Software Variability Management

Workshop, February 13th & 14th 2003

Proceedings

Editors: Jilles van Gurp & Jan Bosch

This workshop is sponsored by:

The European Science Foundation (ESF)
The ESF RELEASE network
Philips Medical Systems
The IST CONIPF project
The RUG Institute of Mathematics and Computing Science

Table of Contents

Preface

Welcome to the 1st workshop on Software Variability Management. During this workshop we hope that interesting discussion takes place. We have received a number of very interesting long papers, which are included in these proceedings, as well as a number of short papers that may be used as a starting point for the discussions. About two thirds of the workshop is dedicated to the paper presentations. However, there will be ample opportunity for discussion during the breaks lunches and dinner. Also we hope that the paper presentations themselves will spark discussion.

What is variability

Most modern software needs to support increasing amounts of variability, i.e. locations in the software where behavior can be configured. This trend leads to a situation where the complexity of managing the amount of variability becomes a primary concern that needs to be addressed. Two causes for the increasing amount of variability are the delaying of design decisions to the latest point that is economically feasible and the transfer of variability from mechanics and hardware to the software in embedded systems. Examples of the first category include software product families, the configuration wizards and tools in most commercial software, the configuration interface of software components in component-based software engineering and even the dynamic, run-time composition of web-services. Examples of the second category can be found in many embedded systems, including car electronics, telecommunications and consumer electronics.

Software variability is the ability of a software system or artifact to be changed, customized or configured for use in a particular context. A high degree of variability allows the use of software in a broader range of contexts, i.e. the software is more reusable. Variability can be viewed as consisting of two dimensions, i.e. space and time. The space dimension is concerned with the use of software in multiple contexts, e.g. multiple products in a software product family. The time dimension is concerned with the ability of software to support evolution and changing requirements in its various contexts.

The reason for identifying software variability management as a core topic is twofold. First, within the software engineering research community, we have come to realize that the fundamental issue in a range of reuse approaches, including object-oriented frameworks, component-based software engineering and software product families, is the management of the provided variability. Basically, the reusability of any software artifact is determined by its ability to support the variability required from it. Second, in several industrial organizations, the complexity of variability management is becoming such that more systematic approaches are required as the limitations of ad-hoc approaches experienced daily. For instance, the number of variation points for industrial software product families may range in the thousands.

Workshop Program

We have put together a very interesting program covering a wide variety of variability related topics. We are very pleased to have attracted so much response from the research community. On day one of the workshop, there will be two plenary sessions where the authors will present their papers. The format of the presentations will be a short 10 to 15 minutes presentation followed by discussion. The second half of the day is reserved for workgroup sessions. On day two we will start with two more plenary sessions. After that we will discuss the results from the workgroup sessions on day 1.

February 13th

9.00 - 9.30	Workshop Registration	
9.30 - 10.00	Introduction by Jan Bosch	
10.00 - 11.25	Plenary Session 1	
	1	B. Tekinerdogan, M. Aksit, "Managing Variability in Product Line Scoping"
	2	K. Schmid, I. John, "Generic Variability Management and Its Application to Product Line Modelling"
	3	M. Becker, "Towards a General Model of Variability in Product Families"
11.25 - 11.30	Coffee break	
11.30 - 12.45	Plenary Session 2	
	4	A. Maccari, A. Heie, "Managing Infinite Variability"
	5	K. Koskimies, "Supporting Variability Management in XML"
	6	S. Demeyer, "Extensibility via a Meta-level Architecture"
12.45 - 14.00	Lunch	
14.00	Workgroup sessions	
18.00	Dinner	

February 14th

8.45 - 9.00	Start of day 2	
9.00 - 10.15	Plenary Session 3	
	7	T. Weiler, "Modelling Architectural Variability for Software Product Lines"
	8	S.A. Roubtsov, E.E. Roubtsova, "Modeling Evolution and Variability of Software Product Lines Using Interface Suites"
	9	D. Beuche, H. Papajewski, "Variability Management with Feature Models"
10.15 - 10.30	Coffee	
10.30 - 11.20	Plenary Session 4	
	10	T. Asikainen, T. Soininen, T. Männistö, "Towards Managing Variability using Software Product Family Architecture Models and Product Configurators"
	11	T. Ziadi, J. M. Jézéquel, F. Fondement, "Product Line Derivation with UML"
11.20 - 12.15	Reports from workgroups	
12.15 - 12.30	Conclusion	
12.30	Lunch	
	End of Workshop	
15.45	Ph.D. defense Jilles van Gurp	

Conclusion

We hope that you will enjoy the workshop very much. As you may know we are also involved in a special issue of Elsevier's Science of Computer Programming on Software Variability Management. We may invite individual authors to submit their paper. In addition, we would like to draw your attention to the upcoming ICSE workshop on Software Variability Management.

Editors:

Jilles van Gurp
Jan Bosch

Full Papers

Managing Variability in Product Line Scoping using Design Space Models

Bedir Tekinerdoğan
Dept. of Computer Engineering,
Bilkent University,
Bilkent 06800, Ankara, Turkey
bedir@cs.bilkent.edu.tr

Mehmet Akşit
TRESE Software Engineering,
Dept. of Computer Science, University of Twente, P.O. Box 217, 7500 AE,
Enschede, The Netherlands
aksit@cs.utwente.nl

Abstract

Product-line engineering aims to reduce the cost of manufacturing of software products by exploiting their common properties. Obviously, to define a product line, the product alternatives that need to be produced must be identified first. This is generally realized either by a product requirements analysis or a domain analysis process. Product requirements analysis focuses on specific products or product characteristics and therefore may fail short to identify those products that are not explicitly stated in the product requirements. Domain models on the other hand are inherently too abstract to identify the product alternatives and reason about these explicitly. To provide a balanced scoping we propose to integrate both approaches and present the so-called design space models (DSMs) as a complementary technique to existing product line scoping techniques. We explain our ideas using an illustrative example for scoping the product-line of insurance systems.

1. Introduction

Product-line engineering aims to reduce the costs of manufacturing of software products by exploiting their common properties and by managing the variabilities [1]. Obviously, to define a product line, the product alternatives that need to be produced must be identified first. A core activity of software product line development is therefore product line scoping, which seeks to define the right set of product alternatives.

An often-used approach for product line scoping is to define a domain model that includes reusable assets to configure the products. The advantage of adopting a domain model is that it is general enough to represent a large set of products. Due to this character, however, it may be difficult to identify and derive specific products from it. To tackle this problem, product requirement analysis techniques can be used in which the specific products and their characteristics are explicitly specified [6]. This provides a concrete product-line scope but may fail in short in identifying those products that are not explicitly stated in the product requirements.

It appears that the adoption of only domain analysis or product requirements analysis techniques is not sufficient to define the right product line scope. In spite of this, both approaches are not sufficiently integrated yet. Although there are some approaches that aim to scope the domain model by considering the product requirements [6][7], their main focus is on the scoping process rather than deriving product alternatives. Approaches that mainly focus on product requirements on the other hand, however, can be too restrictive, because they may not cover a sufficient set of product alternatives. Moreover, both approaches usually do not address the implementation aspects of the products. Products may be implemented in various different ways and different implementations of the product may behave differently with respect to the aimed quality factors, such as adaptability and performance. It may, for example, appear that several implementation alternatives are not required or even not possible and therefore need to be ruled out. Other implementation alternatives may be favorable by the stakeholders due to some implementation specific requirements such as the choice of the platform, the implementation language, or quality criteria such as adaptability and performance. We therefore believe that in addition to the product line (specification) scoping, the *product line implementation scoping (PLIS)* is needed as well.

In this paper a systematic product line scoping approach is presented in which the products are gradually derived from the abstract domain models based on the specific product requirements. To represent the product line scope the concept of *Design Space Models (DSMs)* is introduced. Design spaces represent a set of alternatives for a given design problem. Design space modeling as such consists of representing a design space and defining the semantic information for configuring and depicting the selection and elimination of alternatives within that space. For product line scoping we represent domain models as design space models, define the constraints and reduce the set of product alternatives with respect to the corresponding product requirements using the operations that we have defined for design space models. For product line implementation scoping the domain model is mapped to a design space that includes the set of possible implementation alternatives, and which can be reduced again with respect to the product requirements and the corresponding analysis and design heuristics. The utilization of design space models in product line scoping results not only in a more precise product line scope but also supports the reasoning on the product alternatives.

We will illustrate our ideas using an example from a real industrial project in which we have defined both the product line specification scope and the product line implementation scope using the design space modeling. Hereby we will also illustrate the tool environment *Rumi* that includes a set of tools for supporting the techniques of design space modeling.

The outline of the paper is as follows. In the following section we will describe the problem statement and describe the example from a real industrial project on the scoping of a product line for insurance systems. In section 3, the concept of design space models and its application to product line scoping is described in more detail. Section 4 describes the related work and section 5 provides the conclusions.

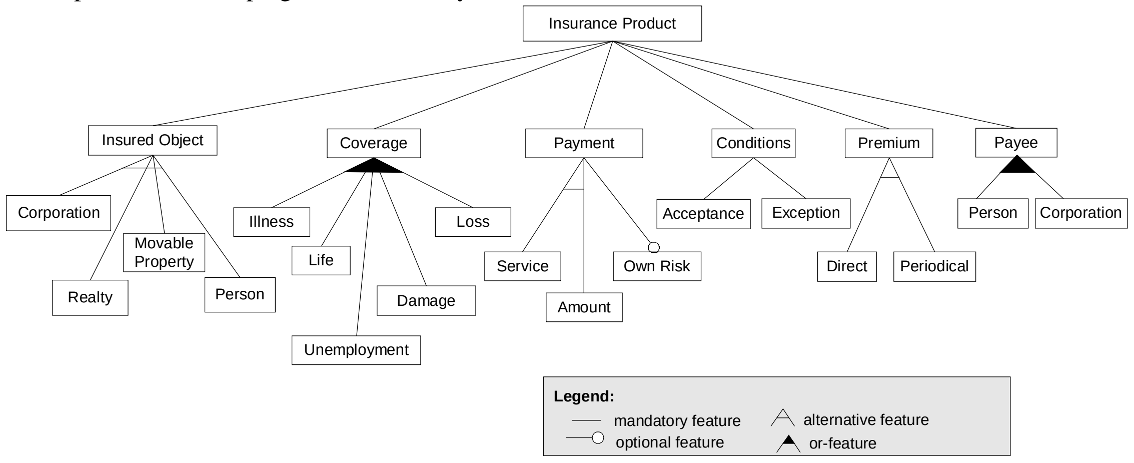


Figure 1. (Top-level) feature model for a product family of insurance systems

2. Problem Statement

2.1 Example: Domain Model of insurance systems

In the following section, we will describe a real world design example, which was developed in an industrial project between our faculty and a software company[1]. The goal of the project was to develop a software product-line for insurance systems. Over the years, the software company has developed an increasing number of insurance systems, whereby each system was practically developed from scratch. This resulted in unnecessary repeating similar design and coding efforts. To save costs, a software product line approach for insurance systems was launched. The fundamental challenge hereby was the decision on the set of products that were to be delivered, i.e. the product line scope.

Numerous and various insurance systems exist, which share some common features that can be exploited for reuse [12]. Figure 1 shows the feature model of a product-line, which was defined through an extensive domain analysis effort. Each insurance product consists of the following (mandatory) sub-concepts (features): *Insured Object*, *Coverage*, *Payment*, *Conditions*, *Premium* and *Payee*. An insured object can either be a person, a corporation,

[1] This project has been carried out together with Utopics, The Netherlands [12].

realty or some moveable property. The feature *Coverage* defines the risk that is to be insured, which can be either risks of *Illness*, *Life*, *Unemployment*, *Damage* or *Loss*. The feature *Payment* includes the mandatory features for the approach of payment and an optional own-risk feature. The feature *Conditions* includes the acceptance and exception conditions for the insurance. *Premium* defines the approach of payment of the premium. Finally, *Payee* defines the features that will benefit in case of the occurrence of the risk that is insured. This feature model defines a product family of insurance systems from which a broad set of insurance products can be derived.

2.2 Problem Description

2.2.1 Balanced Product Line Scoping

A domain model is an intentional representation of the products in the domain in the sense that it specifies the product alternatives in an implicit way. A product is derived from a set of domain concepts and as a composition of domain instances. Not all products that can be derived from the domain model, however, are usually interesting. To reason about the relevance of each product the product alternatives must be derived from the domain model and represented in an explicit way. Enumerating the individual products in the product line and the individual requirements relevant to the products [8], however, may become cumbersome because of the large size of the domain. On the one hand the domain model must be expressive enough to support a large set of product alternatives, on the other hand the combinatorial overhead of the broad set of irrelevant product alternatives must be avoided. In the given example we would be interested in the possible set of insurance systems, and would like to depict these to reason about them explicitly. Finding the balance between an intentional and an extensional representation, however is not trivial.

2.2.2 Scoping Product Implementation Alternatives

Product models are generally derived from more abstract domain models, and can be implemented in many different ways. Similar to the fact that a domain model may express a broad set of products, a product model may also express a broad set of product implementations. After having selected the set of products from the domain models one may choose to implement these using, for example, object-oriented abstractions. The various object-oriented abstractions enable the software engineer to derive different implementation alternatives for the same product and each implementation may, for example, display different quality characteristics. To explain this in more detail consider Figure 2 that depicts, for example, three different implementation alternatives that can be derived from the domain model. In the design alternative of Figure 2a, the concept *InsuranceProduct* of Figure 1 has been mapped to a class *InsuranceProduct* and the sub-concepts have been mapped to the operations *insuredObject()*, *coverage()*, *payment()*, *conditions()*, *premium()* and *payee()*. This means that the various instances of the sub-products are all hidden in the implementation of the corresponding operations. In Figure 2b each sub-concept has been mapped to a class, which are encapsulated by the class *InsuranceProduct*. Finally, Figure 2c shows another alternative whereby for each type of *InsuredObject* a separate class is defined that includes the other subconcepts as operations.

These three implementations are not the only alternatives and actually a considerable number of implementation alternatives may be derived from the same product. We may use a separate class for each sub-concept, define these as abstract methods, map these to single methods etc. Currently, appropriate techniques for systematically identifying and describing the possible product implementation alternatives, is unfortunately missing.

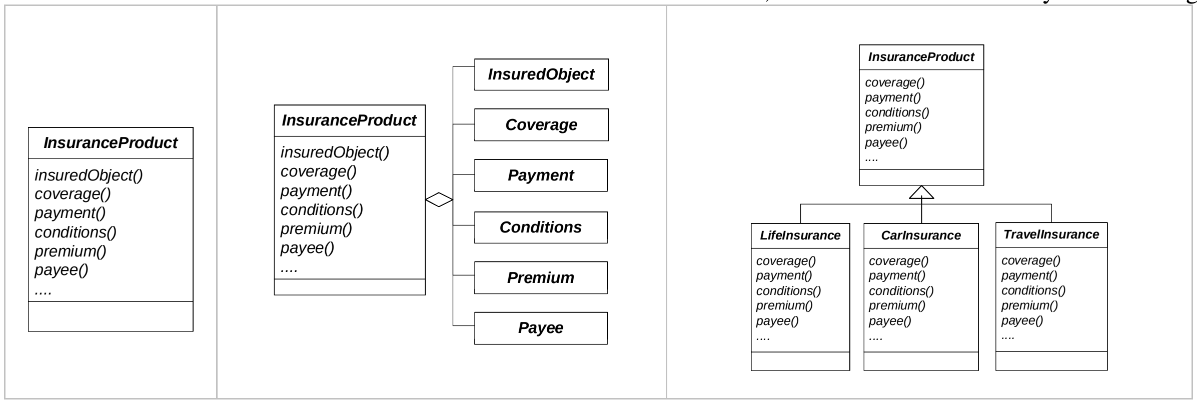


Figure 2. *Three different implementation alternatives of InsuranceProduct*

3. Utilizing Design Space Models

To provide solutions for the problems as defined in the previous section, we propose *design space models* for supporting product line scoping processes. Informally, design spaces represent a set of alternatives for a given design problem. Design space modeling consists of representing a design space and defining the semantic information for configuring and depicting the selection and elimination of alternatives. By representing the domain model as an explicit design space and by providing operations for combining and reducing design spaces, the product line scoping can be defined more precisely. In the following we represent the process for scoping the product line from the product specification to the product implementation levels:

1. *Representing Design Spaces*

The domain analysis process will result in a domain model that represents both the commonality and the variability of the set of products that need to be included in the product-line scope. We will describe the domain model using *design algebra*, which provides a formal representation to define the set of alternatives of a given domain. This is explained in section 3.1.

2. *Defining constraints of alternatives.*

The next step will be to defining the set of rules for identification of valid and invalid alternatives within the specified domain model. This is specified similar to the composition rules as defined in [3]. These constraints will be utilized to eliminate the alternatives within the domain model that are not viable. This is explained in section 3.2.

3. *Unfolding design spaces*

To reason about individual alternatives an extensional view of DSMs will be given. This is supported by the operation *unfold()* in design algebra and implemented in the tools of *Rumi*. The *unfold* operation will derive all the possible alternatives from the design space. This is explained in section 3.3.

4. *Reducing design space*

Because the set of alternatives may be too large, design algebra includes selection and elimination operations to reduce the design space. This is explained in section 3.4.

5. *Mapping design space to implementation domain*

Once the product line scope has been defined, the implementation alternatives of each product in the product line will be considered. For this, the product line will be mapped to the implementation domain, which will consequently result in a new design space. The product implementation space will be reduced with heuristics and constraints. This is explained in section 3.5.

3.1 Representing Design Spaces

Before reasoning about the individual alternatives we will represent the domain model using the concept of *design spaces* as supported by the formalism called *design algebra*. A ***design space*** in this context is defined as a multi-dimensional space from which the set of alternatives for a given design problem can be derived. The design space is spanned by an independent set of ***dimensions.*** We define a dimension as a mandatory feature of a concept. As such the dimensions of `InsuranceProduct` are the sub-features `InsuredObject`, `Coverage`, `Payment`, `Conditions`, `Premium`, and `Payee`. The set of dimensions of a concept is defined as its ***dimension set***. In design algebra, we define the model of `InsuranceProduct` of Figure 1 as follows:

```
InsuranceProduct = (InsObj ∧ Cov ∧ Paym ∧
Cond ∧ Prem ∧ Payee)
```

Here, `InsObj`, `Cov`, `Paym`, `Cond`, `Prem`, `Payee`, represent the features InsuredObject, Coverage, Payment, Conditions, Premium, and Payee respectively. The symbol '∧' defines the composition relation in the feature diagram. A design space for `InsuranceProduct` consists of 6 dimensions that are represented by these features. To be able to reason about the alternatives we introduce the concept of ***coordinate***. We define a coordinate as a sub-feature of a dimension. The set of coordinates of a dimension are defined as the ***coordinate set*** of each dimension. The coordinates of a dimension may be a mandatory feature, alternative feature, optional feature or an or-feature [3]. These different feature properties are represented using the following symbols:

∧ mandatory	; alternative
∨ or	? optional

In the example, the dimension `InsuredObject` includes the coordinates `Corporation`, `Realty`, `Moveable Property` and `Person`. We represent this as follows:

```
InsuredObject = (Corporation; Realty;
MoveableProperty; Person)
```

This indicates that only one of them can be selected. In design algebra we also use symbols to express the other feature properties. Consider for example the concept `Coverage` that is expressed as follows:

```
Coverage = (Illness ∨ Life ∨ Unemployment ∨
Loss ∨ Damage)
```

In this case for `Coverage` either `Illness`, `Life`, `Unemployment`, `Loss` or `Damage` can be selected. The concept `Payment` is represented as follows:

```
Payment = ((Service; Amount) ∧ OwnRisk?)
```

The tool environment *Rumi* includes tools for defining features but we will not present these due to space limitations.

3.2 Defining Constraints

Similar to composition relations [3] in feature models we adopt constraints to express the constraints between various features in the model. These constraints define the semantics between features that are not expressed in the feature diagram. Basically we apply the *mutex-with* and *requires* composition rules. The *mutex-with* rule defines a mutual exclusion relation between two concepts or features, whereas the *requires* rule defines which features the selected feature requires (interdependent relations). In the insurance product systems, for example, we may identify the following set of constraints (the symbol '.' is used to denote the bindings):

1. `InsuredObject.Person` **`mutex-with`** `Coverage.Damage`

   If the ensured object is a person then the insurance product cannot include coverage of damage (for physical objects)

2. `Coverage.Loss` **`requires`** `InsuredObject.MoveableProperty`

   If the insurance product includes coverage for loss then the insured object can only be a moveable property

3. `Coverage.Illness` **`mutex`** `InsuredObject.Corporation`

   If the insurance product includes coverage for illness then the insured object cannot be a person.

4. `InsuredObject.Corporation` **`requires`** `Payee.Corporation`

   If the insured object is a corporation then the claimer should also be a corporation.

Besides of these constraints from the domain also constraints imposed by stakeholders can be defined in a similar way. In *Rumi* these can be defined, updated and eliminated using various tools during the scoping process.

3.3 Unfolding Domain Model

Once the domain model, the corresponding feature models and the constraints have been defined we need to derive the corresponding alternatives. For this, in design algebra the operation *unfold* is applied, which results in the total set of alternatives that can be derived from the given feature model. The model `InsuranceProductScope` in the following specification defines all the product alternatives that can be derived from `InsuranceProduct`:

```
InsuranceProductScope:=
InsuranceProduct.unfold()
```

An alternative is defined by binding the variant features (optional-feature, or-feature, alternative-feature) to the dimensions of the model. For example, based on the feature model in Figure 1 we can bind four alternative features to the sub-concept `InsuredObject`. The sub-concept `Coverage` can be bound in 2^5-1 or 31 ways. The sub-concept `Payment` can be bound in 4 ways (two alternatives and one optional feature). For sub-concept `Conditions` we can bind features in one way since its both features are mandatory. Finally, `Premium` can be bound in 2 ways, and `Payee` in 2^2-1 = 3 ways.

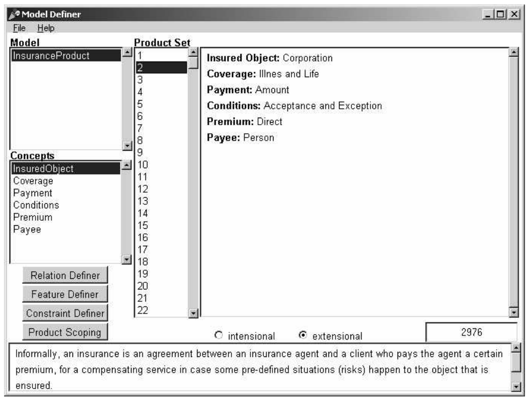
The total set of alternatives that that can be derived from this (simplified) feature diagram is thus 4x31x4x1x2x3 = 2976 alternatives. In design algebra we provide the operation `numAlternatives()` to automatically compute the number of product alternatives from a given domain model:

```
InsuranceProductScope.numAlternative()
```

For example one of these 2976 product alternatives is the following health insurance product that covers illness with own risk and a direct premium:

```
(InsObj.Person ∧ Cov.Illness ∧ Paym.(Amount
∧ OwnRisk) ∧  Cond.(Acc ∧ Exc) ∧ Prem.Direct
∧ Payee.Person) }
```

The `unfold()` and `numAlternatives()` operations have been implemented in the tools of *Rumi*. Figure 3a shows a screenshot of the tool for defining domain models. Hereby the radio button *extensional* has been selected, which results in the execution of the operation *unfold* for the selected domain model. In tandem the total size of the product line is computed which is also shown in the figure (2976). The *unfold* operation also checks whether each possible alternative is valid with respect to the defined constraints and as such the total set of alternatives will be reduced when the constraints are also defined. In the tool every individual product can be selected and the description will be provided in the text field.



Model: InsuranceProduct

Concepts: InsuredObject, Coverage, Payment, Conditions, Premium, Payee

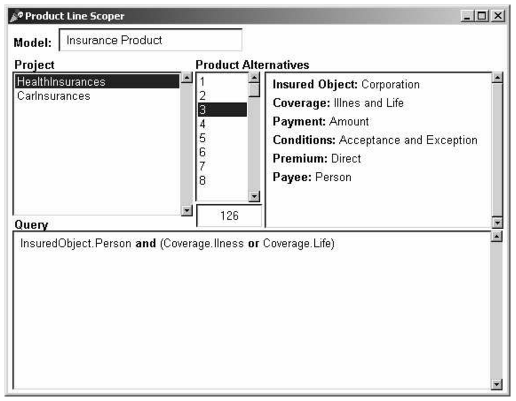
Product Set: 1 [✓ 2] 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Insured Object: Corporation
Coverage: Illnes and Life
Payment: Amount
Conditions: Acceptance and Exception
Premium: Direct
Payee: Person

intensional [✓ extensional] 2976

Informally, an insurance is an agreement between an insurance agent and a client who pays the agent a certain premium, for a compensating service in case some pre-defined situations (risks) happen to the object that is ensured.

a)



Model: Insurance Product

Project: [✓ HealthInsurances] CarInsurances

Product Alternatives: 1 2 [✓ 3] 4 5 6 7 8 — 126

Insured Object: Corporation
Coverage: Illnes and Life
Payment: Amount
Conditions: Acceptance and Exception
Premium: Direct
Payee: Person

Query: InsuredObject.Person **and** (Coverage.Ilness **or** Coverage.Life)

b)

Figure 3. a) Extensional representation of product alternatives and b) Product Line Scoper Tool

3.4 Reducing Design Spaces

In principle, it is possible to list all the alternatives and analyze and select them separately. However, for large design spaces, the number of alternatives may soon lead to a combinatorial explosion and likewise the identification and reasoning about individual alternatives may become very difficult. Moreover, not all the alternatives may be feasible or possible at all and it would be worthwhile to reduce the design space so that only the relevant alternatives are considered. To support this we introduce a query-based approach whereby the domain engineer specifies an expression that includes a condition for either selecting or eliminating part of the design space:

```
Select from Model where <condition>
```

Hereby *condition* can be made up of several (logical) functions: The query will result in a reduced design space that includes the set of alternatives that meets the specified condition. The following query reduces the space of insurance products to include only health insurance products:

```
HealthInsuranceProduct ::
Select from InsuranceProduct
Where <Insbj.Person and (Cov.Illness or
Cov.Life)>
```

The reduction of the design space, i.e. the scoping of the product line is implemented in the *Product Line Scope* tool, which is shown in Figure 3b. In this tool, for the same domain model different scoping *projects* can be defined. In the example a product line of health insurance has been scoped from the domain model `Insurance Product`.

3.5 Mapping domain alternatives to implementation

At this point it is decided on the set of products that needs to be produced and delivered. The product alternatives have been derived from the abstract domain model but the product portfolio consists of a very precise and concrete set of products. However, each individual product in the product line can be implemented in different ways dependent on the selected quality criteria and the computation models. This results in a different alternative space and scoping at this level becomes necessary. This product implementation scoping will be applied by the software engineer who will continue the scoping from the domain engineer, but now at the analysis and design level.

Implementing products can be considered as a mapping from one domain to an implementation domain. We can specify this in the following general form:

```
Model.weave(Property)
```

Here the operation `weave` maps the properties to the products of the model. A property can be considered as a tag to the elements of a corresponding model to denote a specific design decision. Similar to the bindings of the domain features to the dimensions of the model we can bind features of the implementation model to the dimensions. As such `Property` is a set that includes either a model of the computation model in which the product will be implemented or the quality model that will be evaluated. `Property` can specify issues such as hardware platform, implementation language or various quality factors such as adaptability and reusability. Assume, for example, that the product alternatives will be implemented using object-oriented abstractions. In the object-oriented model [2] concepts may be mapped to a class, operation or an attribute. In the same way as for modeling the domain we can define the property set `Object` as follows:

```
Object = (Cl ; Op ; At)
```

Hereby, `CL`, `Op` and `At` refer to class, operation and attribute respectively. The symbol ';' is used to denote alternative features. The following specification defines a new space `Object-HealthInsuranceProduct` that includes all the possible object-oriented implementations of the alternatives in `HealthInsuranceProduct`:

```
Object-HealthInsuranceProduct :=
HealthInsuranceProduct.weave(Object)
```

This set `Object-HealthInsuranceProduct` includes all the alternative object-oriented implementations of the `InsuranceProductScope`. This set includes 512000 implementation alternatives. The following represents an example of a specification of the product implementation:

```
(InsObj.Person.CL ∧ Cov.Illness.OP ∧
Paym.(Payment ∧ OwnRisk).OP ∧
 Cond.(Acc ∧ Exc).OP ∧ Prem.Direct.AT ∧
Payee.Person.AT) }
```

Hereby, `CL` is bound to `InsObject`, meaning that the latter will be mapped to a class. `Cov`, `Paym`, and `Cond` are bound with `OP`, meaning that they will be implemented as an operation. Finally, `Prem` and `Payee` are bound with `AT`, meaning that they will be represented as an attribute in the final implementation. This is only one alternative, and because the space of implementation alternatives is too large we might decide to reduce the space to define the product implementation scope. This may be supported by the utilization of heuristic rules. For example, for design spaces including the dimension `Object` we may utilize the heuristic rules from the object-oriented analysis and design methods [5] for deciding whether an entity has to be selected as a class, operation or as an attribute. Most methods define rules in an informal manner. Nevertheless, method rules can be expressed using conditional statements in the form IF <condition> THEN <consequent> [11]. The consequent part may be an identification or elimination action and as such heuristic rules may be applied both to support the selection and the elimination operations of the reduction of the design spaces. To select alternatives from `Object-InsuranceProduct`, for example, we may utilize the following heuristic rules:

```
IF an entity is relevant
THEN select the entity as a class (CL)

IF an entity describes a structural action or
behavior of an object
THEN select entity as an operation (OP)

IF an entity describes another entity
THEN select entity as an attribute (AT)
```

Note that these are only examples of heuristic rules and many more rules may be extracted from the corresponding methods [11]. The software engineer can apply these heuristics, provide a decision and describe these into queries. *Rumi* provides tools to model these heuristic rules and apply these for design space reduction. The result of these rules is defined as a constraint and is utilized to reduce the scope of the implementation alternatives. Assume, for example, that according to these rules it is decided to include only alternatives in which `InsuredObject`, `Paym` are mapped to a class, `Prem` to operation and the other features to attributes. This may be again specified in a query:

```
Object-HealthInsuranceProduct ::
Select from HealtInsuranceProduct
Where <InsObj.CL and Paym.CL and Prem.OP>
```

Using the operation `numAlternatives()` we can compute the set of alternatives from this set, which is 20. We may further reduce this space by applying other heuristic rules and stakeholder constraints.

