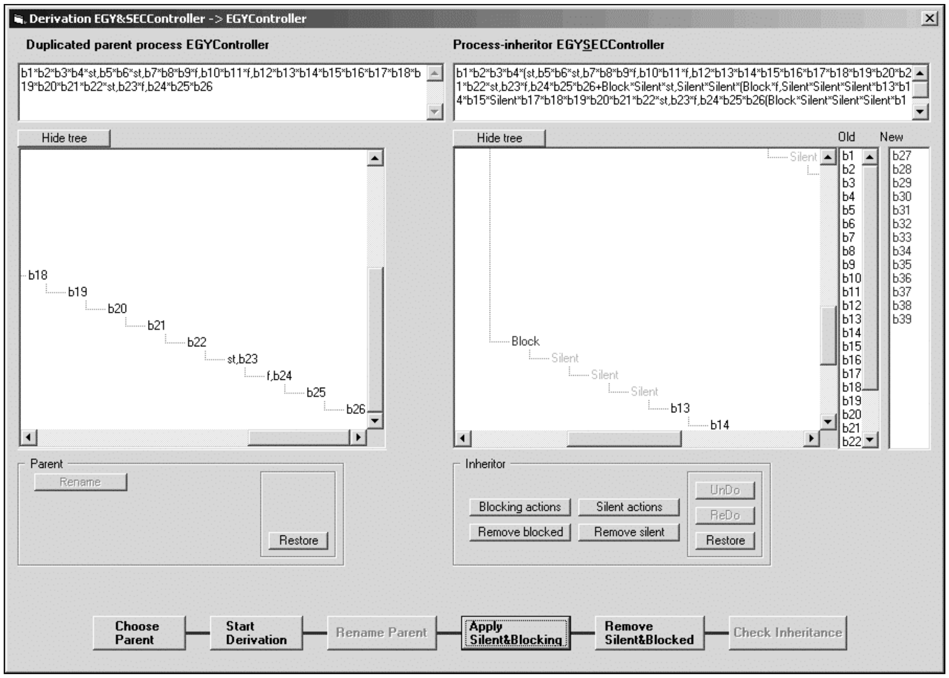
[3] Becker M. Towards a General Model of Variability in Product Families. *Workshop on Software Variability Management. Editors Jilles van Gurp and Jan Bosch. Groningen, The Netherlands. http://www.cs.rug.nl/Research/SE /svm/proceedingsSVM2003Groningen.pdf*, pages 19–27, 2003.

[4] Bosch J. *Design&Reuse of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[5] Bosch J. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *Second Conference Software Product Line Conference, SPLC2*, August 2002.

[6] Bosch J., M. Svahnberg and J. van Gurp. On the notion of variability in software product lines. In *Software Architecture. Working IEEE/IFIP Conference*, pages 45–54, 2001.

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



Derivation EGY&SECController -> EGYController

Duplicated parent process EGYController

b1*b2*b3*b4*st,b5*b6*st,b7*b8*b9*f,b10*b11*f,b12*b13*b14*b15*b16*b17*b18*b19*b20*b21*b22*st,b23*f,b24*b25*b26

Process-inheritor EGYSECController

b1*b2*b3*b4*(st,b5*b6*st,b7*b8*b9*f,b10*b11*f,b12*b13*b14*b15*b16*b17*b18*b19*b20*b21*b22*st,b23*f,b24*b25*b26+Block*Silent*st,Silent*Silent*(Block*f,Silent*Silent*Silent*b13*b14*b15*Silent*b17*b18*b19*b20*b21*b22*st,b23*f,b24*b25*b26(Block*Silent*Silent*Silent*b1

Figure 16: Parent process derivation dialog in the tool

[8] P. Clements and R. Northrop. *Software Product Lines - Practices and Patterns*. Pearson Education (Addison-Wesley), ISBN 0-201-30977-7, 2000.

[9] Dobrica L.,E.Niemela. *A strategy for analysis product line software architectures*. VTT Technical Research Centre of Finland, ISBN 951-38-5599-6, 2000.

[10] P. Donohoe, editor. *Software Product Lines - Experience and Research Directions*. Kluwer Academic Publishers, 2000.

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

[12] Gurp J. van , J.Bosch. Design Erosion: Problems and Causes. *Journal of Systems and Software, 61(2), Elsevier*, 61:105–119, 2002.

[13] Ihme T. A ROOM Framework for the Spectrometer Controller Product Line. *Workshop on Object Technology for Product Line Architecture*, pages 119–128, ESI–199–TR–034, 1999.

[14] MacGregor J. Requirements Engineering in Industrial Product Lines. In *International Workshop on Requirements Engineering for Product Lines, REPL'02*, pages 5–11, Essen, Germany, 2002.

[15] N. Medvidovic and R.Taylor. A classification and comparison framework for software architecture description languages. Technical report, USC Center for Software Engineering http://sunset.usc.edu/ neno/papers/TSE-ADL.pdf.

[16] OMG. *Unified Modeling Language Specification v.1.3, ad/99-06-10 http://www.rational.com/ uml/resources/documentation/index.jsp*, June 1999.

[17] R. van Ommering. Roadmapping a Product Population Architecture. *Workshop on Product Family Engineering, Bilbao, Spane*, 2001.

[18] R. van Ommering, F. van der Linden, J. Kramer, J.Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, pages p78–85, March 2000.

[19] R. van Ommering, J.Bosch. Widening the Scope of Software Product Lines - From Variation to Composition. In *Second Conference Software Product Line Conference, SPLC2*, pages 328–347, August 2002.

[20] Rational Rose 98i. *Rose Extensibility Reference 2000.* http:// www.rational. comwww.se. fh-heilbronn.de/ usefulstuff/ Rational Rose 98i Documentation.

[21] Roubtsov S.A., E.E.Roubtsova. Modeling Evolution and Variability of Software Product Lines Using Interface Suites. *Workshop on Software Variability Management. Editors Jilles van Gurp and Jan Bosch. Groningen, The Netherlands. http://www.cs.rug.nl/Research/SE /svm/proceedingsSVM2003Groningen.pdf*, pages 62–71, 2003.

[22] Roubtsova E. and R. Kuiper. Process Semantics for UML Component Specifications to Assess Inheritance. *Electronic Notes in Theoretical Computer Science, 72,3 Elsevier Science Publishers, Paolo Bottoni and Mark Minas, http://www.elsevier.nl/gej-ng/31/29/23/127/48/show/Products/notes/index.htt*, 2003.

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

[24] Roubtsova E.E., S.A.Roubtsov. UML-based Tool for Constructing Component Systems via Component Behaviour Inheritance. *Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03) To appear in Elsevier Electronic Notes in Theoretical Computer Science, 80 (2003) http://www.elsevier.nl/locate/entcs/volume80.html* , pages 139–154, 2003.

[25] Svahnberg M., Bosch J. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance: Research and Practice, Vol. 11, No. 6*, 1999.

[26] Szyperski C. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, New-York, 1998.

[27] W.Eixelsberger, M.Ogris, H.Gall, and B.Bellay. Software recovery of a program family. *International conference on Software Engineering, Kyoto, Japan*, 1998.

[28] Zhao L., Kendall E. Role Modelling for Component Design. *The 33rd Hawaii International Conference on System Science*, 2000.