4. Related Work

In [8] software product line scoping is categorized in *product line scoping*, *domain scoping* and *asset scoping*. Hereby asset scoping identifies the various elements that need to be made reusable to produce the product alternatives in the product line scope. In this paper we have provided an approach to integrate domain scoping and product line scoping. We did not explicitly consider asset scoping but since every asset can be considered as an alternative element we could describe the *asset alternative space* using design space models in the same way that we did for domain models. In addition we can use the same mechanism for defining the constraints and heuristics to reduce the set of assets. The different issue here is that the applied constraints and heuristics will be specific to the assets. In our future work we will aim to explicitly integrate this asset alternative scoping with the other two scoping processes. It should be noted that in addition to the three categories of scoping in [8] we have also introduced another different type of scoping, which is the product implementation scoping. To the best of our knowledge there have not been any attempts that explicitly deal with this.

Composition and customisation of design spaces with multiple dimensions has also been addressed in [4] whereby so-called *hyperspaces* span a concern space that includes various concerns. Hyperspaces are similar to the concept of design space that we have introduced. Hyperspaces contain different set of so-called *hypermodules* that integrates a set of *hyperslices,* which are selected concerns from the hyperspace. The hyperslices are integrated using so-called *composition rules.* Because the same hyperspace can be used to define hypermodules different systems can be composed. Hypermodules resemble the reduced set of the design space models, which result after applying the various design algebra operations.

5. Conclusion

Product line scoping is one of the key activities for ensuring the success of a product-line engineering approach. Currently, product line scoping is generally realized either by a product requirements analysis or a domain analysis process. Product requirements analysis may miss the products that can not explicitly be derived from the product requirements. Domain models on the other hand are inherently too abstract to identify the product alternatives and reason about these explicitly. We have introduced the concept of design space models (DSMs) as a complementary technique to existing product line scoping techniques. As an example we have explained the scoping process for insurance products that we have carried out within an industrial project.

We have distinguished between product line specification scoping (PLSS) and product line implementation scoping (PLIS). In the PLSS we have scoped the insurance products by formally representing the domain model using design algebra, specifying the constraints between the various features and reducing the product alternative space using *unfold* operation and selection queries. In the PLIS we have mapped the existing product line to the object-model that has been specified in design algebra. This resulted in a new alternative space that we have reduced using heuristics from the object-oriented model. The corresponding ideas have been illustrated using the tool environment Rumi that includes a set of tools for supporting the techniques of DSMs.

The techniques of DSMs are based on well-defined formalisms. This allowed us building tools within the development environment called Rumi. We have verified the approach by applying it for various industrial applications such as insurance products and transaction systems [11][12].

Our future work includes the explicit consideration of scoping from the economic point of view that we have deliberately not considered in this paper since we think that it requires careful study by its own. Once these cost models are developed we think that we can use design algebra, design space modeling, and the related tool Rumi to scope the product alternatives based on these cost models.

ACKNOWLEDGEMENTS

This research has been financed by the Dutch National Organization for Science (NWO) for the IMOORA project.

References

[1] P. Clements & L. Northrop. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[2] G. Booch, J. Rumbaugh & I. Jacobson. *The Unified Modeling Language User Guide*, Addision-Wesley, 1999.

[3] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak & A.S. Spencer Peterson. *Feature-oriented Domain Analysis (FODA) Feasibility Study.* Technical Report, CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, November 1990.

[4] H. Ossher & P. Tarr. *Multi-Dimensional Separation of Concerns using Hyperspaces.* IBM Research Report 21452, April, 1999.

[5] A.J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[6] Software Productivity Consortium. *Reuse-Driven Software Processes Guidebook, Version 02.00.03.* Technical Report SPC-92019-CMC, November 1993.

[7] Software Technology for Adaptable, Reliable Systems (STARS). *Organization Domain Modeling (ODM) Guidebook, Version 2.0.* Technical Report STARS-VC-A025/001/00, June 1996.

[8] K. Schmid. *Scoping Software Product Lines*, in: P. Donohoe (ed.), Software Product Lines: Experience and Research Directions, Kluwer Academic Publishers, pp. 513-532, 2001.

[9] M. Shaw & D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline,*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[10] B.Tekinerdoğan. *Synthesis-Based Software Architecture Design.* PhD Thesis, University of Twente, Dept. of Computer Science, The Netherlands, March, 2000.

[11] B. Tekinerdoğan & M. Akşit. *Providing automatic support for heuristic rules of methods.* In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 496-499, 1999.

[12] R. Willems. *Design of an Object-Oriented Framework for Insurance Products (in Dutch),* Msc. Thesis, Dept. of Computer Science, University of Twente, 1998.

Generic Variability Management and Its Application to Product Line Modelling

Klaus Schmid and Isabel John

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
+49 (0) 6301 707 - 158, +49 (0) 6301 707 - 250
{Klaus.Schmid, Isabel.John}@iese.fraunhofer.de

1 INTRODUCTION

Variability Management is a concern that arises in Product Line development throughout all lifecycle phases [6]. It can actually be seen as *the* key feature that distinguishes product line development from other approaches to software development.

While the basic concerns are similar throughout the different stages of a software lifecycle, the means for addressing them are typically different in the various stages: in the analysis phase mechanisms related to the specific analysis technique are used, typically text-based [21] or UML-related techniques are proposed [10, 13, 19, 4, 26] specific design-based approaches have been proposed [8, 5], and of course implementation mechanisms have been studied [16, 9, 20].

In this paper, we will focus on an approach for the systematic management of variability in the specification phase. In this product line modelling (or domain analysis) phase, a model of the requirements of the product line is developed which expresses the variability required from the product line. Many different notations are in practical use for requirements engineering [22]. While especially text-based and use-case/UML notations are used in the product line context [23], it is desirable that an approach supporting the specification of product lines is open regarding the notation [7]. This lead us to the idea for the approach presented in this paper. This approach aims to *support the modelling of variability for arbitrary specification techniques*.

Our approach can actually be extended into an approach which is sufficient as a basis for variability management across the various lifecycle phases. However, we will focus here on the specification phase and will provide case studies that substantiate our claim.

The key question of course is: why would one want to be independent of the specification technique? There are two fundamental reasons motivating such an approach:

- *The scientific reason:* such a generic approach could be evaluated in an arbitrary set of contexts, thus facilitating the growth of a scientific body of knowledge about it. As the mechanism is applicable in different contexts and in different domains it can be used in a variety of situations and can therefore be validated much easier than an approach that is applicable only with a single specification technique.
- *The pragmatic reason:* Fraunhofer IESE applies its technologies in many different companies, leading to the need for highly adaptable techniques. As we do technology transfer to companies with different organizational structures, in different sizes and in different domains the approaches we develop must be generic and adaptable to many different contexts.

Particularly the latter reason originally lead to the definition of the PuLSE-CDA[1] approach [7, 3] as part of the PuLSE™ method[2] [2]. This approach is a highly customizable domain analysis approach which can be augmented with the variability management mechanisms described here.

2 VARIABILITY MANAGEMENT IN THE SPECIFICATION PHASE

The specific approach to variability management we propose consists of the following components:

- A decision model as a basis for characterizing the effects of variability.
- A range of primitives for describing the relation between variation points and the specific decisions (or group of decisions) on which their resolution depends.
- A common (maximal) set of variation types.
- An accompanying mapping of the variability types on the specific specification techniques to express the variation points.

Only the last point, the mapping, has to be adapted to the specific representation technique. The other three parts as well as the semantic interrelation among the four are independent of the specific representation approach. We will now briefly discuss these four elements.

2.1 The Decision Model

The decision model was initially devised in the context of the Synthesis approach for variability management [11]. In the meantime, this technique has been widely applied both in research and industry [13, 14, 1, 12, 15, 17, 24].

The specific kind of decision model we propose is different from other approaches in two ways:

- It is more comprehensive in terms of the information it

1. CDA = Customizable Domain Analysis;
2. PuLSE is a registered trademark of Fraunhofer IESE

contains

- It does not explicitly relate to the variation points, but rather it defines a decision variable which is then only referenced at each specific variation point using the decision evaluation primitives.

Each of the decision variables that is defined in the decision model is in turn described by the following information:

- **Name:** The *name* of the defined decision variable; the name must be unique in the decision model
- **Relevancy:** The *relevancy* of a decision variable for an instantiation may depend on other decision variables., e.g. the decision variable describing the memory size is only valid if the decision variable describing the existence of memory is true. This can be made explicit by the relevancy information.
- **Description:** A textual *description* of the decision captured by the decision variable
- **Range:** The *range* of values that the decision variable can take on. This can be basically any of the typical data types used in programming languages. However, instead of a real or integer often only a range is important. Moreover, probably the most common type is the enumeration, as the relevant values are often domain dependent. Further, Boolean variables are quite common.
- **Cardinality:** As opposed to other approaches, we do not emphasize the difference between variables which can only assume a single value and variables that can assume sets of values during application engineering. Rather, we define a selection criterion, defining how many of the values of a decision variable can be assumed by it. This is represented by *m–n*, where *m* and *n* are integers and give the upper- and lower-bounds for the cardinality of the set representing the value of the decision variable in the context of a specific application. Thus, basically, all decision variables get a set of values during application engineering. However, we use *1* as a short-hand notation for *1–1* and in this case we also write the value of the decision variable as a single value (without curley brackets) and treat it for the purpose of decision evaluation like a non-set value.
- **Constraints:** Constraints are used to describe interrelations among different decision variables. This is used to describe value restrictions imposed by the value of one variable onto another variable. We use this approach also to describe the *requires relationship*, as this simply results in a special case in our framework. This constraint can of course also contain domain knowledge. Consider for example the following constraint: the value of the decision variable describing the memory size has to be > 16384 if the decision variable describing the existence of memory is true. This constraint at the same time represents the domain knowledge that in the product line the minimum memory size is 16KB.[1]
- **Binding times:** A list of possible binding times when the decision can be bound. This can be *sourcetime*, *compiletime*, *installation time,* etc. [FODA]. Additional binding times may exist, and can be product line specific. As opposed to the FODA work and many related approaches, we allow several binding times, meaning depending on the specific product the variability may be bound at any of these times. This technique was first introduced in ODM [18] as “binding sites”.

1. Of course, this would usually be represented with adequate constants (e.g., 16384 := Min_Mem_Size).

Depending on the specific context of our industrial projects, we sometimes used slight variations of this approach to decision modeling. However, regarding the information content, it was always a subset of this information (if only product line modelling had to be supported) [17].

Using this description of a decision variable, we can define a decision model simply as a set of decision variable definitions. For practical reasons we usually represent them as a table.

2.2 Decision Evaluation Primitives

The way we defined the decision model, it is completely independent of the variation points in a variability model. In order to relate a decision to a variation point we must explicitly describe it. This is done using the decision evaluation primitives together with the variation point representation as discussed below.

The reason why we do not directly relate the impact of a decision variable to the variation points is that the same decision may easily have many different forms of impact on the variation points. This allows us to decouple the decision itself from its impact on the product line model.

Our approach to decision evaluation is very similar to expression evaluation in existing program languages, the main extension being that we may need to deal with set values.

The following list provides some examples of relations we use for decision evaluation:

sub	real subset $\subset$
subeq	subset or equal $\subseteq$
#	cardinality of a set
in	is element of a set
=>	logical implication
<=>	mutual implication (iff)

Using these primitives, logical expressions can be built that can be used to denote in which way a specific variability must be resolved. It is also possible to build value expressions. We will discuss this further in the following section.

2.3 Supported Variability Types

Many different variability types have been mentioned in literature: optionalities, alternatives, set-optionalities (a set of options may be selected), etc.

From our practical experience we deem the following variabilities to be the most relevant:

- **optionality:** a property either exists in a product or not
- **alternative:** two possible resolutions for the variability exist and for a specific product only one of them can be chosen
- **multiple selection:** several variabilities may be

selected for inclusion in a product

- **single selection:** only a single variability out of a group of variabilities may be selected for inclusion in a product
- **value reference:** the value of the decision variable can be directly included in the product line model. (This, of course, only makes sense with decision variables that only assume a single variable in application engineering.)

The optionality and variability refer by nature to a logical expression as constructed using the decision evaluation primitives. Further the multiple and single selection refer to a value expression, as this is used to differentiate among the different possibilities. The value reference, finally, takes an arbitrary decision variable with a single value.

2.4 Representation-Specific Mapping of The Variation Points

As we discussed above, the decision model is basically representation independent. However, we need to represent the variation points in the domain model, which employs a specific specification technique. Therefore we need to map the different types of variabilities to the target specification technique.

As we will see in the next section the specific notation for the variation point may be graphical, textual based, or on any other basis. In order to simplify the adaptation process, we did so far always use the standard description approach for referring to values of decision variables described in Section 2.2.

The different variability types should be mapped in a homogenous manner to the specification language. For each variability type a unique mapping has to be found. This mapping has to take a form so that confusion with other legal expressions in the target specification language can be minimized.

Only this mapping from elements of the decision model to the specification formalism has to be adapted when the approach is applied with a new specification formalism. If the specification formalism uses graphical models, the mapping can be done using extra graphical elements with the decision variables as attributes of these elements. If the UML or a similar modelling approach is used as specification formalism, the model elements can be extended (e.g. with stereotypes, cf. [10]). If the specification formalism is text, markers for the different kinds of variabilities can be introduced into the textual description.

2.5 Discussion of the Approach

The approach outlined above is sufficient to describe all common forms of variabilities and dependencies among them. For example alternatives that are mutually exclusive can be represented using an alternative or a single selection, which refer to a decision variable (in the case of the single selection, this is only sufficient if the decision variable can take on only a single value).

The requires dependency can also be modelled, and it is actually modelled on the level we believe to be the most adequate: it is made explicit on the level of the decision model in the form of constraints on the possible values of the variable.

3 EXPERIENCES USING THE APPROACH

The approach to variability management in product line modeling described above has already been applied in several cases, most notably two industrial applications, where one used a graphics-based approach, while the other uses as a text-based approach. We will now briefly discuss the implementation of our approach in these two vastly different contexts, as this nicely illustrates the different forms of mappings that are made.

3.1 Experiences with a Graphical Representation

We applied our approach to the variability management in the context of product line modelling in an environment, where a graphical notation was required. This notation was the basis for a business process notation (ARIS), which was in turn the basis for requirements definition for systems of the customer [17].

The ARIS notation, which provides the basis for this application of the approach focusses on business processes. The basic notational elements are shown in Figure 1. In this specific case additional elements had to be defined in order to represent also system internal information and control flows [17]. Our approach focussed on augmenting this notation with additional variability elements that could be used both in business process as well as control flow modelling.

In order to describe selections the ARIS modeling notation uses two notational elements: the connector together with the event. The *connector* defines the form of selection, the *event* defines the different cases that can occur and under what circumstances each of these paths is taken. Figure 2 shows such an example business process with a selection. As defined by our approach the first three parts of our approach could be taken verbatim. We only made some minor pragmatic modifications:

- The decision model had the same entries as defined above, with the exception of the binding times. There was no need to capture the binding time as this was always implicitly the modeling phase. Further there was one additional entry: the actual values for the various systems could also be defined as part of the decision description. This had pragmatic reasons, as in this case the number of decision variables was limited and especially the number of systems was small. The decision model was then simply written as a web page, as much documentation in this environment was kept in an intranet-based manner.
- The decision evaluation primitives were used as described in Section 2.2.
- Regarding the different forms of variability, we decided to not support the value-reference, as we did not find a case where we would need this approach. Also the alternative is always described as a single selection.

Based on these decisions we defined the mapping of the

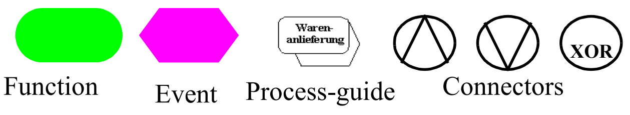


Figure 1. Basic notation for business processes (eEPK)

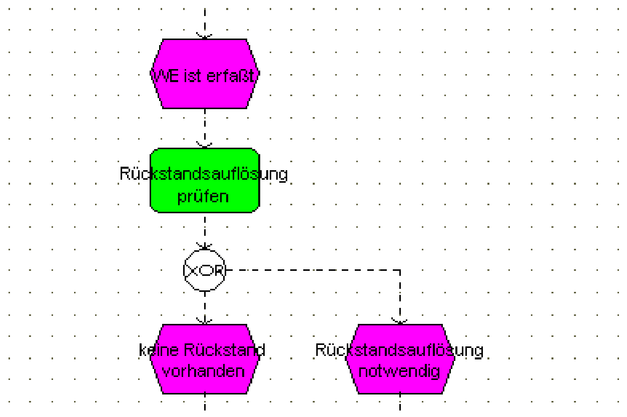

WE ist erfaßt
Rückstandsauflösung prüfen
XOR
keine Rückstand vorhanden
Rückstandsauflösung notwendig

Figure 2. An example business process

basic variability types onto the representation mechanisms. In order to enable the users of the approach to clearly differentiate between the basic notation and any variability information, we defined completely different notational elements, which, however, fit into the overall approach. Figure 3 shows the different notational elements we introduced. We also adopted the differentiation between decision symbol (connector) and selectors for a specific flow (event), which is typical of ARIS.

When mapping the various variant discriminators it is key to keep in mind that we are using here a notation that imposes certain restrictions, for example, by removing some variation the overall flow may not fall apart. Thus, we can only remove certain (alternative) paths from the control flow. Based on these restrictions, we mapped the variability types optionality, multiple selection and single selection, we selected for representation, in the following manner:

Optionality: This implies that a certain path may either be part of a system variant (an instantiation), or not. Thus, we need to attach two forms of information to it: the situation in which this path is part of the final model and if it is part of the final model, the (runtime) situation in which it is actually taken. The second part obviously corresponds to the event mechanism in the ARIS business modeling approach, while the former is the optionality-specific addition. We thus added the optional variant decision to the modelling language. As shown in Figure 3 it consists of a runtime decision and a domain decision part. The domain decision part in turn uses the decision evaluation primitives as described in Section 2.2 in order to describe whether the branch started with this decision should be part of the instantiated model. The runtime decision part in turn is annotated using the ARIS-notation in order to describe what will happen in case this branch is selected.

Single Selection: The single selection is mapped to the unique variant decision (cf. Figure 3), which works similar to a runtime decision in ARIS, with replacing connectors by the unique variant decision, and the events to the variant discriminator (cf. Figure 4). In this case we restricted the expression for selecting among the various paths to a decision variable, with the variant discriminators showing the different values. Note, that upon resolution of the variability none of the notational symbols for variability will remain in the instantiated flow.

Multiple Selection: The multiple selection has been mapped in very much the same way as the single selection. The main reason for having both of them was clarity of the instantiation semantics. In a work flow (or control flow) representation like ARIS, a runtime decision must remain upon resolution of the specification variability in the case of a multiple selection.[1] This is different from the single selection where all variability symbols are removed upon instantiation. Here, they are transformed into run-time variability (if more than one option is chosen).

This approach to modeling was used for modeling several systems in the domain of merchandising information systems and about ten e-commerce shops. We found this approach to be easily applicable to these systems. Especially in the e-commerce context it was also well accepted by the development personnel.

3.2 Experiences with a Text Based Representation

Our variability management approach has been applied in practice also with text-based requirements in an embedded systems company. A textual representation was chosen because the stakeholders in the domain were very familiar with textual representations and not with other forms of

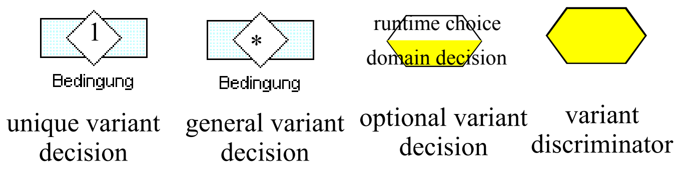


Figure 3. Symbols added for variants

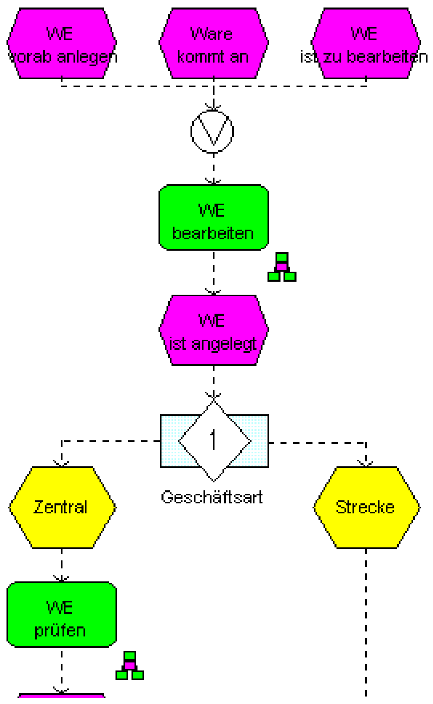


Figure 4. Example for the description of variants

1. Note, that in the context of this case study [17], we differentiated only between specification time and runtime.

requiremetns documents. They had also invested considerable effort into the improvement of their approach to textual requirements documentation.

In order to be able to model and manage variability, the existing mechanisms for writing textual requirements had to be extended into a product line modelling approach. According to our approach, only the mapping of the variability types onto the target representation formalism had to be adapted. However, to be complete, we will now briefly describe the specific realization of all four components of our approach.

- The decision model as described in section 2.1 was introduced. This was realized using an Excel-table. A sanitized version of such a table excerpt is shown in Figure 5.
- We used the decision evaluation primitives shown in Section 2.3.
- We did decide to not support the single selection, as it is a special case of the multiple selection. Moreover, so far most instances we found during our work in this domain were instances of the multiple selection anyway.

This shows that, as expected, we could transfer our concepts in a straight-forward manner to this domain. This leads to the most interesting part of the case studies: how was the mapping of the variation point types performed.

For this mapping the variability types onto the textual specification we decided to use textual constructs framed with "<<" ">>", as these are text fragments which did so far never occur in this domain.

Thus, we wrote optional variability in the following way:

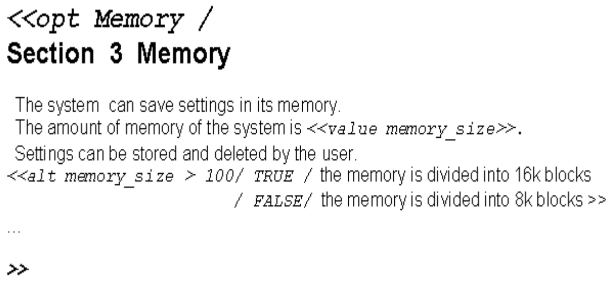
<<*opt expr1* / text >>.

Similarly, for alternative variability, we used the term:

```
<<alt expr2  / value-1 /  text1
             / value-2 /  text2 >>.
```

Here expr1 and expr2 are logical expression as discussed above. These expressions could be constructed using the primitives described in Section 2.2.

For multiple alternative variability we restricted the expression to a decision variable instead of a value expression and introduced the keyword mult:



<<opt Memory /
Section 3 Memory

The system can save settings in its memory.
The amount of memory of the system is *<<value memory_size>>.*
Settings can be stored and deleted by the user.

```
<<alt memory_size > 100/ TRUE / the memory is divided into 16k blocks
                       / FALSE/ the memory is divided into 8k blocks >>
...

>>
```

Figure 6. An example using the textual notation

```
<<mult  decision-variable / value-1 / text1
                          / value-2 / text2
                          .....>>
```

Finally, for values the term <<*value decision-variable*>> was used.

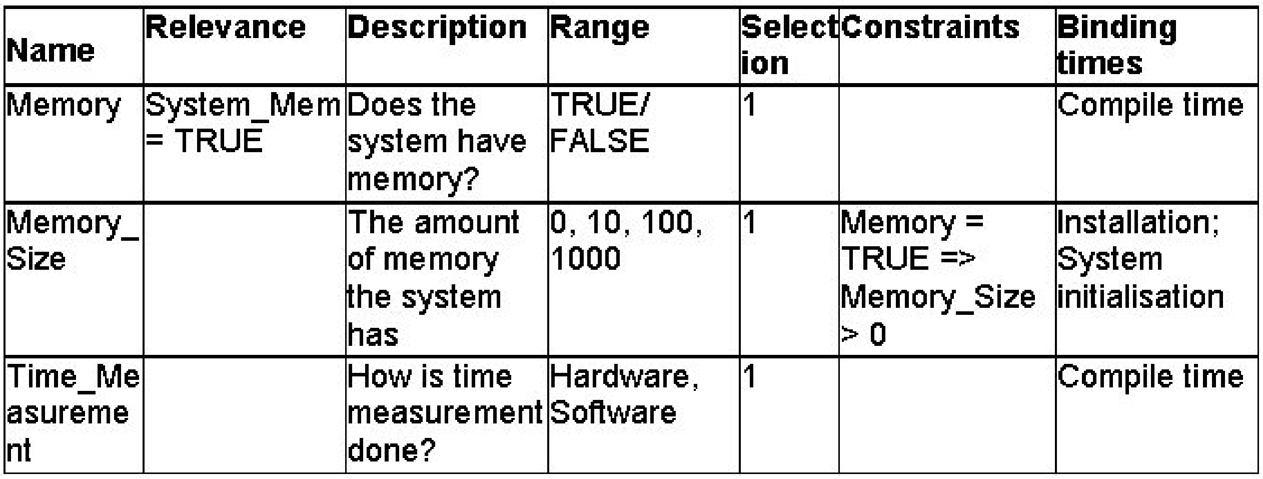
Using this approach we described the product line model. Figure 6 shows a sanitized excerpt of such a product line model document which includes optional, alternative, and value variability.

In this company, we identified so far during modeling about 50 decision variables and about 100 variation points had to be introduced into the documentation. We expect that once the product line model is complete, it will contain more than 100 decision variables and several hundred variation points. The resulting domain models went through inspection by the company and were well accepted by the development team. In particular the notation was considered to be well readable and the resulting models to be well understandable.

4 CONCLUSION

In this paper we described an approach to variability modelling for product line models. The development of this approach was driven from the need for an approach that can be easily applied in a wide range of practical contexts and in combination with many different specification techniques. Based on our experiences in applying this approach, we found that

Our approach to variability management is sufficiently expressive to support modeling variability



Name	Relevance	Description	Range	Selection	Constraints	Binding times
Memory	System_Mem = TRUE	Does the system have memory?	TRUE/ FALSE	1		Compile time
Memory_Size		The amount of memory the system has	0, 10, 100, 1000	1	Memory = TRUE => Memory_Size > 0	Installation; System initialisation
Time_Measurement		How is time measurement done?	Hardware, Software	1		Compile time

Figure 5. Example for the description of variants

for arbitrary specification techniques.

Moreover, we could already apply this approach as part of the PuLSE approach in different industrial contexts, demonstrating that it provides sufficient expressiveness for these situations.

Based on these encouraging results our next steps will be to extend this approach to cover the whole life-cycle and to improve the formal basis upon which it rests.

5 REFERENCES

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.

[2] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, Los Angeles, CA, USA, May 1999. ACM.

[3] J. Bayer, D. Muthig, and T. Widen. Customizable Domain Analysis. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, Erfurt, Germany, Sept. 1999.

[4] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use Case Description of Requirements for Product Lines. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02)*, Sept. 2002.

[5] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[6] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability Issues in Software Product Lines. In E. S. Institute, editor, *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, Oct. 2001.

[7] J.-M. DeBaud and K. Schmid. A Practical Comparison of Major Domain Analysis Approaches - Towards a Customizable Domain Analysis Framework. In *Proceedings of the Tenth Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, June 1998.

[8] O. Flege. System family architecture description using the uml. Technical Report IESE Report No. 092.00/E, Fraunhofer IESE, 2000.

[9] C. Fritsch, A. Lehn, and T. Strohm. Evaluating Variability Implementation Mechanisms. In *Proceedings of the Second International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing (PLEES'02)*, Nov. 2002.

[10] I. John and D. Muthig. Tailoring Use Cases for Product Line Modeling. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02)*, Sept. 2002.

[11] M. Kasunic. Synthesis: A Reuse-Based Software Development Methodology, Process Guide, Version 1.0. Technical report, Software Productivity Consortium Services Corporation, Oct. 1992.

[12] C. Krueger. Variation Management for Software Product Lines. In G. Chastek, editor, *Proceedings of the Second Software Product Line Conference*, LNCS 2379, San Diego, CA, Aug. 2002. Springer.

[13] M. Mannion, B. Keepence, H. Kaindl, and J. Wheadon. Reusing Single System Requirements for Application Family Requirements. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[14] A. Mili and S. M. Yacoub. A Comparative Analysis of Domain Engineering Methods: A Controlled Case Study. In P. Knauber and G. Succi, editors, *Proceedings of the International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Limerick, Ireland, June 2000.

[15] D. Muthig. *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD Theses in Experimental Software Engineering; Fraunhofer IRB Verlag, 2002.

[16] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proceedings of the Net.ObjectDays (NODE'02)*, Erfurt, Germany, Oct. 2002.

[17] K. Schmid, U. Becker-Kornstaedt, P. Knauber, and F. Bernauer. Introducing a software modeling concept in a medium-sized company. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000.

[18] Software Technology for Adaptable, Reliable Systems (STARS). *Organization Domain Modeling (ODM) Guidebook, Version 2.0*, June 1996.

[19] T. van der Maßen and H. Lichter. Modeling Variability by UML Use Case Diagrams. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02)*, Sept. 2002.

[20] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[21] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[22] A. Davis. *Software Requirements: Objects, Functions, and States*. Prentice Hall PTR, 1993

[23] Birgit Geppert and Klaus Schmid (Eds.). *Proceedings of the International Workshop on Requirements Engineering for Product Lines*, Sep. 2002.

[24] M. Coriat, J. Jourdan, and F. Boisbourdin. The SPLIT Method. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pp. 147-166, Kluwer Academic Publishers, 2000.

[25] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[26] M. Morisio, G. Travassos, and M. Stark. *Extending UML to Support Domain Analysis*. 2000.

Towards a General Model of Variability in Product Families

Martin Becker
System Software Group, University of Kaiserslautern
Kaiserslautern, Germany
mbecker@informatik.uni-kl.de

Abstract

The increasing amount of variability in software systems meanwhile leads to a situation where the complexity of variability management becomes a primary concern during software development. Whereas sound methodic support to analyze and specify variability on an abstract level is already available, the corresponding support on realization level is still lacking. The goal of this paper is to pave the way towards more systematic and consequently more efficient approaches to manage variability. To this end, it discusses the different motivations for variability in product families and the interrelationships between the specification and realization of variability. The paper further identifies appropriate concepts and interrelates them in form of a general model of variability in product families. In addition to this meta-model, the paper outlines an instantiation of the model: our language to specify variability in product family assets.

1. Introduction

During the past few years a noticeable shift towards an increased amount of variability[1] in software systems went through the software industry. The reasons for the increase of variability are twofold. First, variability has been recognized as the key to systematic and successful reuse. Especially in family-based approaches as software product lines or software product families, variability is a means to handle the inevitable differences among the systems in the family while exploiting the commonalities. In this case, variability enhances the reusability of software. Second, by providing more variability in software systems the flexibility and maintainability of those systems can be improved, as features can be added or adapted – even at runtime – without releasing new products. This can considerably increase the usability of the products.

Meanwhile the increase of variability leads to a situation where the complexity of managing the variability becomes a primary concern during software development that needs to be addressed explicitly by the software development methods and tools. Whereas sound methodic support to analyze and specify variability on the abstract level – e.g. the feature level – is already available, the corresponding support on realization level is still lacking [10]. This holds for the method as well as the tool support.

The realization and management of variability is for some reasons a non-trivial task. A first fact that hampers the consistent management of variabilities is that they often cannot be localized well but have *widespread impacts* down in the implementation documents. This is especially true, if the variability represents a varying quality of the system, as its overall performance, resource demands or interoperability, for instance. As with invariable solutions, a variability has to be addressed on the different levels of abstraction, e.g. architecture, components, subcomponents, classes, etc. to cope with complexity. In addition to this vertical impact, a variability often shows a horizontal impact, i.e. the variability affects several locations spread over the work products on the same level of abstraction. If the interface of a component is affected by a variability, for instance, then the calling components will be affected by the variability in some way too. However, a widespread impact of a variability results in interdependencies among the solution fragments[2] that have to be considered and managed. Furthermore, variabilities may interfere with each other, i.e. the variants[3] offered by the variabilities may exclude or require each other, resulting in further interdependencies. No matter how, the interdependencies caused by variabilities strongly aggravate the consistent and efficient management of the variabilities, as they raise the complexity of the overall solution and have to be considered throughout the whole lifecycle of the variabilities.

Another fact that complicates the management of variability is that variability appears in *manifold forms and realizations*. Generally, a variability extends the problem and consequently the solution space covered by the comprising system. A system that provides variabilities is planned to be applicable in a broader range of problems than its invariable counterparts. Those extensions are

[1] the capability to be changed or adapted

[2] the so-called variation points

[3] potential incarnation of the variability

neither restricted to certain problems nor to special solutions. In principle, every solution in a software system can be kept variable. A whole string of techniques and mechanisms to realize variability [13][11][17] in the various solution documents are already available, especially to handle variability on the code level but also on the upper levels of abstraction, the architecture for instance. Unfortunately, the impacts of the different realizations are not completely understood yet and there is consequently only little methodic support in the realization and management of variability.

This paper concentrates on the more product-family-related issues of variability management. The experiences we have made with variability management in various domains (building automation, embedded operating system, automotive), give us reason to believe, that the management of variability can be facilitated substantially, if we find a general model of how variability is realized and handled in product families that holds for all kind of variability throughout all abstraction levels. Such a model should:

- provide well-defined concepts to foster a common understanding of variability and its impacts
- identify common issues in the handling of variability, e.g. traceability, variable binding times and evolution
- and thus ease the development of variability aware software development methods and tools

Unfortunately, such a model is still missing, although the required terminology has already been defined quite well [19]. As a consequence, different approaches and slightly differing notions are used to realize and handle variability on the diverse abstraction levels, e.g. architecture, source code, and documentation, which inhibits synergistic effects to appear and complicates the consistent management of variability considerably.

In order to approach such a model, this paper discusses the interrelationships between the specification and realization of variability, identifies appropriate concepts and interrelates them in form of a general model of variability in product families. In addition to this model, the paper outlines an application of the model: our language to specify variability in product family assets.

The remainder of the paper is structured as follows: Section 2 discusses variability in product families. Besides the different motivations for variability, the two levels on which variability is approached are described. Section 3 illustrates the various incarnations of variability in the product family assets and identifies common properties among them. These commonalities in the realization of variability led to our model of variability in product families that is presented in section 4. Section 5 outlines an instantiation of the model: the Variability Specification Language. The paper closes with a conclusion.

2. Variability in Product Families

Product family[4] engineering [14] is a commonly accepted approach to exploit the reuse potential of similar software systems in a systematic and pre-planned way. The rationale behind this approach is to identify common solutions parts in a set of envisioned systems, which only have to be implemented once as so-called assets[5] and can be reused afterwards during the construction of the manifold family members in application engineering processes. This leads to the characteristic development process (six-pack) with the two development tracks: domain engineering (development for reuse) and application engineering (development with reuse).

Commonly, a product family comprises a reference architecture and a string of components. In addition to design and implementation documents, other kinds of assets as requirement specifications, test processes and data, production plans or domain knowledge can be supplied through the family as well depending on their reuse potential. The overall success of a product family approach, however, is closely coupled with the capability to handle the required differences among the family members in a consistent but also economic way. To this end, the family and its members are designed to be variable, i.e. they provide variabilities.

Generally speaking, a variability represents a capability to change or adapt system [19], i.e. the system facilitates certain kinds of modifications. Such a change or adaptation can affect the behavior of the system as well as its qualities. From a more technical perspective of a software engineer, a variability is a means to delay a (design) decision to a later phase in the lifecycle of the software system [19]. If a decision among a set of possible variants cannot be taken at a certain time during the development of the system, then a generic solution has to be realized in the work products at hand that allows to take the decision later on.

An analysis of the driving forces behind variability in software systems in general and product families in special reveals that two *main motivations* can be distinguished:

- **Usability**. By providing variability in a software system, the flexibility and maintainability of the system can be improved, as features can be added or adapted – even at runtime – without releasing new products. This can increase the usability of the products considerably.

[4] group of systems built from a common set of assets[4] [4]

[5] partial solution, such as a component, a design document or knowledge that engineers use to build or modify software products [21]

- **Reusability**. Variability has been recognized as the key to systematic and successful reuse. Especially in family-based approaches like software product families, variability is a means to handle the inevitable differences among systems in the family while exploiting the commonalities and thus increases the reusability of software.

The distinction between both motivations is necessary – although often neglected –, because the respective variabilities are handled differently and influence the software development processes in different ways. In case of increased usability, which can be generally of interest in any software development approach, the respective variability is used to handle an intra-product variation [11] and thus is a feature of the product, i.e. the product contains a mechanism to handle the variability dynamically after the delivery of the product to the customer. Apparently, such dynamic variabilities in principle require no special treatment during the development of the software systems as the can be realized and handled like any other feature of the system. The main issues raised by dynamic variabilities are the mastering of the increased functional complexity and the available implementation mechanisms. The increased reusability, on the other hand, can be considered as a peculiarity of family-based approaches. In this case, variability is used to handle the differences between the members of a family (inter-application variability). Obviously, such a variability is not a feature of the family members but of the comprising family and is handled statically, i.e. once bound to a distinct variant during the derivation of a family member, the variability vanishes and is no longer existing in the family member. Static variabilities affect the development processes considerably and raise a string of new issues, e.g. configuration and instantiation support, management of variants, evolution support etc.

It has to be pointed out, that the above-mentioned motivations do not exclude each other, but can coincide in one variability. In this case, the respective variability will support several binding times[6], and the handling of the variability will therefore depend on the actual binding time of the variability in the application engineering processes. If the corresponding decision is taken early enough in the software development process, then the variability is handled statically, i.e. the work products will be tailored according to the decision, otherwise it will be handled dynamically. A variable binding time allows to handle the trade-off between tailored, highly efficient solutions on the one-hand and flexible but more complex ones on the other. To subsume, from a product family perspective we have to face two motivations of variability: increased usability and reusability, whereas the latter considerably affects the development methods and tools and leads to peculiar issues. The increased usability is primary of interest if it coincides with attempts to increase the reusability of the work products. Consequently, the remainder focus of this paper focuses on static variabilities.

[6] phase in the development process in which the variability is bound to a certain variant

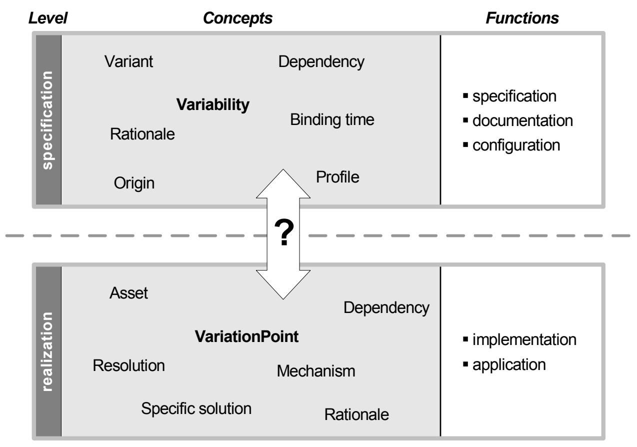


Figure 1. Two levels of variability handling

In family-based engineering approaches, variability is typically approached on two different levels of abstraction (cf. fig. 1): on the specification and the realization level. A distinction between those both levels is sensible, since they fulfill different functions and use different concepts to represent variability.

On the *specification level*, the involved stakeholders put their focus on the externally visible characteristics of variability and suppress realization details. The requirements and knowledge about the variabilities in the family are captured and represented by means of feature models [15] or dedicated variability models [7][20]. These models comprise information about the variabilities themselves, e.g. their origins, the range of offered variants, the reuse potential of the variants and furthermore information about the interdependencies among the variabilities, and information concerning the binding of the variability, e.g. the supported binding times and the roles that can bind a variability. In most cases, concepts of the problem space are used to express information about variability. The main modeling concepts used to represent variabilities are variable features (in the feature models) or variabilities themselves. Besides the information about the supported variabilities, there will also be information about the family members that are instantiated in the product family. This information is captured in application models or profiles that keep track of the variability-related decisions, which were taken during the configuration of the family members and control the resolution of the static variabilities in the application engineering. The information about variability on the specification level is used for various purposes. First, it is a means to analyze and specify the requirements for the implementations. Second, it documents the capabilities offered by the family on an abstract level, and thus is the entry point to understand the family and its members. Third, it forms the basis for the configuration and instantiation of family members [12].

On the *implementation level*, i.e. in the set of reusable assets provided through the product family[7], the software engineers have to realize and handle the required variability that has been specified on the specification level. To this end, they identify the impact of the variabilities in the various software assets offered through the product family and support the demanded variation by using appropriate mechanisms. In the application engineering processes, the application engineers deploy the static variabilities to derive specific solutions. During this derivation, the static variabilities are resolved to specific solutions. The main concept that represents variability on the implementation level is the variation point. A *variation point* is a spot in a software asset where variation will occur [13][19], i.e. where a variability is realized, at least partially. Thus, a variation point can be considered as some kind of generic element in a software asset. This is especially true, if the variability is motivated by reuse concerns.

Whereas sound methodic support to analyze and specify variability on the specification level is already available, the situation on the implementation level is quite different. Although a whole string of variability mechanisms exits to realize variability in the variation points (at least in the source code assets), e.g. appropriate language constructs, pre-processors, external generators etc., only few methodological and tool support is available that meets the rising demands of variability management. Thus, the mapping between the two levels (illustrated through the question mark in fig. 1.) and the management of variability on the realization level often remains a highly creative, individual and consequently complicated task. In order to cope with the rising complexity induced through variability, more systematic approaches are required. To this end, a general model of variability in product families is required, which identifies concepts, issues and patterns that can be applied throughout the whole lifecycle of a product family. Before we present our model, we first take a closer look at the implementation level of variability to reveal commonalities in a way variability is realized in the various asset types.

3. Variability on the Implementation Level

Within a product family any kind of work product used to construct a software system can be provided as a reusable software asset. Generally, some of them are not affected by variability – i.e. they are used as is in every member of the family –, but they usually form the minor part. Most of

[7] the implementation level of variability (all assets affected by variability on the different levels of abstraction) should not be confused with the implementation level of the product family (only code assets).

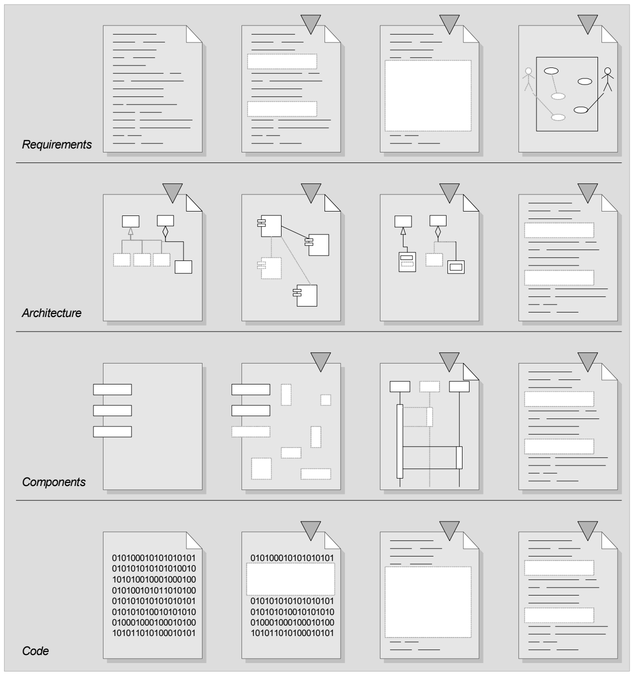


Figure 2. Various asset types in a product family

the assets are influenced by variability in one or the other way (illustrated through the grey triangle in fig. 2). Since the impact of a variability is neither limited to certain abstraction levels nor to distinct asset types, any asset provided through a product family can in principle contain variation points. Examples for such software assets are generic requirement templates, reference architectures, components, source code, test cases and even generic documentation assets (cf. fig. 2).

Apparently, there are different ways to represent the information contained in the assets. The information can be expressed through text, diagrams and binary data and each of these representations can contain variation points (cf. fig. 2). In recent years, especially variation points in diagrams attracted the attention of industry [18][16] and academia [9][2], as variability had to be implemented on the architectural level too, in order to allow for reuse in the large. Regarding the granularity of a variation point it can be stated, that a variation point can extend from multiple files, e.g. in case of software components, over document fragments like blocks, lines or diagram elements down to single information items, as characters or bytes. To summarize, variation points can appear in manifold ways in software assets, which complicates the management of the variabilities considerably, especially if they show widespread impacts.

Although the various incarnations of variation points differ substantially (cf. fig. 2), they also share some common properties. If we abstract from the different asset contents and the concrete realizations of variation points we observe the following *common functions* of variation points:

- **Localisation**. A variation point localizes a variation in an asset.
- **Abstraction**. From an external point of view, i.e. by suppressing internal realization details, a variation point abstracts from the specific realizations of the variants.
- **Specialization**. In addition to the abstraction, a variation point supports its specialization to a concrete solution in an appropriate way. To achieve this, it provides a specification that describes how to specialize the variation point to a distinct variant and a mechanism that realizes the specialization. In order enable variation, the specification of the specialization must be parameterized by the variabilities in some way, i.e. the specification must be a function of the variabilities.

Besides the aforementioned common functions, also *desirable features* can be identified that any variation point should have in order to render its functions and retain manageable (cf. [1]):

- **Identification**. It should be evident what part of the asset is immutable and what part is affected by variabilities. That way, the added complexity has only a limited impact in the asset.
- **Clear Structure**. Variation points in the assets should be structured as clearly as possible. First, they should not obscure the structure of the comprising asset. Second, if necessary, variation points should be structured in a hierarchical way, i.e. they should not overlap partially.
- **Expressiveness**. Along with the variation point its specialization must be specifiable. This is of special interest in the case of variation points that implement static variability, where the specialization is often carried out manually.
- **Localized**. The impact of a variability should be as localized as possible, i.e. the variation points should be designed and implemented in a way that concentrates the impact of the variability to as few points as possible.
- **Tracability**. Bidirectional traces between variabilities and the variation points that implement them must be maintainable in order to interrelate the two abstraction levels. Additionally, traces between the variation points that implement the same variability must be maintainable as well, in order to allow the consistent evolution of a variability.

In spite of the considerable differences between the various realizations of variability, e.g. in the way a variation point localizes variability and the way it supports its specialization in detail, apparently the commonalities among the variation points are substantial. The realization of this led to our model of variability, which is presented in the next section.

4. A Model of Variability in Product Families

In order to pave the way towards more systematic and consequently more efficient approaches to manage variability, we have developed a general (meta-)model of variability in product families that identifies and interrelates the concepts on the two abstraction levels mentioned in section 2. The motivation behind this model was:

- to provide concepts to foster a common understanding of variability and its impacts,
- to identify common issues and patterns in the handling of variability, and finally
- to ease the development of variability aware methods and tools

In fig. 3. you find an excerpt[8] of our model, which will be explained in the following.

The upper box at the right side addresses variability on the *specification level*. The main concepts are Variability and Profile. A *Variability* represents a variability in the ProductFamily and provides a Rationale and a Range of Variants. Between the Variants Dependencies, e.g. requires or excludes relationships, can be stated. As the Variants are associated with Variabilities, the Dependencies consequently concern the respective Variabilities. Furthermore, a Variability provides information about its supported BindingTimes.

A *Profile* keeps track of the variability-related decisions that were taken during the configuration of a family member. Thus, it specifies or identifies a member of the family. A Profile comprises a set of Assignments that can be accessed via the Variability. Each assignment represents a taken decision, e.g. Variant A has been chosen for Variability B at the BindingTime C. If no Assignment is available for a Variability, then the Variability is unbound in the profile.

The lower box at the right side addresses variability on the *realization level*. The main concept is the *VariationPoint*. The Assets provided through the ProductFamily can contain VariationPoints. A VariationPoint implements a Variability of the specification level, at least partially. Usually, a Variability causes several VariationPoints that are spread over multiple Assets. The concrete number of VariationPoints caused by a Variability depends of course on the Variability itself and the Assets provided through the ProductFamily. On the other side, a VariationPoint can be affected by more than one Variability. In this case, the impacts of the Variabilities overlap. Consequently, the multiplicity of the relationship between Variabilities and VariationPoints is n:m.

Local dependencies, i.e. Dependencies between the VariationPoints that are not already expressed through the Dependencies on the specification level, can be stated on the realization level. However, in order to keep the num-

[8] the complete model will be presented in our PhD thesis

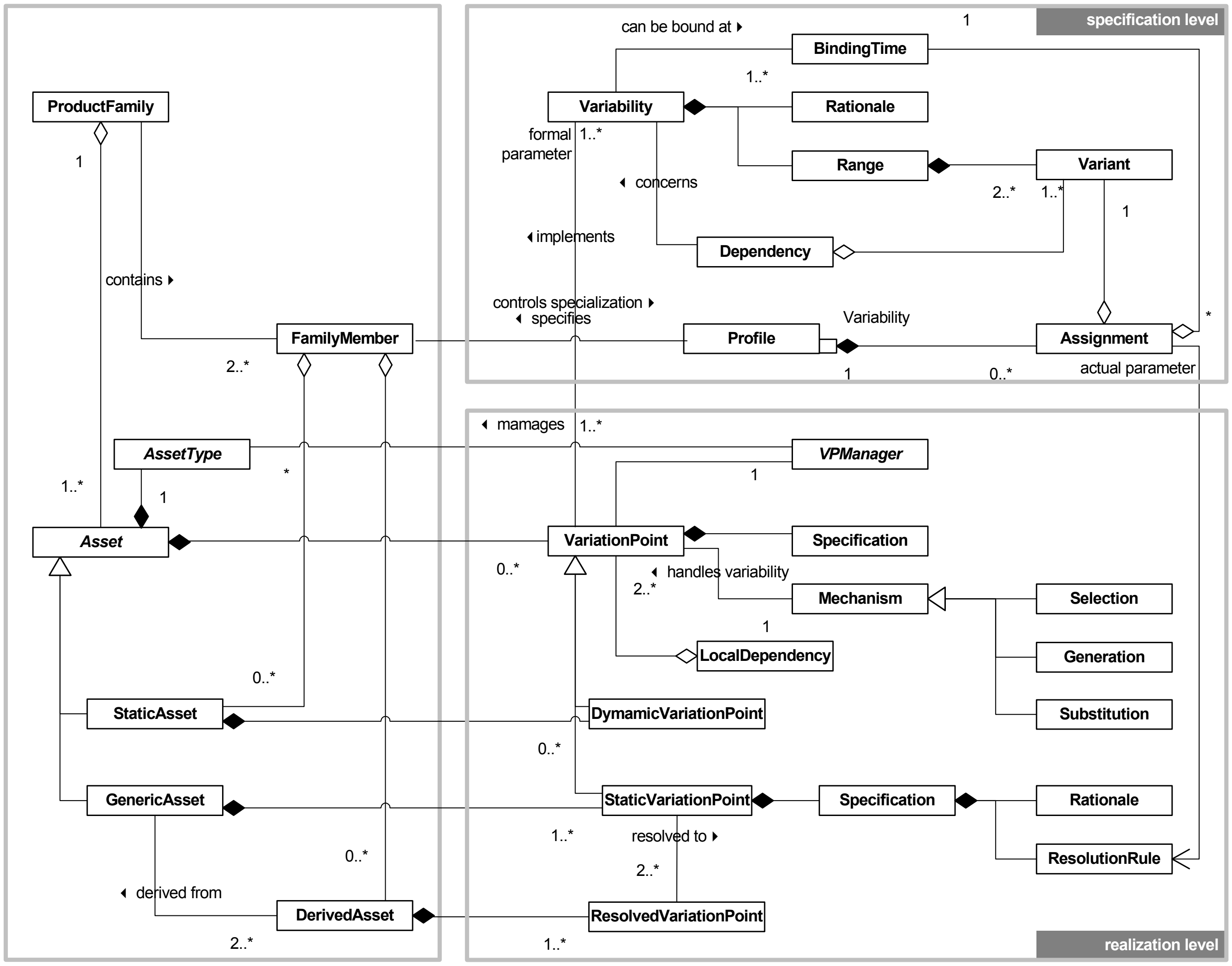


Figure 3. A general model of variability in a product family

ber of dependencies and the effort to manage them as small as possible, dependencies should be specified globally on the specification level, if possible. Dependencies that result from the fact, that VariationPoints realize the same Variability, do not have to be expressed explicitly, they can be derived from the association between Variability and VariationPoint.

A VariationPoint is associated with a Mechanism that handles the Variability. Various Mechanisms can be used to this end. The Mechanisms can be coarsely[9] categorized into three classes [5][6]: Selection, Generation and Substitution. By means of a Selection mechanism, an existing solution can be selected to specialize the variation point. The corresponding specification of the specialization is illustrated in fig. 4. Exemplary selection mechanisms are if/else or switch constructs in preprocessor and programming languages, or inheritance in object oriented languages. A generative mechanism allows the generation of a solution, e.g. through an external generator. The specialization specification forms the input of the generator and the generated output specializes the variation point. Substitution mechanisms are rather simple; they support the specialization of the VariationPoints by unique, externally provided solutions. Therefore, the corresponding variation points can be considered as some kind of gap.

As stated in section 2, two different motivations can be identified for a Variability. Those motivations lead to different types of VariationPoints. The first one, the *DynamicVariationPoint* demarcates a solution in an Asset that allows to handle the Variability late in the lifecycle of the product, i.e. after the delivery. Consequently, DynamicVariationPoints are not specialized during the design of the corresponding FamilyMember. In contrast to them, a *StaticVariationPoint* has to be specialized during the design and implementation of the FamilyMember. The result of such a resolution is a ResolvedVariationPoint, which no longer supports variation. In order to support

[9] more detailed taxonomy of such mechanisms can be found in [17]

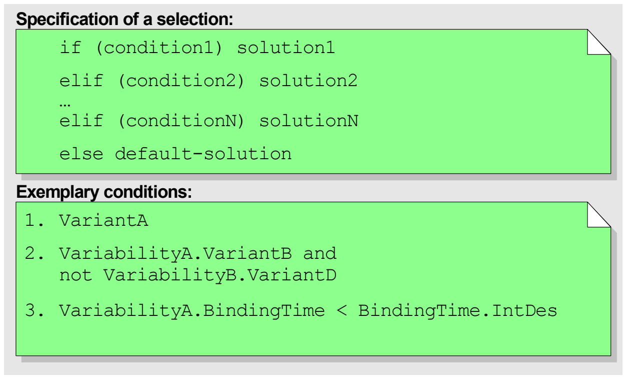
their specialization, StaticVariationPoints provide a Specification, which contains a Rationale and a ResolutionRule. The specialization can be automated through an appropriate mechanism. To facilitate the evolution of a variability realization, the association between StaticVariationPoint and ResolvedVariationPoint should be maintained in the ProductFamily, in order to propagate changes in both directions.

StaticAssets contain no StaticVariationPoints. Thus, they can be used in the application engineering without any specialization. GenericAssets on the other hand contain at least one StaticVariationPoint. The specialization of a GenericAsset results in a DerivedAsset that is used to construct the FamilyMember. DerivedAssets contain no StaticVariationPoints but only ResolvedVariationPoints.

Variabilities control as formal parameters the specialization of the VariationPoints. What serves as actual parameters depends on the type of the VariationPoint. In the case of a DynamicVariationPoint, the specialization is controlled by runtime parameters in the software system. With StaticVariationPoints the assignments in the profiles form the actual parameters of the specialization. If the ProductFamily supports several BindingTimes for a Variability, then the specialization specification of the resulting variation points may also depend on the variability's binding time, e.g. the conditions in a selection (cf. exemplary condition 3 in fig. 4. above). Hence, the variation point's specialization specification is not only a function of the corresponding variabilities but also of their actual binding times.

As illustrated in the model, the only two associations between concepts on both levels are the implements association between Variability and VariationPoint and the association between the Assignments and the ResolutionRules. The first association is established during the implementation of the assets and has to be maintained during the whole lifecycle of the ProductFamily. Along this association, information can be propagated between the both abstraction levels. The second association does not need to be maintained explicitly. It can be derived from the first one. If the actual parameters have to be determined for the specialization of a StaticVariationPoint, then the corresponding assignments can be retrieved from the profile through the variabilities associated with the VariationPoint. Obviously, the first association is of utmost importance for any product family approach. Bidirectional traces between the variabilities and the variation points must be expressible and maintainable in an efficient way. As a prerequisite, the variation points – static as well as dynamic ones – must be identifiable in the assets.

To support the management of variability on the implementation level, VPManager instances can and should be provided for the different AssetTypes of a ProductFamily. A VPManager is a tool that supports the domain and application engineers in the various variability-related tasks, as implementation, identification, resolution, assessment, and evolution of variation points in assets of the respective types. The VPManager class in the model captures the management-related issues and solution patterns or principles, e.g. the resolution in case of variable binding times or the automated evolution of a variabilty. A lot of methodical and tool support is conceivable and required to this end, but only few is available yet.


Specification of a selection:

```
if (condition1) solution1
elif (condition2) solution2
…
elif (conditionN) solutionN
else default-solution
```

Exemplary conditions:

```
1. VariantA
2. VariabilityA.VariantB and
   not VariabilityB.VariantD
3. VariabilityA.BindingTime < BindingTime.IntDes
```

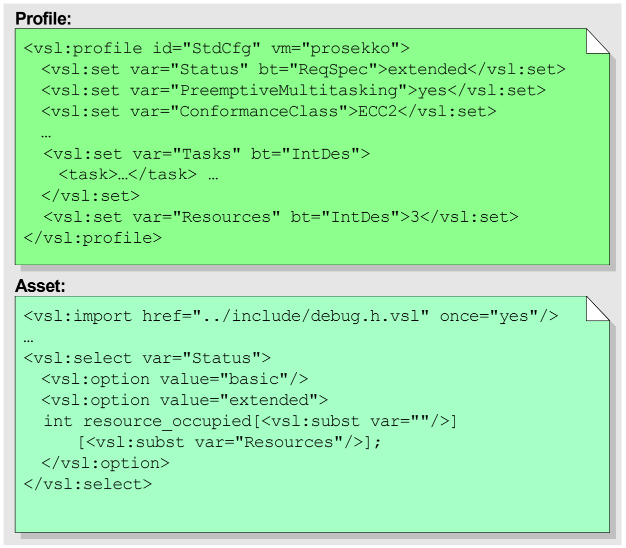
Figure 4. Specification of a selection

5. Instantiation of the Model: Variability Specification Language

Based on the above-mentioned meta-model and the identified demands for variation points, we have developed a language to specify variability in product family assets – the Variability Specification Language (VSL) – and appropriate tools (processor, viewer). VSL is an XML-based language that can be applied in a broad range of documents and thus allows to handle variability in a uniform manner. Besides the previous drivers, VSL has been inspired by the frame technology [3] and the popular C preprocessor. Both of them can be considered as macro languages and the same applies to VSL – at least partially – too.

VSL first of all allows to specify the impacts of variabilities in the assets, i.e. the variation points. Besides the clear identification of the variation points and the variabilities that affect them, the specialization of the variation points can be formulated as well. To this end, VSL provides markup to specify the selection of pre-built variants and the generation (up to now XSLT and JScript are supported) or the substitution of specific solutions and hence supports the basic mechanisms to handle variability.

Based upon the VSL-specifications, specialized solutions (XML or text documents) can be derived from the VSL-based generic assets during the application engineering. This resolution is controlled by profiles, which can be expressed by means of VSL too (cf. fig. 5). Besides the values of the variabilities, VSL specifications can take the variabilities' binding time into consideration. Although the main driving force behind VSL was to support static variability, VSL can be applied with dynamic variability



Profile:

```
<vsl:profile id="StdCfg" vm="prosekko">
  <vsl:set var="Status" bt="ReqSpec">extended</vsl:set>
  <vsl:set var="PreemptiveMultitasking">yes</vsl:set>
  <vsl:set var="ConformanceClass">ECC2</vsl:set>
  …
   <vsl:set var="Tasks" bt="IntDes">
    <task>…</task> …
  </vsl:set>
   <vsl:set var="Resources" bt="IntDes">3</vsl:set>
</vsl:profile>
```

Asset:

```
<vsl:import href="../include/debug.h.vsl" once="yes"/>
…
<vsl:select var="Status">
  <vsl:option value="basic"/>
  <vsl:option value="extended">
   int resource_occupied[<vsl:subst var=""/>]
      [<vsl:subst var="Resources"/>];
  </vsl:option>
</vsl:select>
```

Figure 5. A VSL document and profile fragment

as well. In this case, the VSL markup is not processed by the VSL-processor, but merely serves for identification and specification purposes. A more detailed discussion of the VSL features can be found in [8].

The main advantages in applying VSL to specify variability in a product family can be seen in the uniform and explicit treatment of variability. First, the language can be used to specify the variability in the different asset types. This considerably eases the development of special variability management tools, e.g. to facilitate the evolution of variability, that can be applied throughout the whole product family engineering process. Second, due to the explicit specification of the variability by means of a dedicated language it gets quite easy to identify and assess the impacts of a variability down in the assets. A general advantage of VSL – as with all XML-based approaches – is the extensibility of the language and the remarkable tool support. Although still being in a evolving state, VSL has already proven the feasibility of XML-based variability management. It has been deployed successfully to handle the variability in an embedded operating system on the requirements and the code level (C-Code). In an industrial context we have deployed VSL to specify variability on the architecture level in UML-diagrams.

6. Conclusion

The increased amount of variability in software systems meanwhile requires more systematic approaches to cope with the rising complexity introduced through variability. This is especially true in product families, where variability is a means to handle the inevitable differences among the systems in the family while exploiting the commonalities. Widespread impacts of variability and the various realizations considerably complicate the management of variability in product families. In order foster more systematic and consequently more efficient approaches of variability management we have discussed the commonalities and differences of variability in product families, identified appropriate concepts and interrelated them in form of a general model of variability in product families. The model has been applied to develop a small language to specify and realize variability in product family assets.

We believe that the management of especially static variabilities, which can be considered as a main characteristic of product family approaches, is an issue that can and should be addressed in an explicit and overall manner to keep track with the rising complexity. To achieve this, a common understanding and management of variability is required across the various asset types. The presented approaches intent to pave the way towards this.

References

[1] Bandinelli, S.: Product Family Engineering with XML, Proc. of Dagstuhl Seminar No. 01161 Product Family Development, Wadern, Germany, 2001

[2] Bachmann, F.; Bass, L.: Managing Variabilities in Software Architectures, Proc. of 2001 Symposium on Software Reusability, Toronto, Ontario, Canada, May 2001

[3] Bassett, P.G.: Framing Software Reuse - Lessons From the Real World, Yourdon Press Computing Series, 1997

[4] Bass, L.; Clements, P.; Donohoe, P.; McGregor, J.; Northrop, L.: Fourth Product Line Practice Workshop Report, http://www.sei.cmu.edu/publications/documents/00.reports/00tr002.html, November 1999

[5] Baum, L.; Becker, M.; Geyer, L.; Molter, G.: Mapping Requirements to Reusable Components using Design Spaces, Proc. of IEEE Int'l Conference on Requirements Engineering (ICRE 2000), Chicago, USA, 2000

[6] Becker, M.: Generic Components: a symbiosis of paradigms, 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00), 2000

[7] Becker, M.; Geyer, L.; Gilbert, A.; Becker, K.: Comprehensive Variability Modelling to Facilitate Efficient Variability Treatment, Fourth International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 2001

[8] Becker, M.: XML-Enhanced Product Family Engineering, Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology (IDPT2002), Pasadena, USA, June 2002

[9] Bosch, J.: Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley, 2000

[10] Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, H.; Pohl, K.: Variability Issues in Software Product Lines, Proc. 4th Int'l Workshop on Product Family Engineering, Bilbao, Spain, 2001

[11] Czarnecki, K; Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000

[12] Geyer, L.; Becker, M.: On the Influence of Variabilities on the Application Engineering Process of a Product Family, Proceedings of the 2nd the Second Software Product Line Conference, San Diego, USA, 2002

[13] Jacobson, I.; Griss, M.; Jonsson P.: Software Reuse - Architecture, Process and Organisation for Business Success, ACM Press / Addison-Wesley, 1997

[14] Jazayeri, M.; Ran. A; Van der Linden, F.: Software Architecture For Product Families: Putting Research into Practice, Addison-Wesley, May 2000

[15] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990

[16] Muthig, D.; Atkinson, C.: Model-Driven Product Line Architectures, Proc. of the Second Software Product Line Conference, LNCS 2379, Springer, San Diego, USA, August 2002

[17] Svahnberg, M.; Van Gurp, J.; Bosch, J.: A Taxonomy of Variability Realization Techniques, Technical paper, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002

[18] Thiel, S.; Hein, A.: Systematic Integration of Variability into Product Line Architecture Design, Proceedings of the Second Software Product Line Conference, LNCS 2379, Springer, August 2002

[19] Van Gurp, J.; Bosch, J.; Svahnberg, M.: On the Notion of Variability in Software Product Lines, Proceedings of WICSA 2001, August 2001

[20] Voget, S.; Angilletta, I.; Herbst, I.; Lutz, P.: Behandlung von Variabilitäten in Produktlinien mit Schwerpunkt Architektur, Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1),, Kaiserslautern, Germany, November 2000

[21] Withey, J.: Investment Analysis of Software Assets for Product Lines, http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.010.html, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1996

Managing Infinite Variability

Alessandro Maccari[1], Anders Heie[2]
[1]*Nokia Mobile Software, P.O. Box 100, FIN – 00045 NOKIA GROUP (Finland)*
[2]*Nokia Mobile Phones, 12278 Scripps Summit Dr., San Diego, CA, 92131(USA)*
alessandro.maccari@nokia.com, anders.heie@nokia.com

Abstract

Managing variability is an increasingly challenging task for mobile terminal manufacturers: as new features are launched more and more quickly and the market saturates, coping with variability at the software architecture level is a crucial need for all the companies that operate in the field.
Variability originates from different requirements and features. The need to offer a varied range of terminal to attract different categories of users makes for almost infinite combinations. We analyze the main challenges that lie behind the variability problem, both at the technical and at the organizational level, and illustrate the solutions we have implemented in our organization. We also hint to some suggestions for further research.
In general, the variability problem is likely to increase in complexity, and the ability to successfully tackle it is likely to be a strong factor for success or failure for all companies that want to develop and maintain a product family that addresses different categories of customers and varied regional markets.

1. Introduction: coping with infinite variability

As the market matures, and the competition on price and features becomes harsher, mobile terminal manufacturers are posed with challenges of increasing complexity. We estimated that, in order to maintain market share and avoid profit margin erosion, Nokia must launch and market between 30 and 40 new products every year. In other words, this means launching one product every six to eight working days.

Naturally, diversification among products is necessary in order to address the needs of different categories of customers. Products must be segmented according to the following (rough) criterion.

Low-end products have a simple (often trivial) user interaction pattern, and their features tend to be oriented towards gaming, messaging and multimedia, since that is what the target category (mainly youngsters) are willing to pay for.

High-end products have an abundance of features, and the hardware is typically smaller and more lightweight than in low-end products. The target audience is the "gadget savvy".

Fashion products are designed to cater to the latest, hippest, coolest. An extensive selection of features is essential, as well as the ability to customize the product to suit the needs of the owner, who typically wishes to diversify from the rest of the crowd. Such customization can be hardware-centered (e.g. with plastic covers of different colors), but also software-driven (e.g. capability to support downloadable ring tones, screensavers, etc.).

Business products supply the professionals with tools to help them in their daily jobs. Features such as long battery duration, data synchronization and support for connectivity (email client, infrared, Bluetooth, etc.) are a must.

Moreover, across products, several features introduce variability.

1) A varying number of keys are necessary for both practical and segmentation reasons. For instance, a high-end phone that can browse the web requires more keys than a low-end voice only phone, in order to allow an easy mouse-like navigation.

2) Varying display size and color depth: there are obvious price benefits to black and white displays with respect to color, and the market has expressed no requirement for color until multimedia features (such as multimedia messages, or MMS) were launched.

3) Different feature sets: we try to launch new features very frequently to encourage product replacement (a fundamental drive for profits in a saturated market, such as Western Europe), and create different sets of features for each product segment to suit the target customer needs.

4) Languages: we have to support a very large amount of languages. While most users utilize only one, a typical terminal supports between 4 and 12. The active language must be changeable at run-time.

5) Input methods: these are tightly related to languages. While most Western languages have a standard input method, some (especially Asian) languages have so many glyphs that it is impossible to map each one to a separate key.

6) Backwards compatibility to accessories: mobile terminal manufacturers get a huge economical benefit by being able to mass-produce accessories (such as batteries, handsfree sets, chargers), so that each battery model can be fit in several products. This also increases customer loyalty, as end users are able to use older accessories with newer products, thus minimizing the cost of terminal replacements.

7) Different protocols: a basic necessity, as this defines the network connectivity. Among others, we must support the following protocols: GSM, CDMA, TDMA, PDC, AMPS, some of which work in different frequencies. In some cases we need to support combinations of different protocols in the same product. Since the way network services (e.g. call, messaging) work slightly changes with the protocol, this impacts the user interface (UI) in subtle ways.

These variability points impact almost everything in the terminal. Menus will change if the network changes. The architecture of our software must support different languages. The UI must be compatible with all the input methods. The introduction of a new key will impact the way the UI translate key presses into glyphs.

The challenge is further complicated by the fact that each feature must be:

a) Configurable (on, off, various settings). For instance, the number of characters supported in text messaging using the CDMA protocol is different than for GSM, while some protocols (like AMPS) don't even support text messaging at all.

b) Able to change behavior after product release, typically because of operator requirements. As the market is becoming more mature, operators are looking for ways to differentiate their service from those offered by the competition, and this feeds back to Nokia as operator-specific features and behavior. To simplify production, this variability is most often built into the product at the time of manufacturing.

c) Plug-and-playable. There are two aspects of this. I) Internally, Nokia benefits from being able to quickly add or remove features from products. It cuts our development cycles, and facilitates reuse. A CDMA product can use the same features originally designed for a GSM product. II) Externally, Customers benefit from being able to download applications and run them regardless of the terminal model that they are using. This also creates a large aftermarket that everyone benefits from.

All the business requirements that we illustrated above must be fulfilled in order to allow companies like ours to remain competitive. However, they make the variability challenge very hard to tackle at the software architecture level. This is especially true for large organizations that operate in a continuously changing environment and where product development is distributed among different sites located in different countries or even continents. Practically, the potential combinations of different features are so many that we often talk about "infinite diversity" or "infinite variability". This needs to be managed in an effective way, avoiding the danger of losing control of the software development organization and exponential increase of the workload as the product set augments in size.

In the following sections we report on our experience with Nokia by illustrating the main variability challenges that we must face every day. For each, we outline the solutions we have implemented, with particular emphasis on software architecture issues. We aim to provide an industrial, practical point of view on the problem of software variability. We conclude with some suggestions on a number of issues where we think academic and industrial research should focus.

2. The language challenge

At the time of writing, Nokia's products support approximately 60 languages. Other than Western languages (those based on a Latin character set, such as English, Italian or Danish), these include Arabic, Chinese (which has several variants), Thai and Hebrew. These differ mainly for the input method.

Western languages are typed character by character (one character for every keystroke), and are displayed sequentially from left to right.

Arabic is also entered character by character, but must be displayed sequentially from right to left. The fact that Western words can be inserted amid Arabic text further complicates the matter, since the former are displayed from left to right. Additionally, Arabic letters need to be connected in the display to form a "single sign".

Chinese characters have a different meaning than Western characters. Every character represents a concept, and is made of a sequence of phonetic sounds. Every Chinese character is entered by selecting the sounds that make it up through a series of strokes. This has to be done through some transliteration method, since the keys in the pad correspond to Western characters.

Thai is entered in a similar way than Chinese, but with a different logic.

Hebrew is perhaps the most complicated, since vowels are usually not written, and their transliteration may depend on the context of the phrase.

The popularity of SMS (Short Messaging System) in the recent years has yielded the need for a dictionary. Nokia uses the T9 technology [1], which implements a predictive text input technology, where multiple strokes of the same key to obtain certain characters are no longer necessary. For instance, typing the letter "s" used to require four strokes of the "7" key, but typing a full word (e.g. "same") requires as many characters as there are letters (in this case, four characters, specifically 7-2-6-3). This requires a dictionary to be stored in the terminal, and the input mechanism is further complicated by the fact that some words yield the same sequence of keystrokes (e.g. “same” and “sand” both are typed by entering the above sequence).

The language challenge, however, is not all about inputting and displaying characters in different languages. Most mobile terminals can operate in different languages, i.e. the user can choose which language the menu items, softkey labels and warnings should be displayed. When the user changes the default language, the whole terminal must start operating instantly in the new language (Figure 1).

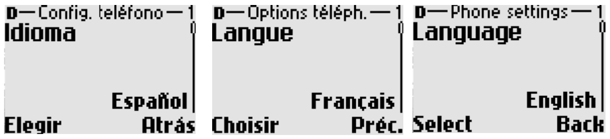


Figure 1. Language change.

Both language challenges are hard to solve in a scalable way if the code is aware of the selected language. Therefore, we had to devise a method to isolate the language knowledge from the code. It is interesting to note that the semantics of each key press does not change when the language changes, nor do the order of the menu items or the functionality thereof. Hence, the language challenge can be formulated in an abstract way: separating behavior (i.e. the semantics of each user action) from appearance (the way the user is allowed to input and is presented output) solves the language problem.

In detail, we solved the language problem with the aid of two artifacts

For text input, we split methods into different components.

a) The physical key press, generated from the hardware, is passed into the visual translation.

b) The visual translation interacts with the user to convert the physical input into a meaningful representation. Several key presses might be necessary before the final logical value (called “glyph”) is complete.

c) The final glyph is the result of one or more physical key presses, translated through the visual representation. This is the interesting part for a software application, as it represents the users intention.

These components allow us to add a new input method (whenever needed), without changing the behavior. Software applications need not know about the physical key presses, or about the translation. Adding a new language is now reduced to the simple exercise of adding a new visual translation.

For text output, we created a text database where every entry corresponds to a string (one or more words). Applications that need to display texts call the database by means of a logical reference, which is independent of the language used. A certain application has the responsibility for managing language selection, and possesses the knowledge of what language is currently active. By means of this, strings in the correct language can be extracted from the database. Note that this allows us to add support for a new language independently from the existing ones.

One additional complication with the output is that the length of strings can differ radically between languages, while the available area does not. Thus care needs to be taken controlling the length of the strings. There are two methods to solve this:

1) Strings can be truncated. This should be done in any case as a precaution.

2) The logical strings must be generated with the knowledge of the available space. This is preferable, as it will ensure the most pleasant UI, but it requires a strong process.

To summarize, the language challenge, as most of the others that will be presented here, can be solved by providing a simple abstraction between the way information is presented and the way it is processed.

3. The hardware challenge

The mobile telephone product concept has evolved massively from the simple, voice-centered products that were in the market in the early nineties. At that time, the display, keypad and hardware features were fairly standard. Nowadays virtually every product has unique hardware features. Here we overview the main variability factors.

a) Keys: products like Nokia’s D111 have no keypad (commands and data are inputted via a connected personal computer, and the terminal acts like a smart modem); “classical” mobile terminals differ in the amount of “soft” function keys (in Nokia’s product family they range from one to three); at the high end of the range, communicator-like products (such as Nokia’s 9210) have a full-fledged, PC-like keypad, complete with some ten “hard” function keys and a few “soft” function keys.

b) Special keys: in some cases, operators or countries request the presence of one or more special keys; an example is the i-mode™ key, which was requested in one of our products to enable users to easily access mobile services in Japan.

c) Scrolling: the small size of displays generates the need for scrolling. Name lists, menu items, received messages, profiles and virtually every other long list of data in the terminal needs to be scrollable. “Classical” terminals are equipped with bi-directional scrolling (supporting vertical directions). However, recently 4-way scrolling was introduced (adding horizontal scrolling), to ease up navigation in tables, such as calendars, and to improve the game playing experience.

d) Sound Playback: ringing tones and games (among other features) require a sound player. Support for MIDI sounds has recently been added to the traditional beeper-style sound that was present in earlier products.

e) Display size: this is perhaps the biggest source of variability. We have made an attempt to standardize display sizes by promoting the Nokia user interface series, where the display size (as well as some other user interface features) is constant for every product belonging to a certain series. For instance, all Series 60 terminals, such as the Nokia 7650, have a 176 x 208 pixel color display [2]. Nevertheless, the variation remains large, and the implications on the user interface software architecture are extensive and perhaps not yet fully understood.

f) Color depth: once, displays were purely black and white, i.e. every pixel could be on (black) or off (white) at any point in time. Gray Scale displays were introduced recently, allowing for several shades of gray. Color displays came next, with different resolutions, just like in PCs. Obviously, applications can use higher color depth to enhance the way they visualize information.

g) Local connections, such as Infrared, Bluetooth, and RS232. Every type of connection that is supported requires its own hardware and software, and must be recognized by all applications that need to use local connectivity.

h) Accessory compatibility: as justified in the introduction, hardware must be kept as backwards compatible with existing accessories as possible (with some obvious exceptions, e.g. when stereo sound output was introduced a new headset was an obvious choice).

Clearly, maintaining software that incorporates all hardware knowledge would mean having too many variation points in the software architecture. While complete hardware abstraction is not possible (and would not be desirable), we need to decouple physical input and output from data management. We will look at the solution we have implemented to overcome this problem in the following sections.

4. The feature challenge

A feature is a chunk of functionality that adds value to the product. Features are normally requested by customers (such as operators or countries). The complexity of today's terminals has boosted the amount of features in the terminal to a level where it's very hard to handle them. The sheer amount of countries and operators we sell to makes for high variability even in the simplest features. For instance, some operators request a separate high-level menu item that facilitates the usage of operator services, or require a different set of call handling features. The growing number of Operator specific changes is one of Nokia's greatest challenges today.

The phenomenon of feature interaction further complicates the matter. In previous publications [3] [4], we have tried to define the problem, categorize the types of interaction and propose some solutions. For the sake of this paper, we will only note that interaction between features can dramatically increase the amount of dependencies between the software components that implement such features. Clearly, software architecture must be designed in a way that such dependencies are minimized, and do not increase exponentially with the number of features.

Also, features evolve and change over time. A typical example is the Phonebook. In the earliest terminals, it was a mere list of names and numbers, where a name could be up to 8 characters and could be associated to only one number. Nowadays, for every phonebook entry (a string which can be made of dozens of character) the user can associate several numbers of different types (home, work, mobile, fax) and even some text (email address, free text notes). Predictably, not all features change and evolve in the same way in all products, which brings additional variability.

5. Solution: client-server architecture

The solution we have devised for this is Client-Server architecture. This is a well-known solution for these kinds of problems, and it suits our case well. We consider our system to be made up of resources (Servers) and user interactions (Clients). A Server represents a basic service in a product, while a Client implements a feature. Clients cannot interact, which means they have no internal bindings. Thus we are able to remove or add a Client without affecting the rest of the System. Furthermore, Clients are designed using another abstraction: UI Components. Preferably, a Client has no direct knowledge of the actual physical representation of its data. Thus a change of the display size would be handled by the UI components, and the Client would never get involved.

While this is what we strive for, it is very hard to obtain that level of separation. We estimate that the majority of our Client code can be left unchanged if the display size changes. This means that most of our code base can remain stable, reducing the possibility of errors. If a service changes, we can typically encapsulate the change in the Server, again reducing the need for the Clients to change to the bare minimum.

Another benefit of the Client-Server architecture is the ability to dynamically add and remove components. Utilizing sophisticated data-transfer models, we can connect a product to another device and run parts of the SW there. This even works across processors, solving known problems with data alignment and endianess (the pattern for byte ordering in native types, such as integers).

6. Solution: decoupled UI architecture

Abstractly speaking, most of the problems we analyzed in the previous sections have one common solution: separating behavior from appearance, or, in other words, enable the decoupling of components from their environment. In our case we have several layers of decoupling. Let's look at an example of a small system as shown in Figure 2.

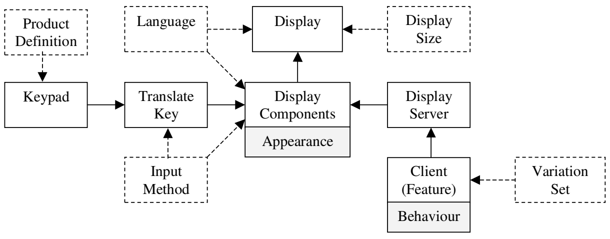


Figure 2. Example of small UI System.

This illustrates the concepts we have been discussing so far, and indicates some of the variability points and where their main impact occurs. As a Client runs, the current variation set determines its functionality. We say the client implements the behavior of a feature. A typical Client will interact with the user, in our case through the Display Server. The Client utilizes a set of display components to show its data. We say these components implement the appearance of the feature.

Thus, we've decoupled the Client from the physical input and output (keypad & display). Although the appearance might change, the behavior stays the same, and vice versa. Abstractly speaking, the behavior defines the data manipulation (logic), while the appearance represents the visualization of the data. This is also known as the observer pattern [5].

We've taken this approach a step further. We allow the Client to not only specify a set of display components, but also basic interactions between them. As it turns out, a large part of a UI consists of simple operations. For instance, playing a beep when a key is pressed, or changing the text on a button when the end of a list is reached. These simple interactions can be described very well by mapping display components input and output together. As the end of a list is reached, a list component can generate an output that can be mapped to a button to change a text. The Client needs not be involved. This creates an even greater separation between appearance and behavior.

Another fact to be considered is that the layout of the display can change from product to product, and even at runtime if the language changes. Layouts are part of the appearance, so we've designed them as dynamic entities that are resolved when the display is updated. The Client never needs to know about the layout in a particular display configuration.

Finally, it is worth noting that the physical input (in our case, mainly the keypad, but also voice recognition, local connectivity, etc.) has no direct interaction with the Client. We allow the input signals (e.g. key presses) to be mapped in the same way as components. Thus, a key press could trigger a text to change or a button to be pressed, all without any behavioral impact. This might sound extreme, but it emphasizes the essence of this model: the behavior constitutes the logical operations performed on a set of data. The appearance constitutes the manifestation of the logical operations. A key press is of no interest to the behavior until is manifests itself as input to a logical operation.

This concept can be hard to grasp without an example. Let's assume that a Client wants to allow a user to write an email. The user can enter the email address, a subject and a body text. He also has an option to press Send & Cancel.

In this case, Appearance includes: entering text using dictionaries or alternate input methods, scrolling in the body text, inserting special characters, moving between fields, beeping when a key is pressed. Behavior, instead, comprises only the following actions: Send, Cancel, verifying the mail address format after it is entered.

Thus, the only interaction between the Display Server and the Client would be to support the three behavior situations. More importantly, the behavior is the same across physical platforms. For another product that does not support beeping when keys are pressed, the behavior of this example client would not change at all. And it should not!

Defining this separation requires great care, but also yields great benefits, in that it allows us to tackle all the scaling-up problems that originate from having to deal with a large and very diversified product family.

7. The organizational challenge and some solutions

As most organizations of its kind, Nokia is a global company. It has regional and global research and development centers scattered across different countries and even continents. The difficulty in this kind of geographical arrangement lies in the fact that, despite all the variability, a large chunk of the software is common between several (or even all) products. Common software must be used in different products, and therefore its changes must be tightly controlled, to avoid undesired propagation of the effects.

The organizational entity that lies at the basis of our software development is called a “software line”. A software line is an organizational entity that is responsible for developing a specific set of features (called “subject area”) for a wide range of products. Examples of well-defined subject areas are Messaging, Phonebook, Calendar, etc. Software lines regularly publish new releases of their code.

This way, the code that every software line publishes has the potential to affect several products. This is why we impose that software lines test their code in as many configurations as possible. Naturally, not all possible future configurations can be predicted at the moment when the code is published, and it could well be that some code release, which has proved to work well with different products in the past, causes errors when integrated into a new product with different functionality. However, requiring fully tested releases certainly minimizes the amount of product-specific testing to be done during integration, thus reducing duplicate work.

The feature interaction (dependency) problem that we outlined before brings another problem at the point of release. Namely, it is important to know what other software lines (components, features) are affected by changes in a certain piece of feature code. We maintain such knowledge in the form of a global database of software dependencies. When a certain chunk of software is changed, the owner software line must look in the database and send information about the change to all the interested parties. We are currently considering the adoption of a tool to perform this task automatically.

In addition, software lines must document all interface and functionality changes. Such documents must be accessible to all products that are affected by the change (i.e. all products that use the code in question). This guarantees that all the products are always up to date with what has been done to the code. Also relevant parties can be invited to reviews, ensuring that no one gets surprised when the new interface is released.

Error management is another crucial area for every global software development organization. Software lines must document known errors in the same way as the code functionality, and ensure circulation of the corresponding documentation (as outlined above). This ensures that all the products that use a certain piece of code are updated on the errors as soon as they are detected. In the same way, error fixes (which usually generate maintenance releases) must follow the same process, and software lines have the responsibility to document them as well.

8. Conclusions and further research

We have outlined the main problems relating to variability that need to be tackled when designing the user interface software architecture for mobile terminals. The main challenges are posed by support for multiple languages, compatibility with different hardware and support for diversified and interacting features. All these problems can be solved by abstracting behavior from appearance, and by decoupling software applications with the services provided by the terminal and the surrounding environment (e.g. network, other connected devices).

Moreover, we shortly discussed the issues that arise when developing software in a distributed, multi-site and global organization, where the dependencies between software entities translate into dependencies between different sections of the organization. We believe that the issues represent valuable input for the research community by providing a practical point of view on large-scale industrial software development.

In the next few paragraphs we digress through some issues for further research in the subject. A more extensive list of issues was presented by one of the authors during a keynote speech at the SPLC-2 conference, in August 2002 [6].

8.1. Designing for complexity

We noticed one interesting practical problem that arose when working with this kind of architecture. Namely, it is generally difficult for developers to work on a system where everything is decoupled. People tend to look at this the practical way, and mainly try to implement a feature in the fastest possible way. In order for this approach to work, it is of course essential that each developer understand the technical aspects. However, this is not enough: every developer must be able to take a step back and define interfaces in very abstract terms. While this can be partly achieved with rules, processes and training, ultimately people must fully understand the underlying concepts (that we explained above) in order to produce efficient code in this framework. If these concepts were included in Software curricula at universities, we believe that the software community would see long-term benefits. We found out that this is often not the case: when seen isolated from the rest, each chunk of code usually is designed in a sensible way. However, when put together, sensible components do not always make a sensible system. Thus, we believe that more training is needed in the subject of designing code for complex software systems that have a lot of variability.

8.2. Assessing the convenience of redesign

A recurring problem in our software development world is to figure out to what extent it is convenient to change a software system as opposed to rewriting it as branches? For instance, suppose we need to implement 5 operator changes that impact ‘a feature’ changing 10% of the code. In that case it is probably convenient to maintain 5 different versions of the component. But how maintainable does that become when the changes impact 50% of the code? Obviously, the code quickly becomes impossible to maintain and errors multiply when too much variability incurs. We have seen no general methods to assess at which point it is economically worth changing and maintaining different versions of a software system, as opposed to rewriting or redesigning it.

8.3. Highlight variability in requirements

The amount of variability in software is dictated by requirements. However, our requirements are fed by numerous business units that operate more or less independently. So far, we have not been able to implement a robust requirements process that allows variability to be transparent straight from the common software requirements. Often, the case is that common and variable features are identified only once design, or even implementation has started. This, of course, increases the amount of work needed to design proper software architecture, according to the principles we discussed above. We believe that research should focus on this issue in a practical setting, i.e. considering the difficulties that arise when working in a large and complex organization, where features and responsibilities change at a high rate.

9. References

[1]: see http://www.t9.com/

[2]: see http://www.forum.nokia.com/html_reader/main/1,32611,2471,00.html?page_nbr=2

[3]: A. Maccari, and A-P. Tuovinen, “System Family Architectures: Current Challenges at Nokia”, *Proceedings of the IW-SAPF-3 workshop, Lecture Notes in Computer Science 1951,* Springer&Verlag, Las Palmas de Gran Canaria, Spain, March 15-17, 2000, p. 107 ff.

[4] L. Lorentsen, A-P. Tuovinen, J. Xu, “Experiences in Modelling Feature Interactions with Coloured Petri Nets”, *Acta Cybernetica* 15(4), Szeged, Hungary, 2002, pp. 621-632.

[5]: see http://c2.com/cgi/wiki?ObserverPattern

[6]: see http://www.sei.cmu.edu/SPLC2/keynote_slides/keynote_1.htm

CAN XML DOCUMENTS BE TREATED AS COMPONENTS?

Kai Koskimies
Tampere University of Technology
Institute of Software Systems
Tampere, Finland
email: kk@cs.tut.fi
(This work was carried out during the author's visit at
the University of Groningen, the Netherlands)

ABSTRACT
Many modern systems make use of components which produce and consume XML documents. Such systems rely on certain structural definitions of the XML documents. However, these structural definitions are often subject to changes and extensions. We argue that the modifications of the system resulting from changed structural definitions of the XML documents are poorly managed with current technology. This is a particularly serious problem in product-line systems using XML technology, where the structural specification becomes one of the variation points of the product platform. We study a possible approach to solve this problem based on the idea of introducing provided and required interfaces for XML documents. This solution makes use of associating attribute grammar like processing rules with XML schema definitions, describing how provided and required services are related in the case of a particular schema.

1 INTRODUCTION

A current trend in information technology is towards global, heterogeneous systems, comprised of different kinds of components and applications interacting with each other directly or over a network. In many cases the interacting parties have been independently developed, and have no previous knowledge of each other. Hence the architectures of those systems must be based on well-defined standards on data transmission between the interacting components. XML (eXtensible Markup Language [W3C02], [Oas02]) provides a natural means for defining such standards. XML is a metalanguage supported by W3C (World Wide Web Consortium), designed originally as a universal format for structured documents and data on the Web. XML is currently used extensively in all kinds of software systems, often as architectural glue integrating components that exchange data expressed in jointly agreed XML format. The technology around XML, including particular XML-based languages and tools for processing XML-documents in various ways, has rapidly expanded and become widely adopted by the industry.

When used as architectural glue in a software system, XML often replaces the static interfaces with dynamic interfaces in the sense that components communicating via XML files have only a very generic static interface, simply allowing the receipt of an XML file. All the parameters affecting the response of the receiving component are given within the XML file, and thus identified dynamically during the parsing of the XML file. This makes systems very flexible and configurable: the functionality of components can be radically changed without affecting the static interfaces.

However, the problem is that the system becomes implicitly dependent on the structural specification (*schema*) of the XML documents. In many cases this is a serious drawback in using XML. For example, if the structural specification of the XML files is changed even a little, the receiving component may or may not work any more, and there is no way of knowing which is true without looking into the code of the component. XML schemata become crucial software artifacts that cannot be changed without the danger of invalidating a number of unknown components that implicitly rely on the structural definition.

A solution to this problem is to use schema extensions, allowed by the current w3c schema standard [W3C02]. This facility makes it possible to build schema hierarchies, analogous to class hierarchies. Thus it is possible to give a "superschema" that is extended by several "subschemas". Any client that is able to process an XML-document according to the "superschema" can also process documents that follow a "subschema". In principle, this allows the extension of a schema without affecting the clients of the original version, thus solving some of the problems originating from schema modifications. However, this kind of schema polymorphism is not a general solution in the sense that it only narrows down the schema dependency, but does not remove it. Any change in the schema concerning the parts a client is interested in necessarily implies changes in the client as well. Hence we need to separate the consumer of an XML-document from the actual schema definition.

The question of managing changes in the schema definitions becomes particularly important in the case of a product-line platform making use of XML. Generally, a product-line platform has a set of variation points, defining the range of supported variation and describing how a particular variant is implemented on the basis of the product-line architecture. If XML is an essential part of the product platform, there should be techniques to define and exploit certain variation points in the XML-schema as well. The schema hierarchies, as explained above, are one way to support this: in that case the "superschema" belongs to the platform, and different applications define their own "subschemata". This corresponds closely to the use of inheritance hierarchies as the basis of variation points in conventional programming. It is a "white-box" specialization approach in the sense that the internal structure of the base schema must be known to the specializer. Exactly as in the case of traditional classes, it becomes difficult to precisely specify how a "subschema" should be given so that it would conform to the product-line architecture. The so-called fragile base class problem [Szy98] of traditional classes becomes even more difficult to manage in the case of XML: any change in the "superschema" can lead to a mismatch in an application.

We argue that a "black-box" approach would be more appropriate for the realization of XML variation points. In this approach, the direct relationship between the schema and the consumer of an XML-document is removed using interfaces. The interfaces express precisely, on an abstract level, what a client can expect of an XML-document, and what the XML-document can expect of its client. In conventional terms, these interfaces correspond to the provided and required interfaces of the "XML-component", respectively.

In this paper we will outline a solution based on introducing interfaces for XML-documents. We emphasize that the proposed techniques have not been tried in a real case study, nor have they been implemented. The main contribution of this paper is the formulation of the problem of variability management in XML, and the discussion concerning the problems and design choices of an interface-based approach. We have aimed at a practically feasible solution, but the usability of the solution still has to be verified.

We proceed as follows. In the following section we will briefly discuss the types of variability problems. In Section 3 we outline a solution to managing variability in XML, based on the idea of viewing XML-documents as components with provided and required interfaces. This solution is further refined in Section 4, showing how provided and required interfaces are interpreted in the context of XML. Some related work is discussed in Section 5, and concluding remarks are presented in Section 6. We assume only superficial knowledge of XML [W3C02] in this paper.

2 VARIABILITY ISSUES IN XML

In principle, variability can appear in two forms as far as XML is concerned: either the schema of an XML-document consumed by a component is allowed to change, or the consumer of an XML-document is allowed to change (or both). These two patterns are illustrated in Figure 1.

To make the variability problems more concrete, assume that an enterprise information system makes use of XML to transmit purchase orders among different, independent subsystems. Since the data represented in purchase orders is sensitive to various changes in the environment, the XML schema experiences many changes during the lifetime of the system. For example, the structure of some data elements may need to be changed. However, assuming that the same logical tasks can be performed for XML-documents following both the new and the old schema, we may still want to use the old XML-documents together with new ones, following the revised schema. Thus we have the situation depicted on the left side of figure 1: a client should consume various XML-documents constructed according to different schemata, being dependent only on the logical operations to be performed on the XML-documents, rather than on their schemata.

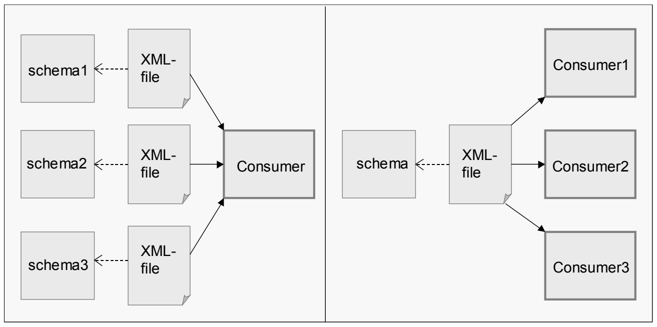


Fig. 1. Variability issues in XML. On the left, a single consumer should be able to process XML-files conforming to different schemas but providing the same logical information; on the right, several consumers should be able to process the same XML-file, varying the actions performed upon the XML data.

On the other hand, assume that several subsystems process same XML-documents, but they perform different actions on certain elements in the documents. For example, suppose that one subsystem sends a purchase order through email, while another simply prints the order. Thus these subsystems will repeat the same or similar XML processing code, but the actions performed on the data elements in the XML-file vary. This corresponds to the situation depicted on the right side of figure 1.

3 OUTLINE OF A SOLUTION

A possible approach to solve the variability problems is to introduce an interface-based type mechanism for XML. This implies that an XML document becomes a component-like entity that conforms to a particular interface. As long as the interface remains the same, the schema of an XML document can be freely changed without affecting the processing of the XML documents by the component. Essentially, the interface defines the assumptions the users of the document can make about the tasks that can be carried out with the document.

On the other hand, a different interface is needed to define the assumptions the XML document can make about the services that help to carry out those tasks. These two types of interfaces correspond closely to the conventional provided and required interfaces of components, respectively. The provided interfaces define the services the "XML-component" can give to its users, and the required interfaces define the callback functions to be called by the "XML-component" when carrying out its services. Naturally, an XML-document (or its schema) can provide and require several interfaces.

Consider again the two types of variability problems discussed in Section 2 (figure 1). The variation point of the first type can be realized using provided interfaces of the XML-document. That is, the XML-documents are viewed as components providing certain services related to their information contents. In this case all the different XML-documents implement the same interface, and only the latter is known to the client component. For example, the service could be "process all the purchase orders by producing statistics on the demand of each product". The XML-documents for purchase orders may follow different schemata, and there may be completely different kinds of XML-documents (say, billing documents) that can provide the same service.

The variation point of the second type can be realized using required interfaces of the XML-documents. The XML-documents are in this case interpreted as components calling the services of other components through a well-defined interface. For example, such a service could be "process a single purchase order". Typically (although not necessarily) the component which provides these services is the same component that calls the services in the provided interface of the XML-document. Thus the client of an XML-document can perform an action on the XML-data, and specialize it by giving its own implementation for the callback function called during the processing of the XML-document.

To summarize, in the case of unmanaged variability, the variation points are scattered throughout the schema specification and the client component processing the XML-document (figure 2a). In the case of extension-based (or inheritance-based) variability management, both the schema and the client are extended with product-specific parts (figure 2b). In the case of interface-based variability management, the fixed part of the architecture consists of the interfaces, while the schemata and the XML clients requiring and providing these interfaces are (or can be) product-specific (figure 2c). In the sequel we will study the interface-based approach in more detail.

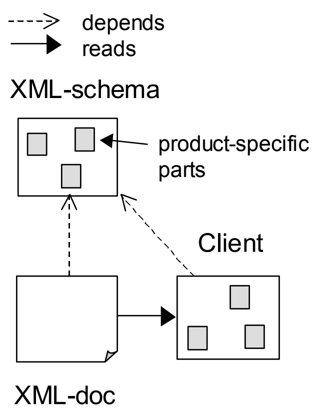


Fig. 2a. Unmanaged variability in XML

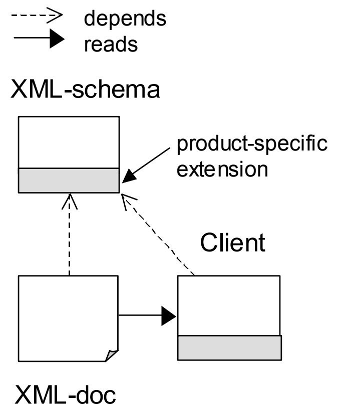


Fig. 2b. Extension-based variability in XML

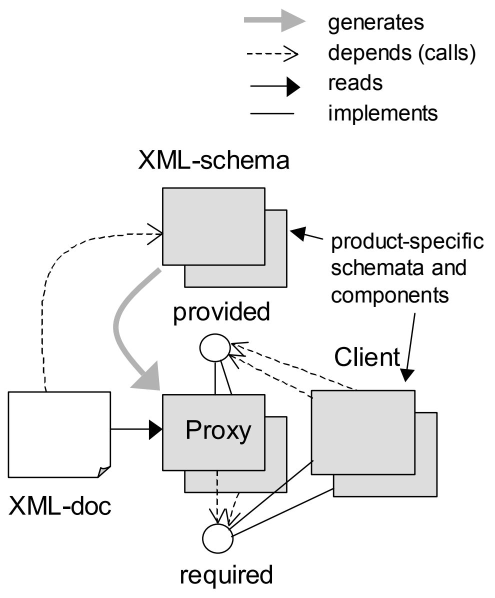


Fig. 2c. Interface-based variability in XML

To be able to attach traditional interfaces to an XML-document, a proxy object is needed that actually implements the provided interface and calls the methods in the required interface (figure 2c). The proxy component is an executable representative of the XML-document in the environment of the consumer; the proxy is actually the “XML-component”. The proxy is generated automatically on the basis of the schema of the XML-document. The consumer calls the "services" of the XML-document by calling the methods of the provided interface of the proxy. On the other hand, the proxy registers the client component, and calls back the client’s services through the required interface.

This solution implies that the architect should figure out the roles an XML-document can play in the system, and present these roles as provided interfaces of the XML-documents. In a platform architecture, these interfaces become a variation point, under which different XML-schemata can be introduced, implementing the same interfaces. If a new logical task emerges for the XML-documents, a new interface must be introduced, and the schema must be augmented with an implementation for that interface. However, if the structural parts of the schema remain the same, the old XML-documents can still rely on the new schema.

We have assumed that it is possible for the XML-document (or its schema) to define how the services of the provided interface are to be implemented making use of the services of the required interface and the information present in the XML-document instance. In principle this problem is analogous to the problem of attaching computation to a hierarchic structure. A solution to this problem has been presented a long time ago: attribute grammars [Knu68]. Attribute grammars associate semantic attributes to the nodes of a syntax tree of a context-free grammar, and rules defining the relationships of the attribute values in the branches. Various methods have been developed to compute the values on the basis of the rules, and to generate efficient evaluators from the attribute grammar. Attribute grammars have been the most successful technique for structure-oriented processing, applied mostly in the realm of compiler generation. Since XML elements can have attributes as well, the idea of applying attribute grammars looks very attractive.

However, in their general form attribute grammars are too clumsy and difficult to use for an average schema writer. We will apply a simplified version of attribute grammars, which is more close to so-called L-attributed grammars [LRS74]. The idea of L-attributed grammars is to restrict the dependencies of attributes in such a way that the evaluation of attributes can be carried out during a single left-to-right, top-down pass over the hierarchical structure. A benefit is that the schema writer can think of an attribution rule as a simple assignment statement executed at a time determined by its position in the structure. This makes the writing of the statements intuitively easier and close to normal programming. Any complex type definition in the schema can be augmented with such statements. The statements are executed in the left-to-right, top-down order with respect to the DOM-tree (that is, the internal object representation of an XML-document produced by an XML parser).

We have now the constituent parts of the solution on a very abstract level: provided and required interfaces, and a mechanism to express how the provided interfaces are implemented using the required interfaces and the data values in the XML-file. We will next refine these concepts.

4 REFINING THE SOLUTION

Provided and required interfaces are specified within the schema. From the viewpoint of the schema writer, a *provided interface* is a set of global output variables, whose values are computed during the processing of an XML-file. For each output variable, there can be any number of input variables, whose values are set by the consumers of the XML-file and used in the computation of the output variable. An XML schema can have several provided interfaces.

Similarly, a *required interface* is a set of global input variables whose values are functions. These values are set by the user of the XML-file. For each input variable, there can be any number of output variables, whose values are computed during the processing of the XML-file. These output variables are used as parameters of the required functions. An XML schema can have several required interfaces. The provided and required interfaces are depicted in Figure 3.

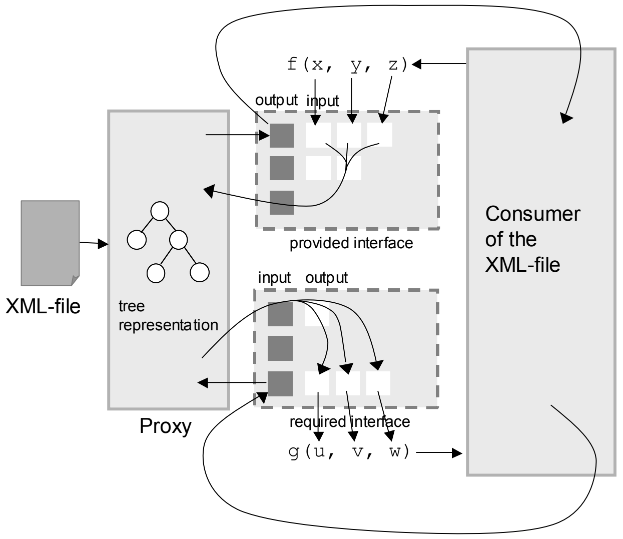


Fig. 3. Provided and required interfaces of an XML-document. Input and output variables are shown with small boxes inside the interfaces.

The rationale behind this kind of interface concept is that the variable-based computation model becomes much simpler than the specification of a function in the context of XML. Since the rules contributing to the computation of a function can be scattered throughout the XML schema, it becomes unnatural to view this kind of computation strictly as a function. Nevertheless, in an abstract sense the output variables of a provided interface correspond to a function providing a value for the users of the XML-file, and the input variables correspond to the parameters of that function. In the case of a required interface the need for a variable-based interpretation is less obvious, but the additional flexibility it brings in the computation of the parameter values for required functions can sometimes be welcome. Symmetry reasons favor this choice, too.

The correspondence between an output variable in a provided interface and a function becomes very concrete in the implementation: a provided interface is eventually mapped to a Java interface which has a function for each output variable. The input variables are in turn mapped to the parameters of that function. This is the reason we group the input variables under a particular output variable. The same applies to required interfaces, the roles of output and input variables being exchanged.

Let us illustrate the implementation of the proxy object with an example. In the case of the purchase order example, the proxy could look as follows:

```
public class PurchaseOrderProxy implements
  PurchaseOrderServices {
  PurchaseOrderSupport support;
  XMLrepresentation doc;
  public PurchaseOrderProxy() {...}
  public void register(PurchaseOrderSupport
    client) {
    support = client;
  }
  public void readXMLfile(file f) { ... }
  public void processOrders() {
    ...
    support.handleOrder(doc.getOutput("price"),
                        ...);
    ...
  }
  public Integer totalValueForArea(Positive
    areaCode) {
    doc.setInput("areaCode", areaCode);
    ...
    return doc.getOutput("totalValueForArea");
  }
}
```

In this case the schema has defined output variables `processOrders` and `totalValueForArea`. For the latter, there is an input variable `areaCode`, which becomes a parameter for the function. Initially, the client component (`support`) is registered for the proxy, and the XML-document is parsed into an internal representation (`doc`) using the appropriate functions of the proxy. In the body of the function `totalValueForArea`, input variables are given initial values for the processing of the XML-document. Then the internal representation is traversed, and the computation rules are executed. These rules compute the value of the output variable `totalValueForArea`, calling the operations of `support` when determined by the rules. In the example, the provided operation `processOrders` calls one of the operations of the required interface, `handle-Order`, using the output variables of the required interface as parameters. Finally, function `totalValueForArea` returns as its value the final value of the output variable of the provided interface.

Let us next study how the computation rules are given in a schema in more detail. We will not discuss their concrete XML form here, but instead discuss the main principles they follow. A possible concrete form of the computation rules is presented in [Kos03]. This part requires some knowledge of XML terminology.

A computation rule is always given in a *context*. A context is a complex type (that is, a structural type) definition in an XML-schema; a computation rule is given as a subelement of the complex type that serves as its context. The *left context* of a computation rule consists of the attributes of the subelements preceding the computation rule in the complex type definition; the *right context* consists of the attributes of the subelements following the computation rule. In addition, the attributes of the complex type itself belong both to the left and to the right context.

A computation rule takes the form of an assignment, given as the value of a particular attribute of a rule element. The left hand side of a rule is an attribute belonging to the right context of the rule, or an output variable. The right hand side is an expression consisting of attributes belonging to the left context of the rule, or input variables. As customary in attribute grammars, we allow simple arithmetic operations on the right hand side. If an input variable denotes a function, the conventional parameterized notation can be used as well; in that case the actual parameters are assigned to the corresponding output variables before executing the function. A computation rule can also be conditional, executed only if a given boolean expression is true. The left hand side of a computation rule can be omitted.

Note that here we deviate from the classical L-attributed grammar by treating attributes simply as variables, instead of dividing them into inherited and synthesized single-valued data containers. However, we do retain the left-to-right direction of data flow characteristic to L-attributed grammars. In principle, we could give up this restriction and allow arbitrary data flow between the attributes in the context of a rule: we could simply state that the rules are executed in the left-to-right, top-down order, and leave it to the schema writer to ascertain that the sequence of assignments makes sense. However, the left-to-right data-flow makes the computation safer in the sense that the attributes of an element are not used before the subtree rooted by that element is processed. Thus, the schema writer can imagine that some of the attributes in the root represent the "result" of processing the subtree. Note that it is still possible that some attribute does not always get a value, or that some attribute is assigned many times. Tool support should be provided to statically check the rules and warn about these cases.

5 RELATED SOLUTIONS

An even more refined extension model for XML element types is introduced in XInterfaces [Nöl02]. This model is based on the idea that each client has its own view on the data in an XML-document, defined by itself. A type extension mechanism guarantees the conformance of the extended types with existing views. The main difference between XInterfaces and our proposal is that we define the view of a client as a normal programming interface, while in XInterfaces the views are still XML element types. Our mechanism introduces more complete isolation of the client from the XML-schema, but also deviates more from the XML world.

Not surprisingly, the integration of attribute grammars and XML has been already studied in few papers ([PC-R99], [Fer01]). However, the aim of these papers is different: they regard attribute grammars as a general mechanism to add semantics to XML. This allows, for example, the presentation of stronger semantical validation conditions in the schema, which is one of their primary motivations.

6 DISCUSSION

We have presented a solution outline for making XML-documents first class architectural elements that comply to normal interfaces. We strongly believe that this is a problem that has to be solved one way or another. On one hand, the use of XML is steadily increasing. At the same time, various kinds of software platforms or product-line architectures are becoming more and more common in many domains, emphasizing the issue of variability management. These two trends make it necessary to develop techniques for variability management in XML as well. In this respect the current level of technology is far from satisfactory.

Our proposal follows the line of thought in which provided and required interfaces are seen as contracts between software components. We have adopted an approach in which it is the duty of the XML-schema to define how provided services are obtained using the required ones. This approach leads to the introducing of some level of processing capability within the XML-schema. We feel that the most natural existing model for this is the concept of an attribute grammar. However, we have reduced the needed processing facilities to the minumum, trying to avoid excessive complexity.

There are still many open questions, and the applicability of the approach has to be tested in real case studies. We have also not yet defined a full schema language based on this idea; thus we cannot say to what extent some features of, say, the W3C Schema language [W3C02] contradict with our model. These are our next steps in this research.

Tool support is one of the open questions. Although the principles of the proxy generation tool seem fairly straightforward, it is possible to apply various optimization techniques to improve the performance of the provided operations. It would also be desirable to build special editing support for the schema specification; for example, an editor could present for each output variable of the provided interface a slice of the schema that contains only those parts that are relevant for that variable.

Finally, it should be noted that our solution is actually not specific to XML. We have shown that in principle any grammar-based language specification can be augmented with interfaces in such a way that instances of the language can be treated as components conforming to those interfaces, using a proxy. However, the technique is beneficial if the language is subject to change and the rest of the system should not be affected by the changes.

ACKNOWLEDGEMENTS

This work has been financially supported by the Academy of Finland and NWO (the Netherlands).

REFERENCES

[Fer01] Ferenc H.: XML Semantics Extension, in *Proc. of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001),* Szeged, Hungary, 2001.

[Knu68] Knuth D.E.: Semantics of context-free languages. *Mathematical Systems Theory* 2 (1968), 127-145.

[Kos03] Koskimies K.: A Technique for Variability Management in XML. To be presented in the Second ASERC Workshop on Software Architecture, Banff, Canada, February 2003.

[LRS74] Lewis P.M., Rosenkrantz D.J., Stearns R.E.: Attributed Translations. Journal of Computer and System Sciences 9 (1974), 279-307.

[Nöl02] Nölle O.: XInterfaces – A new schema language for XML. Diploma thesis, University of Freiburg, Germany, June 2002.

[Oas02] http://xml.coverpages.org/, Oasis 2002.

[PC-R99] Psaila G., and Crespi-Reghizzi S.: Adding semantics to XML. In *Proc. Second Workshop on Attribute Grammars and Their Applications (WAGA99)*, Amsterdam, The Netherlands, March 1999, pp. 113-132. http://wwwrocq.inria.fr/oscar/www/fnc2/WAGA99/proceedings/psaila /psaila.pdf.

[Szy98] Szyperski C.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley 1998.

[W3C02] http://www.w3.org/XML/, W3C 2002.

--TO APPEAR--

version 1.3 -- February 3, 2003
Journal of Network and Computer Applications 24(??), January (?) 2001

Extensibility via a Meta-level Architecture

Serge Demeyer
Lab on Reengineering (LORE)
University of Antwerp, Department of Mathematics and Computer Science
Universiteitsplein 1, B-2610 Wilrijk (Belgium). *Tel:* ++32 (0) 3 820 24 14. *Fax:* ++32 (0) 3 820 24 21.
E-mail: serge.demeyer@uia.ua.ac.be. *WWW:* http://win-www.uia.ac.be/u/sdemey/

Abstract. Meta-level architectures are recognized as a means to achieve run-time extensibility, and have been applied as such in existing hypermedia systems. Yet, designing a good meta-level architecture is notoriously hard and remains an art rather than a science. This paper shows how to derive a meta-level architecture for hypermedia navigation, thereby providing a way to control how third-party components interact with the linking engine. This extra level of control allows for a better and safer integration between an extensible system and the third-party components extending it.

1. Introduction

Nowadays, a considerable amount of effort is spent on the design of extensible systems. This phenomenon can be observed in fields such as operating systems, databases, inter-operability standards, programming languages and —last but not least— hypermedia. The web has most certainly been an aggravating factor in the search for hypermedia extension mechanisms, especially enforcing the need for *run-time* extensibility [1].

Run-time extensibility implies that a deployed system may extend its capabilities by allowing users to plug in extra *third-party components*. During certain operations, the deployed system hands over control to a third-party component, trusting that the component will return control when required. This relation of trust is the Achilles heel for all run-time extensible systems, because there is always the risk that the system loses control over the operation and consequently arrives in an inconsistent internal state.

As an example of what might happen when a hypermedia system loses control over the navigation operation, consider the typical case of a web-browser extended with a third-party application for viewing PDF files. A PDF document might itself contain hyperlinks, some of them pointing to external and some of them to internal locations within the document. Unfortunately, only the activations of external links pass through the link engine of the web browser and consequently activations of internal links will leave the log of navigation actions in an inconsistent state. As a result, pressing the ‘back’ button on the web browser will not always return the reader to the expected location which is confusing and adds extra cognitive overhead.

One way to avoid the Achilles heel of run-time extensible systems is to secure this relationship of trust by means of a *meta-level architecture*.Via a meta-level architecture, a system is able to watch over its inner actions regardless of the components involved, thus making it possible to adapt the internal representations accordingly. To achieve this self-awareness, a system with a meta-level architecture (see the architectural pattern "Reflection" in [2]) provides two separate interfaces: the *base-level interface* —which provides the usual way of accessing the systems functionality— and the *meta-level interface* —which provides an interface for inspecting and changing aspects of that system behaviour. In the example of the extensible web-browser, the base-level interface allows third-party applications to invoke operations on the link engine, while the meta-level interface allows the web-browser to examine all of them and thus ensure the navigation log remains consistent.

Today, meta-level architectures have become part of the standard repertoire of programming techniques. For example, Java, CORBA and ActiveX all provide meta-level interfaces for checking object types and interfaces and sometimes even for dynamically invoking object operations. Thus, it should not come as a surprise that hypermedia systems as well have been incorporating some form of a meta-level architecture. Hyperform for instance, is a hyperbase where the set of services provided can be extended using a meta-level interface [3]. As a second example, the DHM system incorporates a so-called "embedded interpreter" to allow end-users to extend the functionality of the hypermedia engine [4]. And recently in the context of the web, the XML standard exploits meta-languages as a way to extend the set of document types understood by web browsers.

Yet, even though meta-level architectures have proven their value in practice, designing a "good" meta-level interface is notoriously difficult. First, it is difficult to predict the functionality that must be provided in the meta-level interface. Second, it is difficult to establish a clean separation between the base-level interface and the meta-level interface.

This paper derives a generic meta-level architecture for hypermedia link engines based on two design guidelines, namely "turn contracts into objects" and "turn the configuration into a factory object" (section 2.). Next, we show how the meta-level architecture makes it possible to dynamically extend the way a hypermedia system logs navigation actions, arguing that the design guidelines indeed provide a "good" meta-level interface (section 3.). Finally, we discuss how we validated our claims and explain the pros and cons of meta-level architectures as an extension technique (section 4.).

2. Deriving the Meta-Level Architecture

This section provides a practical illustration of the derivation process for the meta-level architecture by first specifying an object protocol for a generic navigation operation and then extending that protocol with the necessary contracts in the form of pre- and postconditions. Next, we apply the two design guidelines to derive the actual meta-level architecture.

2.1. Generic Navigation Operation

To validate the practical applicability of the derivation process, we show how to derive a meta-level architecture for a hypermedia navigation operation. We base ourselves on the well-known Dexter specification [5], although we certainly do not restrict ourselves to Dexter compliant systems. In fact we argue that this navigation operation is representative for many of the hypermedia systems in use today, thus that the navigation operation is indeed generic.

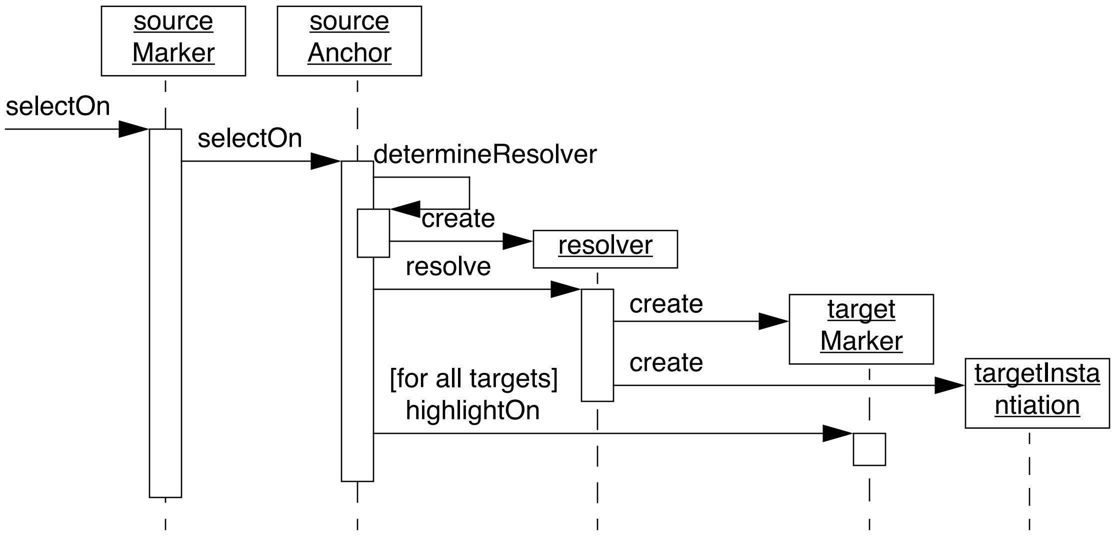


Figure 1 The object protocol for the generic navigation operation

As shown in Figure 1, the generic navigation operation starts by invoking the `selectOn` operation on a marker (the object named `sourceMarker` representing the visible part of the link), which forwards this operation to its associated anchor (the object `sourceAnchor` representing the persistent part of the link). This anchor infers the targets of the navigation by invoking `resolve`, and then invokes `highlightOn` on all resulting pairs of markers and instantiations (objects `targetMarker` and `targetInst`).

Applying this to a web browser, the `sourceMarker` represents the visible part of a link anchor (typically a bit of blue underlined text) while the `sourceAnchor` corresponds to the URL embedded in that marker. The resolve function then interprets the URL and creates objects representing the target of the navigation, thus `targetInst` (representing the target document) and `targetMarker` (representing the target location within that document). Finally, the `highlightOn` operation displays the target document in the browser and scrolls to the appropriate location.

The same design might also be used for traversing "generic links" as defined within MicroCosm [6] and its derivatives (see among others [7] for a discussion on the use of generic links in multi-media information). The `sourceAnchor` object then corresponds with the "tagged link description", holding various fields describing the contents and location of the selected piece of information in the source document. The `resolve` function passes this information through a number of "filters", where each filter matches the `sourceAnchor` against its own linkbase and adds or removes navigation targets to or from the result.

The generic navigation operation may also serve as a basis for the structural computing paradigm as advocated by the HBn/SBn series of hypermedia systems [8]. In such a case, both the `sourceAnchor` and the `resolve` function correspond with structural computations ("Sprocs" in HBn/SBn terminology) while the `sourceMarker` holds the input data for these computations. We have used such structural computations to build source code browsers in programming environments [9], [10].

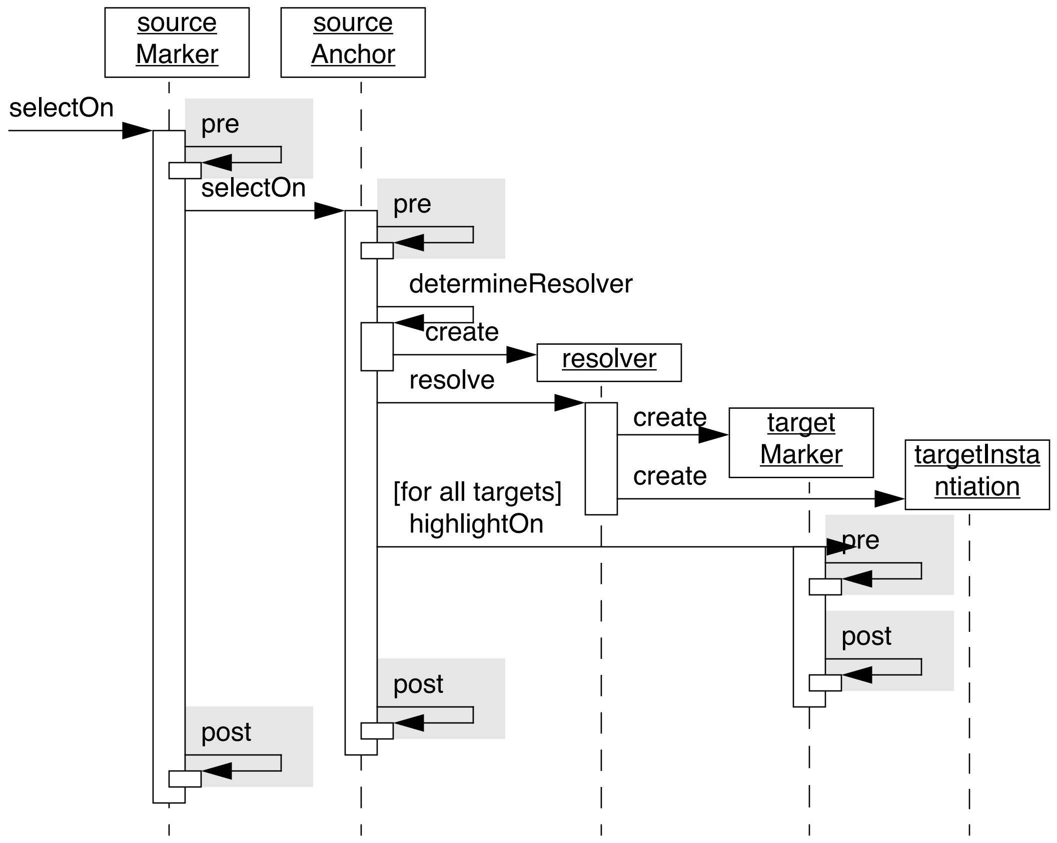


Figure 2 The extended protocol for the Navigation Operation, including pre- and postconditions

To summarize, the object protocol is able to model quite a range of navigation styles: from embedded links (web-browsers), over links that are stored in a separate link base ("generic links" in MicroCosm) up until structural computations ("Sprocs" in HBn/SBn). Therefore, we conclude that the object protocol depicted in Figure 1 indeed represents a generic navigation operation.

2.2. The Navigation Contract

The navigation operation depicted in Figure 1 specifies how the different objects in the system are supposed to interact. However, in extensible hypermedia systems, some of these objects may be provided by third parties. Thus, to ensure that the system functions properly, it is wise to protect against faulty components. Therefore, we extend the specification by including extra reliability checks.

An appropriate way of incorporating reliability checks is by means of the "Design by Contract" principle [11]. In short, this principle states that every operation on an object should assert its precondition (a statement of how the object expects the world to be before it executes the operation) and postcondition (a statement of how an object should leave the world after it has executed an operation). Pre- and postconditions are usually provided by means of predicates that check whether the corresponding statement is true for a given object, hence we include them as such in the specification of the navigation operation.

The extended specification of the navigation operation is depicted in Figure 2, where the pre- and postconditions appear against a grey background. As implied by the "Design by Contract" principle, its up to the protocol to specify what exactly constitutes the reliability checks, although participating objects may strengthen the contracts. In the general case, the preconditions for the `selectOn` operation verifies whether the marker and anchor objects may indeed launch a link traversal, while the postcondition verifies that we arrive in a valid location in a hyperdocument. The precondition for the `highlightOn` operation verifies whether the target marker represents a valid location within an existing document, while the postcondition verifies whether the target location is actually visible. A good example of what strengthening the contract implies can be found in the example of a web browser. There the precondition for the `selectOn` operation on an anchor verifies whether the source anchor contains a syntactically valid URL. Also, the postcondition for `highlightOn` on a marker verifies a typical feature of web-style navigation, namely that the source document is properly closed.

2.3. The Meta-Level Architecture

Now that we obtained an object protocol for hypermedia navigation including pre- and posconditions, we can derive the actual meta-level architecture. This is done by applying two generic design guidelines which appeared in [12], later recapitulated in [13]. The design guidelines start from a system designed according to the "Design by Contract" principle and derive a meta-level architecture by refactoring the pre- and postconditions and the object constructors into special purpose *meta-objects*. The resulting meta-objects plus the implied interaction protocol with the base-level objects constitute *the meta-level architecture*. As argued in section 3., the interaction protocol between the meta-objects and the base-level objects indeed allows to system to analyse its inner actions and adapt its internal representation accordingly.

The design guidelines state that a system designer should "turn contracts into objects" and "turn the configuration into a factory object". Applying these guidelines on the navigation protocol results in the meta-level architecture depicted in Figure 3, where the newly created meta-objects are set off against a grey background. The first guideline recommends to move all pre- and postconditions into a separate meta-object, named "`aNavigContract`" in the figure. The second guideline introduces one global meta-object (called "`globalFactory`") which is responsible for creating new objects. Thus, during a navigation operation it is the responsibility of (i) the *contract* object to verify the pre- and postconditions while (ii) the *factory* object must supply the appropriate contract, resolver and navigation targets.

To return to the example of a web browser, when the `sourceMarker` starts the navigation operation, it first requests the `globalFactory` to return the contract object that will supervise the navigation operation (`aNavigContract`). Next, that contract object verifies the precondition (i.e, whether the marker corresponds to an anchor) and then control is transferred to the `sourceMarker` object. Here as well the contract object verifies the precondition (i.e., whether the anchor contains a syntactically valid URL) after which the `resolver` function is invoked. This resolver function interprets the URL, but the creation of the objects representing the navigation targets is delegated to the globalFactory. After the `resolve` function returned, the navigation targets are highlighted, but the pre- and postconditions are again verified by the contract object. Finally, the contract object verifies the post condition for the `selectOn` oper-

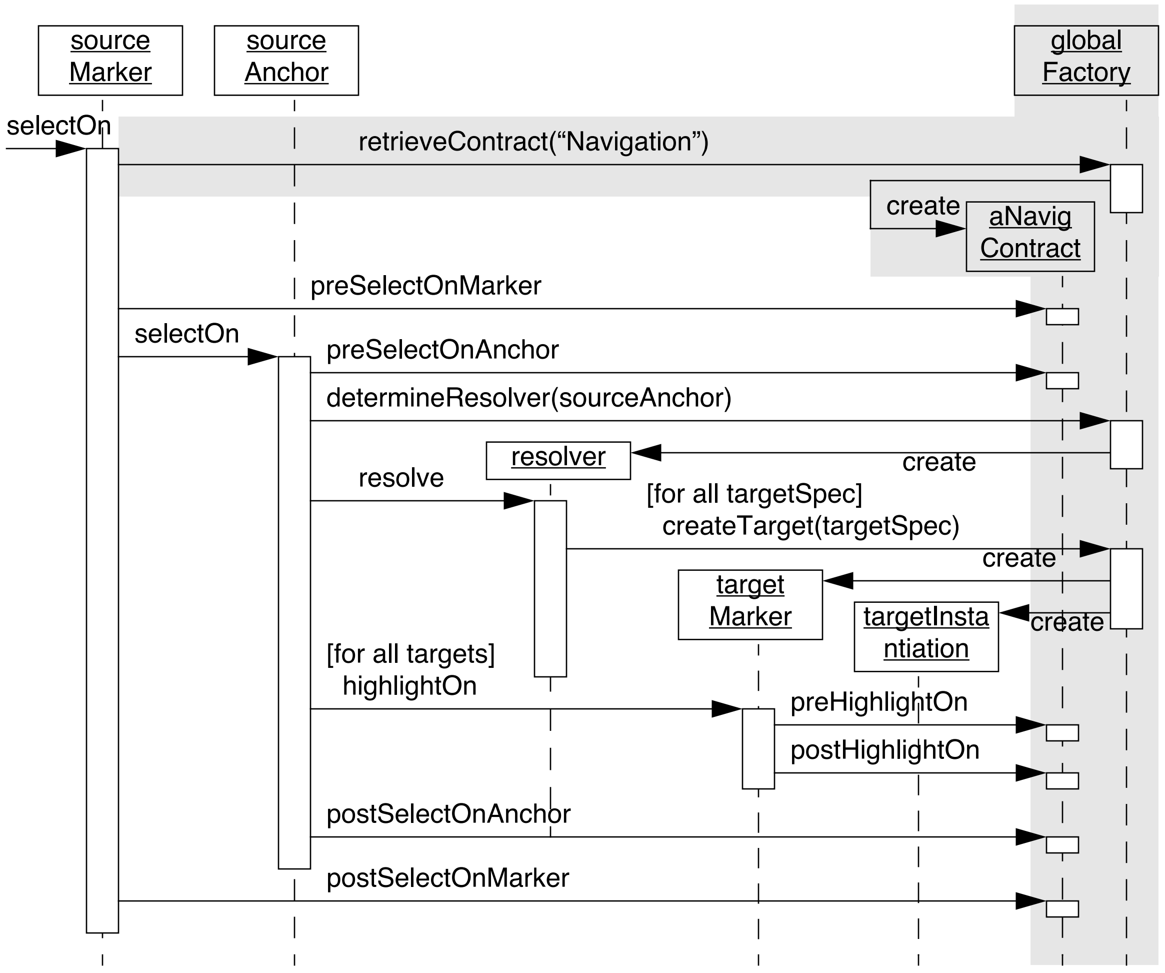


Figure 3 The navigation protocol with meta-level architecture (set off against a grey background.) One meta-object represents the navigation contract (`aNavigContract`), another meta-object represents the system configuration (`globalFactory`).

ation on both the `sourceAnchor` and the `sourceMarker` which terminates the navigation operation.

3. Extensibility via the Meta-level Architecture

Given the meta-level architecture depicted in Figure 3, we now explain how to exploit its presence to wrap additional behaviour around crucial operations, this way allowing a system to analyse its own behaviour and adapt it when necessary. This way, we argue that the design guidelines indeed provide a "good" meta-level interface, i.e. one that is open for future needs and establishes a clean separation of concerns.

3.1. Maintaining a Navigation Log

One of the recurring features in hypermedia systems is a "back" button, which in essence boils down to fetching the previously visited location from the log of navigation actions and navigating to that location. To work properly, this scheme requires that the all navigation operations are

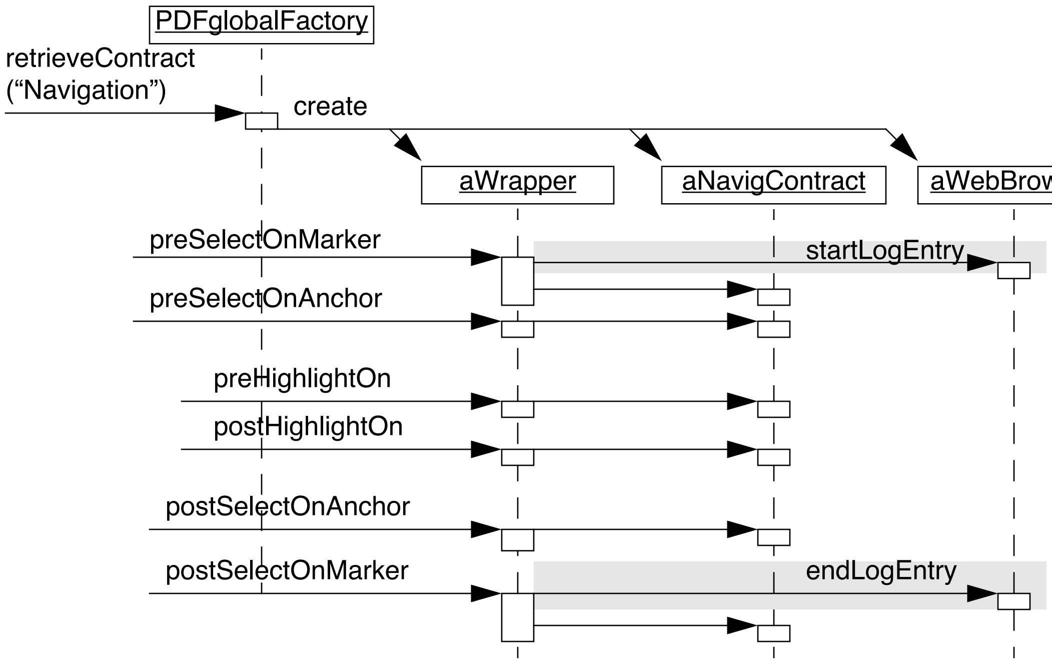


Figure 4 Maintaining the navigation log consistent by wrapping the meta-object. The globalFactory object is patched in such a way that it creates an extra wrapper object (`aWrapper`) which creates the log entries (`startLogEntry` and `endLogEntry`, set of against a grey background) and then forwards the pre- and postconditions to the original navigation contract.

logged consistently. With a monolithic hypermedia system this is feasible, since all the objects that participate in the navigation operation are known in advance. However, in extensible hypermedia systems —where document viewers may be provided by third parties and loaded at run-time— we do not have control over all objects, hence cannot guarantee the log's consistency.

It is during such "necessity of control" situations that the meta-level architecture comes to the rescue. Indeed, *all* markers and anchors —even when provided by third parties and loaded at run-time— must notify the navigation meta-object (aNavigContract) by means of the pre- and postconditions. If objects do not notify the meta-object, they deliberately choose to neglect the contract and such malicious intentions cannot be avoided. Of course, for many if not all of the third-party applications this involves extra patchwork, but this can be accomplished by means of scripting languages or wrappers [14], [6]. Consequently, the navigation meta-object monitors all navigation transition states independently of the base-level objects involved.

As a concrete example of how to ensure consistency via the meta-level architecture, let us return to the example of a web-browser extended with a PDF viewer introduced in section 1.. In this example, the link engine of the PDF viewer is separated from the one in the web-browser which sometimes results in inconsistencies. Avoiding these inconsistencies requires a PDF-viewer which adheres to the meta-level architecture in Figure 3 and a web-browser which has an API that allows to make entries in the navigation log. Like depicted in Figure 4, the person configuring the system must patch the `retrieveContract` operation for the `globalFactory` object inside the PDF-viewer. The patch returns a wrapper object which knows how to invoke the API of the actual web browser being used (we used `startLogEntry` and `end-`

`LogEntry` but this will of course depend on the web browser). After invoking the API, the wrapper object will forward control to the original navigation contract. This way, the PDF viewer acts as the base system which is extended in order to integrate properly with the link engine of the web-browser.

3.2. Quality of the Meta-level Architecture

Ensuring consistency is but one instance of a "necessity of control" situation. Especially in a distributed hypermedia system with multiple users having concurrent access to hypermedia documents there are more situations that require extra levels of control. For instance, we have applied the same guidelines on other object protocols in a hypermedia system to achieve concurrency and access control (see [15] for further details).

Consequently, we claim that with respect to the criteria in the Introduction, the guidelines actually derive a "good" meta-level architecture. First of all, we point out that it is the explicit representation of the contracts which provides the necessary hooks for extensions. Since a contract forces the designer of an object protocol to make the important state transitions explicit, it provides an ideal place to monitor these state transitions. Therefore, the "Design by Contract" basis implies that the meta-level controls the important operations, thus most likely *those places where extensions are necessary*. Secondly, since the contract objects only allows to verify pre- and postconditions, one can use these hooks only for wrapping additional behaviour and never for direct intervention into the base-level operations. Thus the design guidelines always result in a *clean separation* between the base-level and the meta-level.

4. Discussion

4.1. Experimental Validation

The meta-level architecture described in this paper has been experimentally validated in the Zypher hypermedia system as part of a PhD effort combining state-of-the art object-oriented software engineering techniques with open hypermedia technology [15]. The resulting artefact used the world-wide web to seamlessly navigate between source-code and its design documentation [10].

Part of the PhD work has been summarised as a set of design guidelines that derive a tailorable framework from an open design space [12]. Two of these design guidelines have later been rephrased and refined in the context of distributed systems [13]. The same two design guidelines are put to use in this paper to derive the meta-level architecture for hypermedia navigation.

4.2. Potential Drawbacks

While a meta-level architecture permits to control system extensions, it should be clear that this comes at a cost.

- *Extra complexity*. As can be observed in the difference between Figure 1 and Figure 3, a meta-level architecture implies a few additional objects and a considerably larger object protocol. Also, once we actually start to exploit the meta-level architecture, the number of wrapper objects quickly explodes. This is without a doubt the most important drawback of a meta-level architecture.

- *Late binding technology*. The meta-level architecture in itself does not provide run-time extensibility, it only provides an extra level of control on how third-parties may extend the base system. To actually achieve run-time extensibility one must use other forms of late binding technology, either a language with built-in features (Smalltalk and Java) or otherwise some form of embedded scripting language (like in [3], [4]).
- *Performance penalty*. The meta-level involves a lot of extra message-passing between the base-level and the meta-level. This will most likely impose some performance penalties.

4.3. Potential Benefits

Even though a meta-level architecture is quite costly, it has some unique advantages that makes it worthwhile for many hypermedia systems.

- *Very flexible*. The main advantage of a meta-level architecture is that it permits a lot of powerful extensions to the base system without actually changing it. For hypermedia systems applied in many different contexts (like most of the hyperbases [16]) this is a very desirable feature as it permits to deploy a stable core which is extended as needed in the particular context.
- *Complementary to other extension techniques*. A meta-level architecture should not be used on itself, but rather be combined with other extension techniques. Ideally, the meta-level interface is accessible via an API and third-part applications are adapted via scripting and wrapping (see [14], [6]) to properly invoke that API.
- *Poor men's reflection*. Languages such as CLOS or Smalltalk provide built-in reflection mechanisms, which makes it easy to monitor and control any slice of an object protocol ([17], [18]). Our design guidelines result in a kind of "poor men's reflection", where the meta-objects provide the means for a limited form of method-instrumentation applicable in mainstream object-oriented languages such as C++ and Java.

5. Conclusion

In this paper, we have derived a meta-level architecture for a hypermedia link engine. Next, we have shown how such a meta-level architecture makes is possible to control the way third-party applications interact with the link engine, as such making it possible to ensure a consistent internal state. Finally, we have argued that while a meta-level architecture provides the necessary hooks for ensuring consistency, it also adds considerable complexity thus should only be applied when the situation calls for it. However, with the growing demand for hypermedia functionality, it is possible that these meta-level facilities may one day be provided by the underlying operating system, precisely because these also require an extra level of control.

Acknowledgements. I want to thank my colleagues Sander Tichelaar and Franz Achermann who have proofread earlier versions of this document: their suggestions improved the paper considerably. The same applies for the anonymous reviewers which helped to clarify the contents of this paper. Last but not least, I want to thank all participants in the series of "Open

Hypermedia Systems" workshops (http://www.ohswg.org/) for the many fruitful discussions we had over the years.

6. References

[1] Robert Laddaga & James Veitch 1997. Dynamic Object Technology. Communications of the ACM, 40(5), 36-38.

[2] Frank Buschmann & Regine Meunier & Hans Rohnert & Peter Sommerlad & Michael Stad 1996. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons.

[3] Uffe Kock Wiil & John J. Leggett 1992. Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems. In *Proceedings of the ACM Conference on Hypertext (ECHT'92),* Milano - Italy, November 1992. ACM Press, 251-261.

[4] Kai Grønbæk & Jawahar Malhotra 1994. Building Tailorable Hypermedia Systems: the embedded-interpreter approach. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications,* Portland-Oregon, October 1994, 85 - 101.

[5] Frank Halasz & Mayer Schwartz 1994. The Dexter Hypertext Reference Model. Communications of the ACM, 37(2), 30 - 39.

[6] [Davi94a] Hugh C. Davis & Simon Knight & Wendy Hall 1994. Light Hypermedia Link Services: A Study of Third Party Application Integration. In *Proceedings of the European Conference on Hypertext (ECHT'94)*, Edinburgh - UK, September 1994, ACM Press, 41-50.

[7] Paul H. Lewis & Hugh C. Davis & Steve R. Griffiths & Wendy Hall & Rob J. Wilkins 1996. Media-based Navigation with Generic Links. In *Proceedings of the ACM Conference on Hypertext (HT'96)*, Washington - USA, March 1996, ACM Press, 215-223.

[8] Peter J. Nurnberg & John J. Leggett & Erich R. Schneider 1997. As we should have tought. In *Proceedings of the ACM Conference on Hypertext (HT'97)*, Southampton - UK, April 1997, ACM Press, 96-101.

[9] Serge Demeyer 1999. Structural Computing: The Case for Reengineering Tools. In *Proceedings of the 1st Workshop on Structural Computing - Hypertext'99*, Darmstadt - Germany, February 1999. At: http://win-www.uia.ac.be/u/sdemey/Pubs/.

[10] Serge Demeyer & Koen De Hondt & Patrick Steyaert 2000. Consistent Framework Documentation with Computed Links and Framework Contracts. ACM Computing Surveys, 32(1).

[11] Bertrand Meyer 1997. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[12] Serge Demeyer & Theo Dirk Meijler & Oscar Nierstrasz & Patrick Steyaert 1997. Design Guidelines for Tailorable Frameworks. Communications of the ACM, 40(10), 60-64.

[13] [Tich00a]Sander Tichelaar & Juan-Carlos Cruz & Serge Demeyer 2000. Design Guidelines for Coordination Components. In *Proceedings of the ACM Symposium on Applied Computing 2000 - Track on Coordination*, Como - Italy, March 2000. ACM Press, Vol I, 270 - 277.

[14] E. James Whitehead, Jr. 1997. An Architectural Model for Application Integration in Open Hypermedia Environments. In *Proceedings of the ACM Conference on Hypertext (HT'97)*, Southampton - UK, April 1997. ACM Press, 1-12.

[15] Serge Demeyer 1996. *ZYPHER Tailorability as a link from Object-Oriented Software Engineering to Open Hypermedia*. Ph.D. Dissertation, Vrije Universiteit Brussel, Belgium, Departement of Computer Science. At: http://win-www.uia.ac.be/u/sdemey/Pubs/

[16] Uffe Kock Wiil & Peter J. Nürnberg & John J. Leggett 1999. Hypermedia Research Directions: An Infrastructure Perspective. ACM Computing Surveys, 31(4es).

[17] Gregor Kiczales & Jim Des Rivieres & Daniel Bobrow 1991. *The Art of the Metaobject Protocol*, MIT Press.

[18] St phane Ducasse 1999. Evaluating Message Passing Control Techniques in Smalltalk. Journal of Object-Oriented Programming, 12(6), 39-50.

Modelling Architectural Variability for Software Product Lines

Thomas Weiler
Research Group Software Construction, RWTH Aachen, Germany
thomas.weiler@cs.rwth-aachen.de

Abstract

In this paper requirements for a concept to model software product line architectures are presented. Furthermore a process for SPL architecture modelling is described which incorporates the concept of the model driven architecture (MDA) into SPL architecture modelling. Besides a metamodel for SPL architecture modelling elements is shown, which – combined with the process for SPL architecture modelling - fulfils the requirements deployed in the first part.

Modelling variability and traceability of requirements within a software architecture thereby possesses the main focus. Therefore a detailed breakdown of different kinds of variability found in product line based software architectures is given. The presentation concludes with an small excerpt from a case-study within the context of an e-shop, which should clarify the application of the elements of the metamodel presented before.

1. Introduction

Software Product Lines (SPLs) are an advancement in software reuse. In the scope of SPLs reuse however refers to all documents that evolve during the development of (similar) products. Examples for these documents are requirements, architecture models or database designs.

SPL development is divided into two main parts, which execute interactively. Within the *domain engineering* the common and variable parts of products, which belong to an *application domain,* are analysed and described. The resulting documents of this process form the basis of the product line, the so-called *Product Line Platform (PLP)*. During the *application engineering* concrete *products* are then derived from this PLP. Thereby the terms *application* and *product* will be used synonymous below.

By maximising the reuse of documents in the product line-based software development, time-to-market as well as development costs can be significantly reduced [1]. Furthermore a correct applied product line-based approach encourages the quality of the end products by careful development and intensive tests of the common parts of the SPL.

2. Present approaches

Most approaches in the scope of SPLs are focusing on the requirements engineering. They primarily consider the delimitation of the application domain during the process of *scoping* as well as the acquisition and modelling of requirements for SPLs.

Thereby it is identified to be crucial, to explicitly model the variability of requirements for products of a SPL. Furthermore a dedicated mechanism is needed, which allows the product developer to resolve the modelled variability for a concrete product in a way desired by the developer of the PLP.

Within all these approaches it is often neglected that product line-based software development can only lead to full success if it is recognized as an integrated concept, which involves all phases of the software engineering process. In the following this article concentrates on architecture modelling for SPLs.

3. SPL architecture modelling

Architecture modelling for SPLs partially demands similar requirements as architecture modelling for *conventional* systems. But many of these requirements need a more intensive attention in the scope of SPLs, because the PLP architecture often forms the basis for a huge set of derived product architectures. This simultaneously is the risk and the chance of SPLs.

In the following requirements for a SPL architecture modelling concept are presented which are determined during the case study presented in section 8 and are additionally the result of a comparison of existing approaches in the context of SPLs, see also section 9. Thereafter a SPL architecture modelling process and a metamodel for SPL architecture modelling elements will be presented which fulfil the specified requirements.

Entities and relations: First of all – as with every other architecture modelling language – there must be a possibility to model the central building blocks of a system – the entities – and their connections, the relations. Thereby the entities describe central units of the system to be modelled and the relations describe structural and behavioural connections of this units like e.g. hierarchical or uses relations.

Separation of concern: Architecture modelling for SPLs must provide the possibility to concentrate on specific aspects of a system [10]. This concept known as *separation of concern* is divided into two dimensions: Along the *horizontal dimension* it is possible to designate the focus on a part of interest (*clipping*). The *vertical dimension* allows to magnify a given fixed cutout step by step in order to get a more and more exact image of the cutout in question.

A combination of both dimensions is the so-called *zooming*, in which an aspect is magnified step by step whereby the observed cutout is simultaneously scaled down and vice versa. This may be seen analogous to a photographic lens with zoom-function where a longer focal length (higher magnification) results in a smaller angle.

Traceability: Traceability of requirements down to the architecture and finally to the source code (and back) is a vital task to ensure the comprehensibility and maintainability of a software system. In the scope of SPLs the claim for traceability is so much important because resolving the variability of the requirements has direct impact on the design and therefore the source code of the SPL. Only if the traceability of requirements down to the design and furthermore the source code is guaranteed, one can fully benefit from the possibilities of reuse and therefore of cost-saving.

Evolution: Similar to conventional software products a SPL isn't resistant against changes during its life cycle. By and by changing requirements lead to changed architectures and products. Therefore a mechanism is needed to track these changes over time. In the context of SPLs this not only means versioning but also to decide when and how to migrate already derived products when changing the PLP.

Technical platform independence: To maximise the benefit of reusing components, the design of a system and components respectively should be independent of the implementation technique used as long as possible along the levels of abstraction. Thereby the term *component* is not meant to denote a component known from e.g. CORBA or EJB but a higher building block used in architecture modelling. This will be discussed in more detail in section 6.

The request for technical platform independence complies with the *Model Driven Architecture (MDA)* approach conceived by the OMG [4]. In the scope of architecture modelling for SPLs, this technical platform independence refers to the development of the PLP architecture as well as the architectures of therefrom-derived products.

It should be mentioned that the term *platform* is used in the scope of SPL engineering as well as in the *MDA* approach. So one should not mix up the two meanings of the term *platform*. While in the context of SPLs this term describes all documents on which the product line is based, in the context of the MDA it refers to the *technical platform* used. So if not explicitly mentioned context should clarify which meaning was meant by. The relationship between SPLs and the MDA will be discussed in more detail in sections 4 and 5.

Variability: Modelling different variability within a SPL is vitally important for the requirements engineering as well as for designing the architecture. Combined with the *traceability* arises the possibility to resolve variability at the level of requirements during product configuration and to implement it through the design level down to the implementation level, see also section 4. Therefore a concept for SPL architecture modelling needs to provide the possibility to distinguish between common and variable parts of the products derived from a PLP.

Decision support: In order to resolve variability offered in the PLP architecture in a way intended by the platform developer a mechanism is needed, which helps the product developer to make the needed decisions. Therefore each variability modelled in the PLP architecture must be furnished with an annotation – normally formulated in natural language – which provides the product developer with the needed information to resolve given variability.

Dependencies: By modelling the variability within a SPL it must be taken into account, that there might be dependencies between components of the system. This can mean that for example the existence of one component requires the existence of another component. Therefore a concept for SPL architecture modelling needs to support an appropriate type of relationship.

Having described the requirements for SPL architecture modelling in the next section a process will be presented, which illustrates the necessary steps and the dependencies by modelling SPL architectures.

4. SPL architecture modelling process

This section presents a process for SPL architecture modelling. As already mentioned in section 1 SPL architecture modelling is organized in the two areas *domain engineering* and *application engineering*. In Figure 1 the part of architecture modelling gets more improved.

Within the *domain engineering* initially the requirements for the entire PLP are collected together with the identified variability and afterwards compiled into a *requirements model* for the PLP, which among other things contains e.g. a *feature graph* [2]. This requirements model forms the basis for the top-level layer of the PLP architecture. Starting from this still abstract architecture layer the PLP architecture gets more and more improved in further architecture layers. This procedure is according to the *Model Driven Architecture (MDA)* approach introduced by the OMG [4], see also section 5.

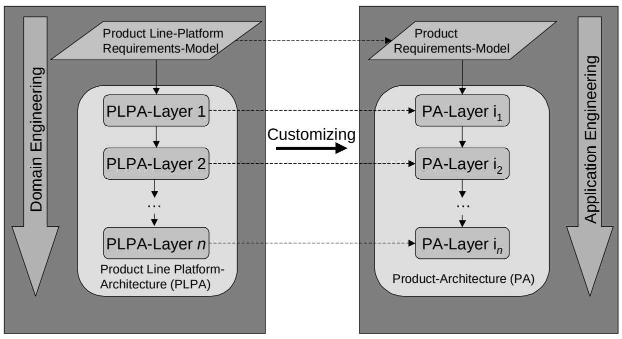

Domain Engineering
Product Line-Platform Requirements-Model
PLPA-Layer 1
PLPA-Layer 2
PLPA-Layer *n*
Product Line Platform-Architecture (PLPA)
Customizing
Product Requirements-Model
PA-Layer i_1
PA-Layer i_2
PA-Layer i_n
Product-Architecture (PA)
Application Engineering

Figure 1. SPL architecture modelling process

In the last step within the domain engineering the that way specified generic architecture gets realized as far as possible. Thereby – according to the differentiation in common and variable components – both finished and incomplete components are placed in the PLP, see also section 6.

At the beginning of the *application engineering* firstly the requirements for a concrete product are determined on base of the requirements for the PLP. Afterwards – similar to the domain engineering – a first coarse architecture layer for the product is developed, which is based on the layer of the same abstraction level as in the PLP architecture. In the following this top-level architecture becomes more and more improved analogue to the layers of the PLP architecture.

Thereby the variability included in the PLP architecture is resolved conform to the previously identified product requirements. In the last step the executable system is implemented based on this product architecture.

5. MDA and SPL architectures

To fulfil the requirement of technical platform independence - see section 3 - the *Model Driven Architecture (MDA)* approach of the OMG [4] can be incorporated into a model for SPL architecture modelling. Figure 2 shows an approach to integrate the MDA in a concept for modelling SPLs.

Thereby the *core model* known from the MDA is specialized to a *domain specific core model*, which offers modelling elements adapted on a given domain. These modelling elements are used to define a *platform independent PLP model* conforming to the MDA, based on the analysed requirements for the PLP. The platform independent PLP model consists of several *abstraction layers*, which give from top to bottom a more and more complete view of the modelled system. It is then - according to the MDA - mapped to a *platform specific PLP model*, which also consists of several abstraction layers.

During the *application engineering* initially the product requirements are determined based on the requirements of the PLP and then implemented by a *platform independent product model* pursuant to the

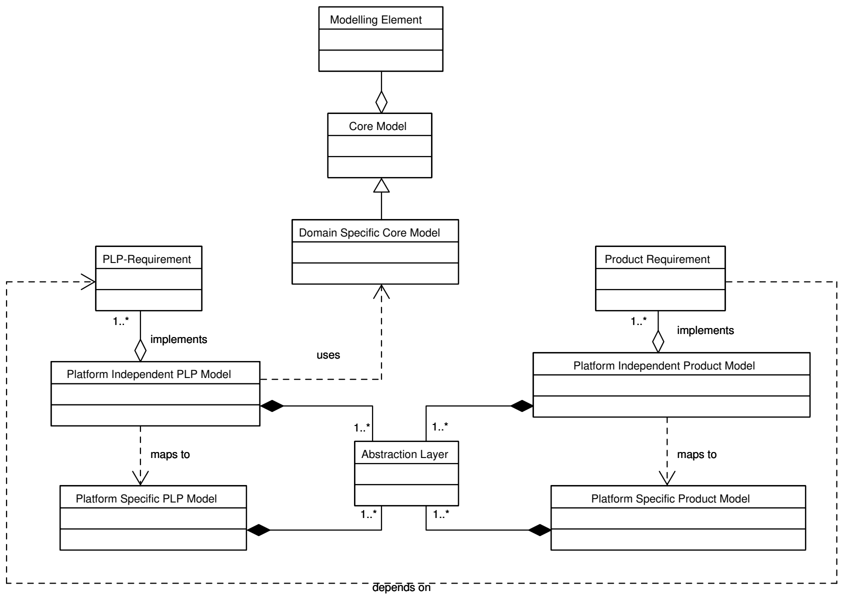

Modelling Element
Core Model
Domain Specific Core Model
PLP-Requirement
Product Requirement
1..*
implements
uses
Platform Independent PLP Model
Platform Independent Product Model
1..*
Abstraction Layer
maps to
Platform Specific PLP Model
Platform Specific Product Model
depends on

Figure 2. MDA and SPLs

MDA. This consists – analogue to the platform independent PLP model – of several abstraction layers and is mapped to a *platform specific product model*, which in turn consists of several abstraction layers.

6. Feature components

The central building blocks for modelling the PLP and application architectures in the approach presented here are *feature components*. A feature component can be seen as a self-contained unit, which represents a specific characteristic of the system to be modelled. They are an adaptation of the *feature* concept introduced by the *Feature Oriented Domain Analysis (FODA)* to the level of architecture modelling for SPLs [1].

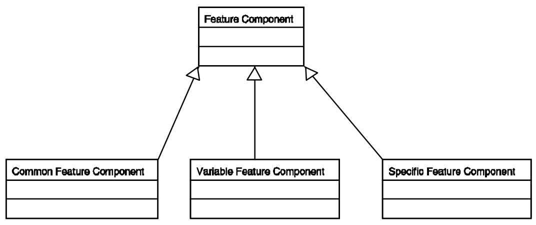


Figure 3. Feature Components

It must be mentioned that the feature components at the level of architecture modelling aren't necessarily identical to the features according to FODA, which are identified at the level of the requirements analysis [2]. For example it might be possible that a set of features identified in the requirements analysis together build a feature component at the level of architecture modelling. It might also be possible, that a feature is implemented by a set of feature components likewise *aspects* in the *Aspect Oriented Development* [5]. Furthermore feature components need – contrary to their name – not to be realised at the implementation level as components provided by for example CORBA or EJB. As shown in Figure 3 feature components can be divided into three different types.

Common feature components are used in a PLP architecture and describe feature components, which can occur in every application based on this architecture. Common feature components occur in derived application architectures without modification.

Variable feature components are feature components, which can occur in every derived application architecture only by resolving the offered variability of type *incomplete specification*. This type will be described in more detail in section 7.1.

The last type of feature components is represented by *specific feature components*. They are special building blocks needed to construct a specific application architecture derived from a PLP architecture. At this it must be taken into account, that in the course of the evolution of a SPL an initially product-specific feature component at a later date can be incorporated into the PLP and thereby become a variable or even a common feature component of the PLP, see section 3.

7. Metamodel

After this preparatory work in this section a metamodel for SPL architecture modelling elements will be given which – in conjunction with the SPL architecture modelling process presented in sections 4 and 5 – fulfils the requirements described at the beginning. In section 8 an example will illustrate the elements presented in the metamodel shown in Figure 4.

The central modelling element is the *feature component* mentioned in section 6. Thereby each feature component memorises the requirements covered by it. In doing so *traceability* of requirements down to the architecture level is supported as asked for in section 3.

Feature components can participate in *relations* with the aid of *relation ends* as known from the UML [3]. Thereby a relation can be a *dependency* – see also section 3 – or a *hierarchy* relation.

Among a *dependency*-relation two different kinds of dependencies between feature components can be distinguished:

- Prohibited
- Required

A dependency of type *prohibited* is an undirected relationship between two feature components. In a *prohibited*-Relationship the existence of one feature component forbids the existence of the other feature component in a derived product architecture.

A dependency of type *required* is a directed relationship between two feature components. It is used if the existence of one feature component of the PLP architecture depends on the existence of another feature component of the PLP architecture within a derived product architecture.

A *hierarchy*-relation depicts a conceptual structure between a super- and a – possibly set of – sub-feature component(s). It should be seen more as a *is part of*-relation than a generalisation similar to the connections used in a *feature graph* in FODA [2].

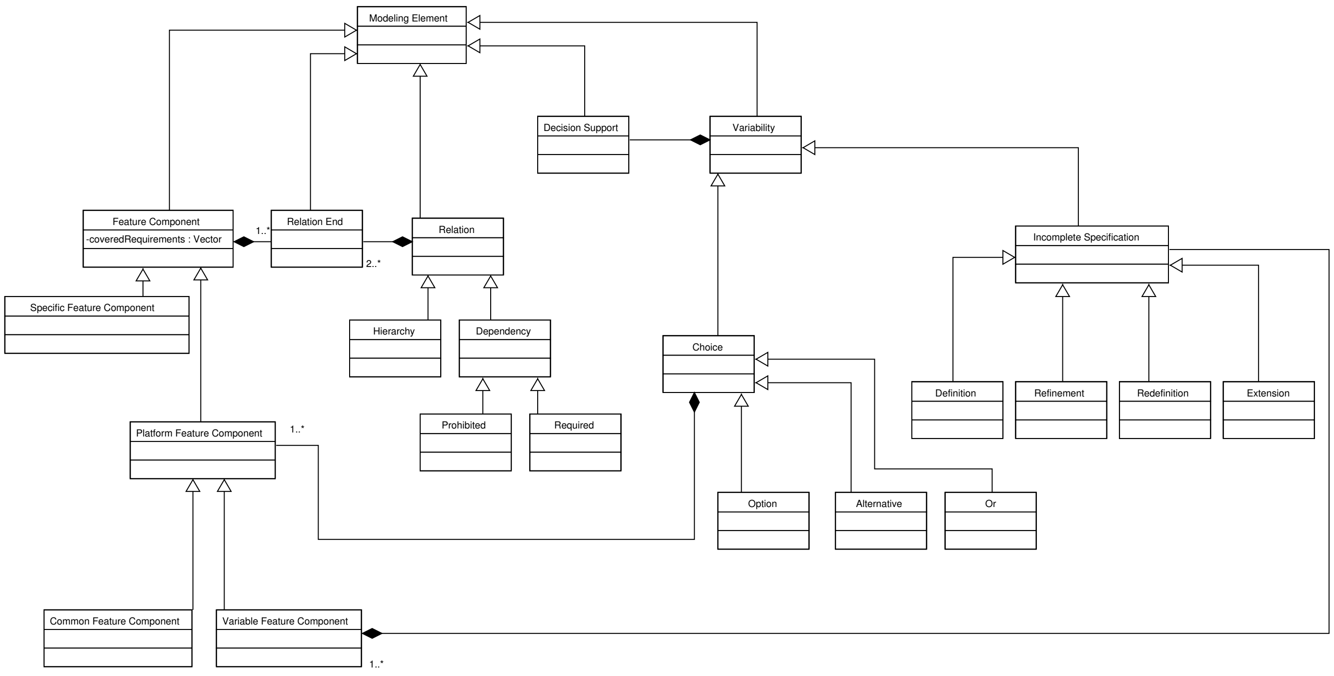


Figure 4. Metamodel for SPL architecture modelling elements

The other major part of the metamodel pertains to the modelling of *variability*. Thereby two types of variability can be distinguished: *incomplete specification* and *choice*.

7.1. Incomplete specification

Variability in the form of an *incomplete specification* is characterised by a missing or incomplete specification of a component. At this four different types can be distinguished:

A *definition* only determines the skeleton of a feature component likewise an interface. The detailed specification is done during the *application engineering*.

A *refinement* defines the behaviour and data of a feature component in an abstract way likewise a template- or hook-feature component. The exact design will be defined product-specific.

At the *redefinition* a specification for the feature component exists already but it can be renewed product-specific. This can serve for the definition of a preset specification of a feature component, which can be product-specific redesigned.

Similar to the *redefinition* the *extension* also defines a (standard) specification of a feature component. However this specification can be product-specific extended by functions or data.

Beyond these four types of incomplete specification *redefinition* and *extension* are *optional* variability because in these cases a sufficient complete specification of the feature component in question is given. On the other hand variability of type *definition* or *refinement* must always be resolved.

7.2. Choice

The second type of variability between members of a SPL concerns the *choice* from a set of offered feature components from the PLP. It can be distinguished in the following three types:

- Option
- Alternative
- Or

In case of an *option* the product developer has to decide, if he takes over an optional feature component from the PLP to the product architecture. In case of an *alternative* exactly one feature component must be chosen from a set of offered feature components.

An *or*-choice describes a set of feature components from which one ore more feature components must be chosen. Table 1 shows the different types by illustrating the used cardinalities of the choice and selection sets. It should be mentioned that these three types could also be combined to obtain a broader variety of possible sets to choose from.

Table 1. Choice

	Cardinality of choice	Cardinality of selected set
Option	0..1	1
Alternative	1	*
Or	1..*	*

When resolving variability during the *application engineering,* incomplete specifications must be completed that means defined, refined, redefined or extended. Furthermore the product developer has to come to a decision about the feature components to choose from sets of offered feature components in variability of type choice.

Regarding all types of variability a *decision support* is provided which supports the product developer resolving given variability, see section 3.

8. Example

In the following a small excerpt from a first case-study is presented to illustrate the application of the metamodel elements. This case study models a SPL in the context of an Internet e-shop.

In Figure 5 a feature graph modelling the *order* subsystem of an e-shop product line is shown. Thereby an extended notation compared to FODA is used [2].

The order system consists of an optional feature *payment* denoted by the circle above the feature element. The feature graph defines different types of payment methods among which the product developer can chose one or more. Within this *or*-choice – see section 7.2 – the feature *other payment method* is a placeholder for further payment methods which can be defined product specific.

On the right hand of the feature graph a feature *order confirmation*, which denotes the kind of order confirmation for the seller, is described, where the product developer must decide, which one of the alternatives offered he chooses, see also section 7.2. Amongst the three offered alternatives the feature *fax* needs to be redefined in a derived application, see section 7.1.

The two remaining optional features are the possibility to distinguish a *delivery address* from a billing address and to make use of a *gift service*. Thereby the *gift service* depends on the feature *delivery address* because one rarely wants to send one's gift together with an invoice. This is shown by the use of a *requires* relationship between this two features.

In the feature graph shown every variability is numbered, whereby the numbering scheme should be read from top to bottom. For example the variability of type *definition* at the feature *other payment method* has number 1.1b.1 because it is under the or-choice number 1.1, which in turn is under the optional feature *payment*, which has number 1.

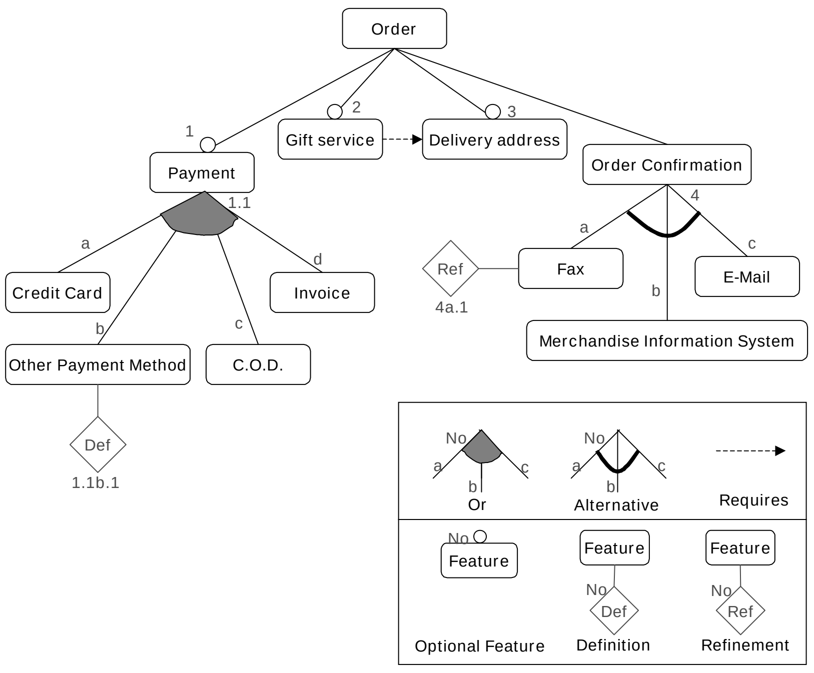


Figure 5. Feature graph e-shop

By using this numbering scheme the product developer can move along a *decision tree* build up from this hierarchical variability numbers. Together with a *decision support* for every variability modelled, that way the product developer can easily resolve the variability offered by the PLP.
After this description of an feature graph for the *order* part of the e-shop the associated PLP architecture will be presented in part. It is constructed as a three-layer architecture.

The PLP architecture is made up of a *presentation layer*, which visualises the outcomes of the subjacent *business logic layer* and serves in addition as the communication interface from the end user to the e-shop system, normally by means of a web browser.

The *business logic layer* contains the functional components of the e-shop, e.g. order handling or customer management. In the following this layer will be described in more detail.

The lower most layer is the *database layer*, which provides the business logic layer with the functionality needed to manage the dates with the help of a database system.

It should be mentioned that the layers described here aren't identical to the PLP architecture layers mentioned in sections 4 and 5. Here the three layers describe a logical segmentation of the system to be modelled (a tier-architecture) whereas in the second case the layers describe the hierarchy of abstraction of the modelled PLP architecture.

The variability described in the feature graph in Figure 5 is brought down to the PLP architecture of the e-shop. Figure 6 presents a part of the business logic layer, which amongst other things consists of the feature components *order_system, data_access_support, customer_management, application_control,* and *catalog_management.*

It is visible that the feature component *order_system* is influenced by two types of variability presented in the feature graph in Figure 5. Furthermore the feature component *catalog_management* has a variability annotated, which was modelled in another here not shown part of the feature graph.

The feature component *data_access_support* in the above figure shall depict a feature component, which has no direct conjunction with features from the feature graph but is a feature component needed for technical realisation. It should be mentioned that it is possible, that certain variability arises not until architecture level. Thus it is imaginable, that a feature component can be realised in many different ways – for example a DBMS can be realised relational or object oriented.

The two other feature components in Figure 6 will not deepened and are only shown for reasons of completeness. In the following the feature component *order_system* will be observed in more detail.

Figure 7 shows a detailed view of the feature component *order_system* mentioned before. Here the abstraction level allows using a well-known modelling language – here the UML – in order to describe the specific characteristics of this feature component. As can be seen in Figure 7 the different types of variability modelled in conjunction with the features *payment* and *order confirmation* in the feature graph of Figure 5 can be regained in the feature component *order_system*.

The optional feature *payment* is mapped to the now optional class *PaymentMethod* depicted by the circle with annotation *Opt* and number 1. Similar the alternative number 4 and the or-choice number 1.1 are represented in this feature component. Three additional classes are shown, which describe an order based on a (virtual) shopping cart. These two classes come from another feature not modelled in the feature graph shown in Figure 5.

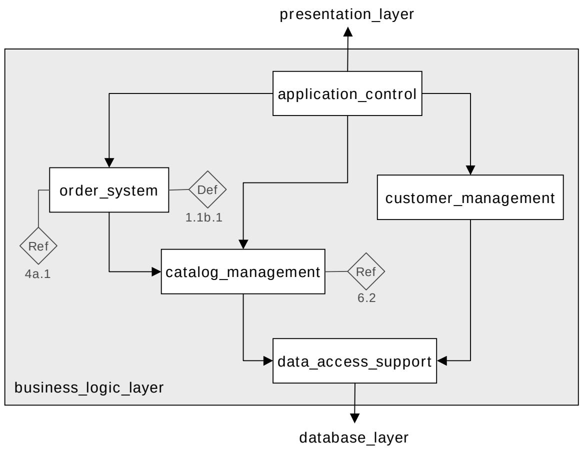


Figure 6. Business logic layer

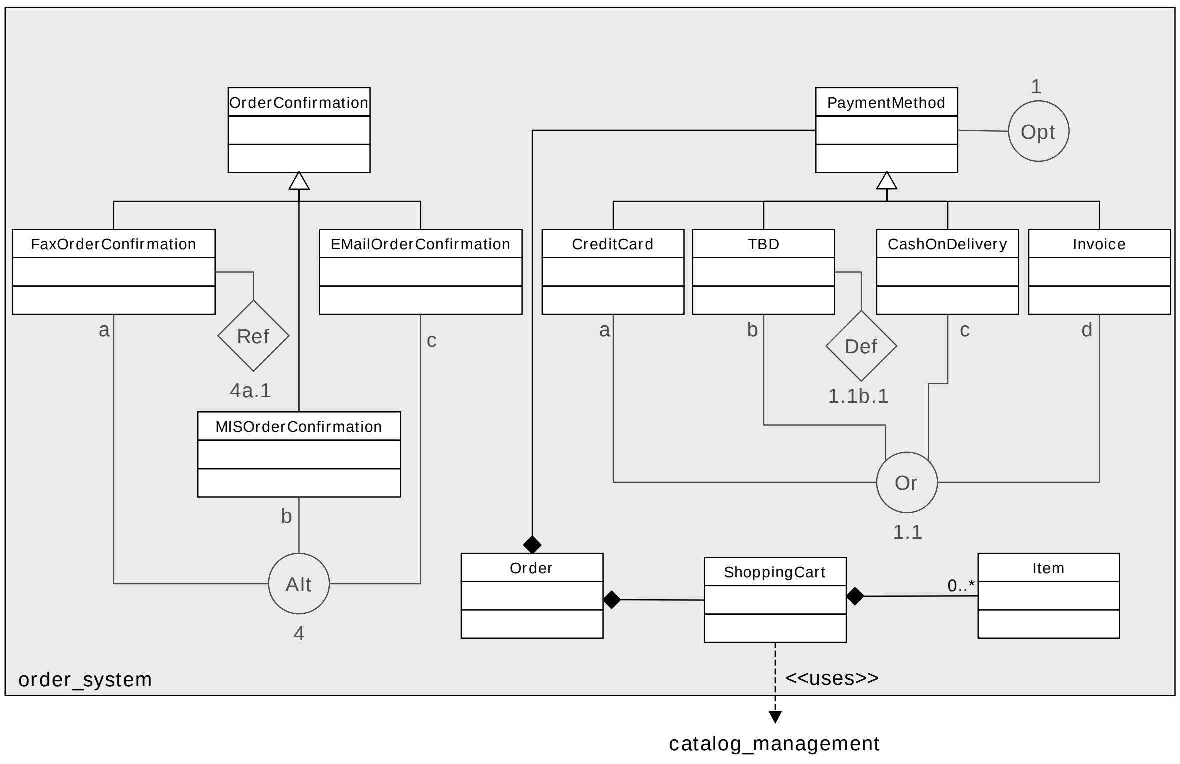


Figure 7. Order system

It should be pointed out that the types of variability shown in the feature graph not only have impact on the *business logic layer* and therefore the feature component *order system* but also on the other layers *presentation layer* and *database layer* and their corresponding feature components. For example the or-choice number 1.1 between the different types of payment methods must also be modelled (and implemented) at the *presentation layer*, so that e.g. the end user can choose his preferred payment method. As can be seen in this example, the mapping of features from the feature graph doesn't need to match one-to-one with the feature components modelled at the architecture level, as already mentioned in section 6.

The next step is to bring the modelled variability down to the source code. This can be achieved by annotating the source code with appropriate tags to depict the different types of variability. Because this actual is work in progress it will not deepened here.

9. Related work

As stated in section 2 most of the existing approaches concerning SPLs are focusing on the requirements engineering. Nevertheless some approaches exist which try to concentrate more on the downstream phases of the development process like the design, whereby some of them had certain influence on the approach presented in this article. As also stated by Muthig et.al. in [8] existing approaches often seem to be more pragmatic solutions resulting from practical modelling experiences in a particular domain or environment whose results are not universally transferable.

In [6] Flege describes an approach for using the UML [3] for system family architecture description. Thereby he focuses solely on construction-time variability, because only this type of variability results in different products and is therefore essential for developing SPLs. Presence of variability at later stages like e.g. at binding or runtime doesn't require special attention in the context of SPLs because they only affect one single product, see also [8] and [9].

The drawback of Flege's approach is the lack of elements in the UML for explicit modelling of architectural variability. Flege uses UML's stereotypes to depict variable architectural elements. Thereby he only models optional elements by neglecting e.g. alternatives among modelling elements. In Flege's approach alternatives should be modelled at the level of the decision model. At the design level this leads to optional elements (the single alternatives) which are no more distinguishable from other, real optional elements. Therefore the approach presented in this article explicitly distinguishes the different types of variability presented in section 7 at the design level to allow traceability from the requirements down to the design and the source code.

Furthermore Flege focuses exclusively on variability with a complete set of specified variants by discarding variability of type incomplete specification that might be used by product developers in an unanticipated way. As per Flege the reason for this is that unspecified variability has no impact during the instantiation of a reference architecture. In the approach presented in this paper variability of type incomplete specification is explicitly included. At first different specifications of elements among products of a SPL – resulting in incomplete specification in the PLP architecture – are a distinguishable characteristic of these products and therefore represent one type of variability within a SPL. Furthermore only by explicitly modelling variability of type incomplete specification – including the corresponding decision support – one can help the product developers to use the offered variability only the way intended by the PLP developers.

In [7] Batory et.al. refer to the need for higher-level modelling elements when modelling SPL architectures. Therefore they use features at the design level instead of e.g. modules. These features are then step-wise refined during the design resulting in a more and more precise architecture description. In their approach Batory et.al. concentrate more on the transition from the design to the implementation by introducing templates for JAVA. The *feature components* presented in section 6 also try to offer higher-level architecture modelling elements but are – contrary to Batory et. al. – clearly differentiated from the features of FODA [2] used during the requirements analysis.

10. Conclusion and future work

In this paper requirements for a concept to model SPL architectures were presented. Furthermore a SPL architecture modelling process was described which incorporates the concept of the *model driven architecture* into SPL architecture modelling. Besides a metamodel for SPL architecture modelling elements was shown, which – together with the described SPL architecture modelling process - fulfils the requirements deployed in the first part.

A first practical application in the context of a case-study from which parts were shown in the example illustrated in section 8 has shown the load capacity of the presented concepts for a medium sized application. Within this case-study a domain for e-shops was analysed and based on a requirements model including a feature graph for this domain a PLP architecture using the modelling elements offered by the presented metamodel was developed.

For the time being two products were derived from this PLP to show the load capacity of the given concept. Thereby it turned out that – although the concept was useful – a meaningful and broader application can only be achieved if the concepts are supported by tools. Otherwise the PLP and product developers can hardly manage the given complexity.

This leads to another aspect, which requires more work to be done: The transitions from requirements engineering to architecture design and from architecture design to the level of implementation must be supported in a concept for modelling SPL architectures. Otherwise the lack of systematics makes the stability and durability of a SPL solely depending on the intelligence and creativity of the developers involved.

11. References

[1] Donohoe P. (editor), *Software Product Lines: Experience and Research Directions*, Kluwer International Series, 2000.

[2] Kang, et. al., *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report SEI-CMU, Pittsburgh, 2000.

[3] OMG, *Unified Modeling Language Specification, Version. 1.4*, Technical Report, OMG, 2001.

[4] Soley R., OMG, *Model Driven Architecture*, White Paper, OMG, 2000.

[5] AOSD Steering Committee, *Aspect-Oriented Software Development*, http://aosd.net

[6] Flege O., *System Family Architecture Description Using the UML*, IESE-Report No. 092.00/E, 2000

[7] Batory, Johnson, MacDonald, and von Heeder, *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study*, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 11, Nr. 2, pp. 191-214, 2002

[8] Muthig and Atkinson, *Model-Driven Product Line Architectures*, SPLC 2002, LNCS 2379, pp. 110-129, 2002

[9] Thiel S. and Hein A., *Systematic Integration of Variability into Product Line Architecture Design*, SPLC 2002, LNCS 2379, pp. 130-153, 2002

[10] van Zyl, *Product Line Architecture and the Separation of Concerns*, SPLC 2002, LNCS 2379, pp. 90-109, 2002

Modeling Evolution and Variability of Software Product Lines Using Interface Suites [*]

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

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

Abstract

Evolution of a software product line means extending the product line by new products. A new product keeps relevant features of old products and introduces new features defined by domain requirements. In this paper, we propose an interface-role UML based approach to construct software product line variations. A product line and its variations are specified in a UML design profile, which has a process semantics and a defined inheritance relation on specifications. Using the definition of inheritance we construct a product line model, specify new product variations and check that the new variants do not affect behaviour of the old products.

1. Introduction

The concept of a Software Product Line (SPL) is one of the complex concepts of software reuse that covers business, organization, process and technology [1]. A software product line is a set of products sharing a common architecture and a set of reusable components. Software product lines employ a top-down approach to software system development restricting a set of products in the SPL and identifying common and different requirements to all products. Requirements are usually collected by different diagrams of the UML (Unified Modeling Language [2, 3, 4]) and by a feature graphs [5, 6, 7, 8]. These groups of requirements define an SPL model in form of UML diagrams, a shared SPL architecture and an implementation of reusable components. Finally, actual products are derived from this common basis [6].

However, domain requirements tend to change and *the model of a concrete SPL can not be static, developed in advance.* Adding new classes and behavioural diagrams to an SPL model can destroy the behaviour of the old products. So, we need a methodology that guarantees that modifications of an SPL do not change the old features. The modeling approaches which support software product lines have a lack of mechanisms for modeling the SPL behaviour evolution. Moreover, the relations between behavioural specifications are not defined in the UML.

In this paper we offer an evolutionary way to construct an SPL model. We adapt the role approach [9, 10, 11] extending it by the inheritance relations on behavioural views and complete specifications.

We use a special kind of the role approach, interface-role modeling [12, 13, 14]. The interface-modeling 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 SPL 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 of different *components* [13] *as well as the interface suite itself represents a software component system*. So, an SPL model in form of an interface suite is related to the standard component architecture model [15].

Third, the UML profile of the interface-role approach has a process semantics and the inheritance-specialization relations defined on specifications. We use *the inheritance of interface suites* [14] *as an instrument of the evolution of an SPL model.* So, the SPL model represented by an interface suite is not static. The inheritance mechanism guarantees that a new product variant inherits some products of SPL and does not destruct the previous products of the SPL. New features and behaviour, caused by changing domain requirements, are modelled by the inheritance-specialization mechanism in such a manner that does not destruct the previous SPL features and behaviour.

The remainder of the paper is the following. In Section 2 we give an SPL example. In Section 3 we show how to

[*]The work of S.A. Roubtsov is supported by The European Economic Interest Grouping ERCIM (European Research Consortium for Informatics and Mathematics). His work is a part of the VTT Electronics Agile project: http://agile.vtt.fi.

specify the changes of an SPL in our UML profile using inheritance-specialization relations. In Section 4, we relate the definition of interface-suite inheritance with different ways of SPL evolution. We also demonstrate how to use our SPL model for constructing an SPL feature graph and a product component model. Section 5 gives a conclusion.

2. Software product line *Graph Designer*

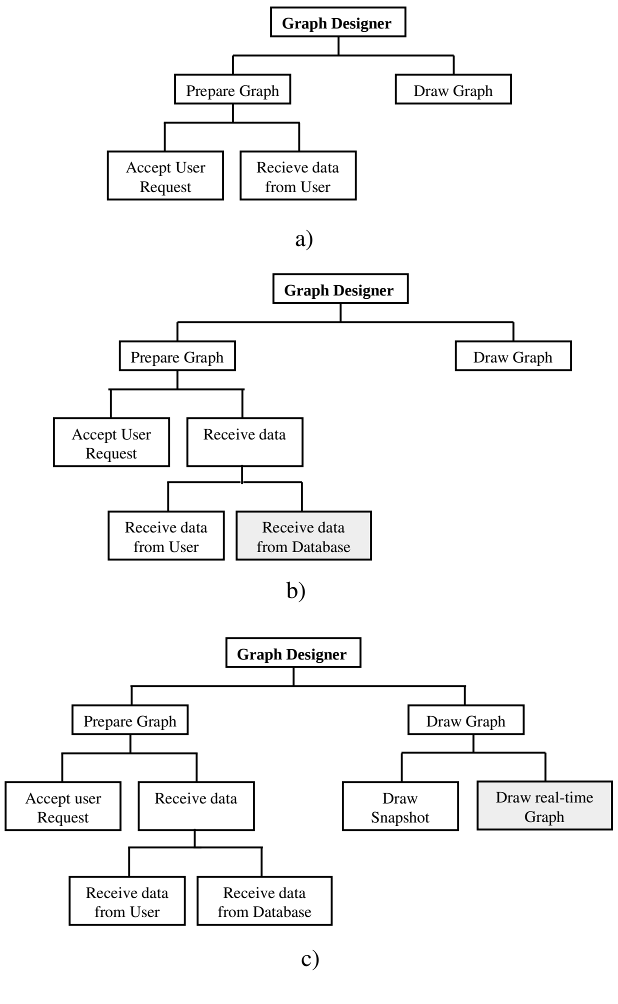


Figure 1. Feature graphs of *SPL Graph Designer*

To show variations of an SPL, we use a simple example. Let us consider software product line *SPL 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 such as type, title, legend, colour set and so on. The feature graph [5, 7] of this product is shown in Figure 1 a. All features of this feature graph are mandatory.

The second product of *SPL Graph Designer* can take data series both from a user and form a database. The feature graph of *SPL Graph Designer* is extended by feature RECEIVE DATA FROM DATABASE (Figure 1 b).

The third product draws real-time graph periodically updating data series from a database. A user starts and stops the drawing. The feature graph of *SPL Graph Designer* is enriched by feature DRAW REAL-TIME GRAPH (Figure 1 c).

We can continue constructing products, however, we have developed a case sufficient for illustrating our ideas.

3. A UML profile with inheritance relations for modeling of Software Product Lines

We specify SPL products and complete SPLs as **interface suites** (IS). An interface suite is a set of roles communicating via interfaces [13]. Roles and interfaces are abstractions both from desired product features and from the implementation. On the one hand, roles and interfaces allow representing features of a product. For example, feature ”role *Graph Maker receives* data from role *User”* has verb *receives* that represents an interface provided by role *User* and required by role *Graph Maker* (Figure 2). On the other hand, roles and interfaces can be mapped on the implementation components: several roles with interfaces can be implemented as one component or one role with provided interfaces can be implemented by several components.

3.1. A UML profile with process semantics

We use a UML profile which consists of an interface-role diagram and a set of sequence diagrams [14]. To present the product variability in this profile we use the inheritance relations. We have defined those relations both on interface-role diagrams and on sequence diagrams [14].

3.1.1. Interface-role diagram. An interface-role diagram is a UML class diagram where roles are represented by classes with stereotype ≪Role≫. Interfaces of those diagrams specify sets of operations, provided by roles.

An interface-role diagram (Figure 2) is a graph

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

with two kinds of nodes and three kinds of relations:

- R is a finite set of roles . Each role $r \in R$ depicted by a box has a set of players PL_r (instances of roles). If the number of players $|Pl_r|$ is more than one, the number is drawn near the role.

- I is a finite set of interfaces depicted by circles. Each interface $i \in I$ has a set of results Res_i of the interface. Results are shown as sets of values near the interface.

- $PI = \{(r,i)|\ r \in R, i \in I\}$ defines interfaces provided by roles. Each role provides a finite set of interfaces, $|PI \cap R \times I'| \geq 0, I' \subseteq I$. The relation is depicted by a solid line between a role and an interface.

- $RI = \{(r',(r,i))|\ r',r \in R, i \in I, (r,i) \in PI\}$ defines interfaces required by roles. Each role requires a finite set of provided interfaces

  $|RI(r,PI')| \geq 0, PI' \subseteq PI$. A required interface is drawn by a dashed arrow connecting a role and a provided interface. The arrow is directed to the interface.

- $RR = \{(r,r')|\ r,r' \in R\}$ is the relation of inheritance on the set of roles. The relation is shown by a solid line with the triangle end $r' \multimap r$ directed from role-child r' to role-parent r (Figure 3).

3.1.2. A sequence diagram. A sequence diagram is a tuple

$$s = (R \times PL, T_s, A_s), \text{where}$$

- $R \times PL$ is a set of players of roles. A player of a role is represented by a box with a line drawn down from the box (Figure 2) [2];

- $T_s = \{(v,w,l)\,|\,v,w \in R \times Pl,\ l \in L = I \times Res\}$ is a labelled relation.

  Notice, that correct set

  $$T_s \subseteq R \times Pl \times R \times Pl \times I \times Res = RI$$

  corresponds to the set of required interfaces from the interface-role diagram. The relation T_s is represented by a labelled arrow between lines drawn down from boxes v and w (Figure 2,3).

- $A_s = \{(r, n : (v,w,l))$

  $|\ (v,w,l) \in T_s,\ n = 1,2,...,N,\ r = \{\omega, s_i, f_i\}\}$

  is a function which gives natural numbers to required interfaces at a sequence diagram. A_s specifies the set of actions at the sequence diagram. A natural number at the arrow allows to distinguish several occurrences of an action $a = (v,w,l)$ $(1:a)$, $(2:a)$ etc.

  Repetition symbol r is used to indicate the begin $r = st_i$ and the end $r = f_i$ of a repeated subsequence i. In principal, the sequence can have several repeated subsequences $i = 1..m$, however, it is a very rare situation in the practice of specification. By convention, we omit the empty value $r = \omega$ for all labelled arrows that do not start or end any repeated subsequence.

3.1.3. Process semantics for the UML profile. The set of diagrams in our UML profile has a process semantics of type

$$P = (p, A, T, p*, p_F)\ [16]:$$

- p is the initial state of the process. In this paper, the states are abstract. States are named by letters with numbers: $p, p_1, p_2, ..., p^F$.

- A is a finite set of actions.

- T is a set of transitions. A transition $t \in T$ defines a pair of states (p', p''), such that p'' is reachable from p' as a result of the action a, denoted $p' \overset{a}{\Longrightarrow} p''$. If we define an abstract set of all possible states $\mathcal{P}$, then $T \subseteq \mathcal{P} \times A \times \mathcal{P}$.

- $p*$ is the finite set of states reachable from the initial state p, $p* \subseteq \mathcal{P}$. The reachability relation on the set of states $\overset{*}{\Longrightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ is the smallest relation reflexive and transitive for any $p, p', p'' \in \mathcal{P}$, $a \in A$, $p \overset{*}{\Longrightarrow} p$, $\quad (p \overset{*}{\Longrightarrow} p' \wedge p' \overset{a}{\Longrightarrow} p'') \rightarrow p \overset{*}{\Longrightarrow} p''$.

- p_F is the final state of a process, $p_F \in p*$. If $p'' \neq p_F$ then exists a nonempty subset of states $p''* \subseteq \mathcal{P}$ reachable from p''.

Set of actions A is specified by the set RI of required interfaces at the interface-role diagram. An action $a = r^1.r^2.i$ is specified by interface-role diagram

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

if $i \in I, \quad r^1, r^2 \in R,\ (r^2, i) \in PI$ and $(r^1, (r^2, i)) \in RI$. If we take into account that the use of each interface can return different results res from the set Res and that a role has a finite set of instances named players $Pl,\ pl \in Pl$, then the set of possible actions is defined completely.

Set of actions A of the process is exactly defined by the set A_s of actions at the sequence diagrams. The construction of the process from the diagrams of the profile has been shown in [14]. In this paper, we assume that for each UML specification in our profile we have the corresponding process.

3.2. Inheritance Relation in the UML profile

Inheritance relation defined on the set of UML specifications is a key element for modeling Software Product Lines. Specifications in our UML profile are behaviour oriented. The inheritance of behavioral diagrams is not defined in the UML, therefore we use our own definitions of inheritance both on the interface-role diagram and the sequence diagram levels.

3.2.1. Inheritance at the interface-role diagram level. To define inheritance between interface-role diagrams, we use inheritance on roles, which is defined in the UML and represented by arrow with the triangle end. If role r_1 inherits role r_2, then we note this as follows $r_1 -\triangleright r_2$.

Let interface-role diagrams be given:

$$IR_{p_1}, ..., IR_{p_n} \text{ and } IR_q$$

$$IR_{p_i} = (R_{p_i}, I_{p_i}, PI_{p_i}, RI_{p_i}, RR_{p_i}),$$
$$i = 1...n, \ IR_q = (R_q, I_q, PI_q, RI_q, RR_q).$$

Interface-role diagram IR_q inherits interface-role diagrams IR_{p_i}, if and only if there is an interface-role diagram

$$IR_{new} = (R_{new}, I_{new}, PI_{new}, RI_{new}, RR_{new}), \text{ (Figure. 3)}$$

such that

1. Roles. $R_q = R_{p_1} \cup ... \cup R_{p_n} \cup R_{new}$, $R_{p_1}, ..., R_{p_n}, R_{new}$ are disjoint ,

2. Interfaces. $I_q = I_{p_1} \cup ... \cup I_{p_n} \cup I_{new}$, $I_{p_1}, ..., I_{p_n}, I_{new}$ are disjoint,

3. Inheritance relation on roles.
   $RR_q = RR_{p_1} \cup ... \cup RR_{p_n} \cup RR_{new} \cup RR_{d_1} \cup ... \cup RR_{d_n}$,
   where $\forall i = 1..n$:
   $RR_{d_i} = \{(r_{p_i}, r_{new}) | \ r_{p_i} \in R_{p_i}, \ r_{new} \in R_{new}, \ \&$
   $r_{new} -\triangleright r_{p_i}\}, \ RR_{d_i} \neq \emptyset$.
   So, *the relation RR_{d_i} defines subset of roles*
   $R_{d_i} \subseteq R_{new}$, which have parents in set R_{p_i}. For example, role *New Graph Designer* (Figure 3) has three parent roles. However, there is a new role *Graph Data Source* which has no parents.

4. Provided interfaces.
   $PI_q = PI_{p_1} \cup ... \cup PI_{p_n} \cup PI_{new} \cup \ PI_{d_1} \cup ... \cup PI_{d_n}$,
   $PI_{d_i} = \{(r_{d_i}, i) | \quad r_{d_i} \in R_{d_i}, \quad i \in I_{p_i}$,
   $\exists r \in R_{p_i}$, such that $r_{d_i} -\triangleright r$, and $(r, i) \in PI_{p_1}) \}$.
   Provided interfaces from roles-parents are duplicated in roles-inheritors. For example, role *New Graph Designer* (Figure 3) provides the same interfaces as its parents: *IDraw, IGetGraph, IDataSeries* .

5. Required interfaces.
   $RI_q = RI_{p_1} \cup ... \cup RI_{p_n} \cup RI_{new} \cup RI_{d_1} \cup ... \cup RI_{d_n}$,
   $RI_p = RI_{p_1} \cup ... \cup RI_{p_n}$;
   $RI_d = RI_{d_1} \cup ... \cup RI_{d_n}$,
   $RI_{d_i} = \{(x_{d_i}, (r_{d_i}, i)) \mid r_{d_i}, x_{d_i} \in R_{d_i}, \ i \in I_{p_i}$,
   $\exists r, x \in R_{p_i}$, such that $r_{d_i} -\triangleright r, \ x_{d_i} -\triangleright x$
   and $(r, i) \in PI_{p_i}$ and $(x, (r, i)) \in RI_{p_i}\}$.
   A required interface i is inherited by role x_d from role x if there is a new role r_d, which inherits role-provider r of this interface. For example, role *New Graph Designer* requires interface *IDraw* because this role inherits both the parent-provider *GraphDrawer* and the parent-requirer *Graph Designer.*

The main feature of our definition is that the roles of the interface-role diagram IR_q cannot require interfaces of parent roles from the interface-role diagrams IR_{p_i} and roles from IR_{p_i} cannot require interfaces of roles from IR_q. To be used the parent interfaces should be duplicated in roles-inheritors.

The set of required interfaces RI_q specifies the set of possible actions in the product behavior. An interface-role diagram defines n duplicating functions $RI_{p_i} \longrightarrow RI_{d_i}$, $i = 1..n$, one duplicating function for a parent. We use those functions to derive parent processes from processes [14] of new specified products and check inheritance on the sequence diagram level.

3.2.2. Inheritance at the sequence diagram level. Inheritance at the sequence diagram level is defined as inheritance of processes constructed from the set of UML sequence diagrams. Process q constructed from an inheritor specification inherits process p_i built from the parent specification if and only if process p_i is derived from the process q in the process algebra PA_q. In work [14] we have shown that the actions of process algebra PA_q are defined from the interface-role diagram of the inheritor. We also have investigated the process derivation in detail. In this paper, we assume that each new product variant, which is specified in an SPL, inherits processes of its parent products. This inheritance is checked as inheritance of processes constructed from UML specifications of products.

3.3. Example of a product line specification in the UML profile with inheritance relations

First product of *SPL Graph Designer***.** The interface-role diagram of the first product is presented in Figure 2. Role *Graph Maker* provides interface *IGetGraph*, which is required by role *User*. (We have called this role '*User*' to avoid a mix up of notions. More likely it is a graphic user interface). These two roles and the interface specify feature ACCEPT USER REQUEST from the feature graph (Figure 1). In the same way we may say that roles *User* and *Graph Maker* present feature RECEIVE DATA FROM USER via interface *IDataSeries*. At last, feature DRAW GRAPH is realized by the pair of roles *Graph Maker* and *Graph Drawer* interacting via interface *IDraw*.

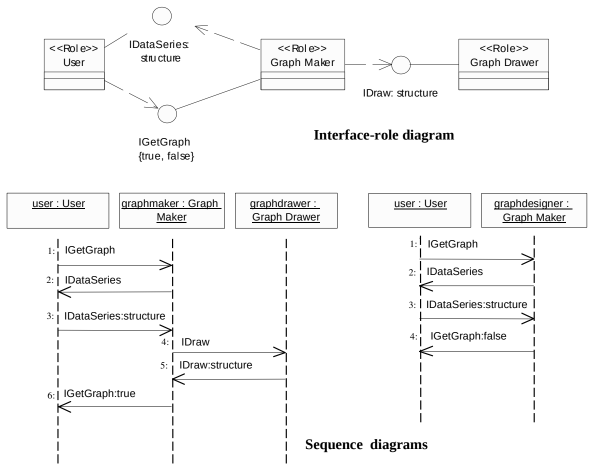
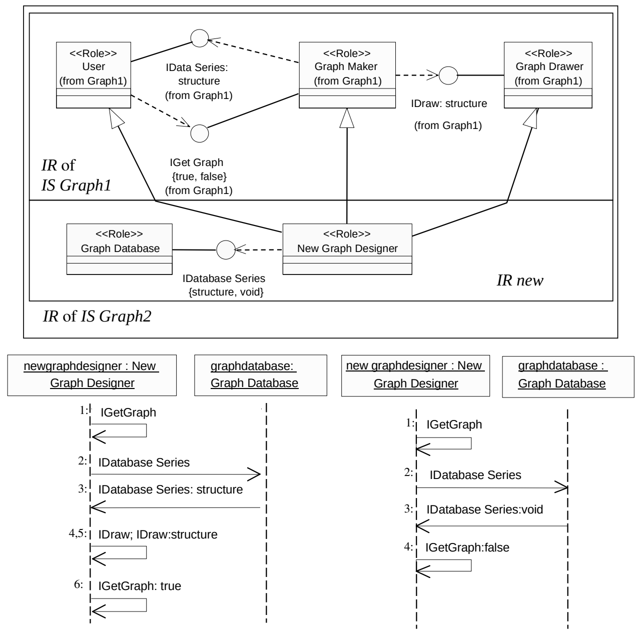


Figure 2. Interface suite for the first product of *SPL Graph Designer*

The behavioural pattern of *Graph Designer* is presented by the set of sequence diagrams (Figure 2). 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 behaviour patten 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 structures to be drawn and returns them as a result via the same interface. The last action is a 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 first sequence diagram (Figure 2). The second sequence diagram in Figure 2 corresponds to the case, when the user's data are not complete or correct to be drawn. In this case, *Graph Maker* returns result *IGetGraph:false* to *User*.

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 (Figure 3). 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 first 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*.



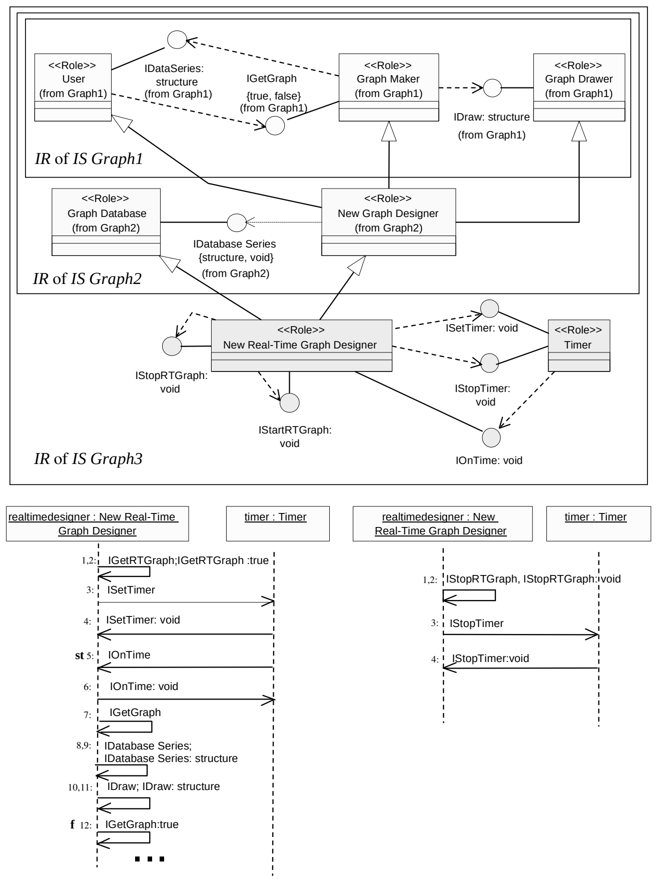
<<Role>> User (from Graph1)
IData Series: structure (from Graph1)
<<Role>> Graph Maker (from Graph1)
<<Role>> Graph Drawer (from Graph1)
IDraw: structure (from Graph1)
IGet Graph {true, false} (from Graph1)
IR of IS Graph1
<<Role>> Graph Database
<<Role>> New Graph Designer
IDatabase Series {structure, void}
IR new
IR of IS Graph2
newgraphdesigner : New Graph Designer
graphdatabase: Graph Database
new graphdesigner : New Graph Designer
graphdatabase : Graph Database
1: IGetGraph
2: IDatabase Series
3: IDatabase Series: structure
4,5: IDraw; IDraw:structure
6: IGetGraph: true
1: IGetGraph
2: IDatabase Series
3: IDatabase Series:void
4: IGetGraph:false

Figure 3. Interface suite for the second product of SPL named *Graph Designer which receives data from a database*

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 *New Graph Designer* inherits provided interfaces *IGetGraph* and *IDraw* and can require these interfaces (Figure 3) .

The set of sequence diagrams of the second product (Figure 3) differs from the set of sequence diagrams of the first one (Figure 2). However, if we construct process p from the first product and process q from the second one and rename actions of p using inheritance of roles, for example, $a = GraphMaker.User.IDataSeries$ is renamed to $a' = NewGraphDesigner.GraphDatabase.IDataSeries$, then the renamed process p is derived from q. This indicates that the behaviour has been inherited.

Third product of the SPL named *Real-Time Graph Designer* is presented by Figure 4. 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



<<Role>> User (from Graph1)
IDataSeries: structure (from Graph1)
IGetGraph {true, false} (from Graph1)
<<Role>> Graph Maker (from Graph1)
IDraw: structure (from Graph1)
<<Role>> Graph Drawer (from Graph1)
IR of IS Graph1
<<Role>> Graph Database (from Graph2)
IDatabase Series {structure, void} (from Graph2)
<<Role>> New Graph Designer (from Graph2)
IR of IS Graph2
<<Role>> New Real-Time Graph Designer
ISetTimer: void
<<Role>> Timer
IStopRTGraph: void
IStopTimer: void
IStartRTGraph: void
IOnTime: void
IR of IS Graph3
realtimedesigner : New Real-Time Graph Designer
timer : Timer
1,2: IGetRTGraph;IGetRTGraph :true
3: ISetTimer
4: ISetTimer: void
st 5: IOnTime
6: IOnTime: void
7: IGetGraph
8,9: IDatabase Series; IDatabase Series: structure
10,11: IDraw; IDraw: structure
f 12: IGetGraph:true
...
realtimedesigner : New Real-Time Graph Designer
timer : Timer
1,2: IStopRTGraph, IStopRTGraph:void
3: IStopTimer
4: IStopTimer:void

Figure 4. Interface suite for *Real-Time Graph Designer*

drawing.

First, *New Real-Time Graph Designer* starts *Timer* via interface *ISetTimer* (Figure 4). *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*. (For the sake of simplicity assume that all graph snapshots are successful in this case.)

Using the sequence diagram of the third product we have constructed the process term z corresponding to the sequence and we have proved that process z inherits process q of the second product.

Specifying a product of a product line in the defined UML profile we check inheritance of the specified behaviour. Our definition of inheritance allows to check that features of the old products are kept in the new products.

An SPL-model. Combining the interface-role diagram of the third product with the sequence diagrams of all products we construct the model of the complete *SPL Graph Designer* in terms of roles, interfaces and sequence diagram sets.

4. Interface-role SPL design and variability modeling

In the previous section we have shown how to specify a software product line by an interface suite (IS) in the UML profile. This approach allows us to derive new product variants, i.e. it supports the SPL variability modeling.

The definitions of IS inheritance at the interface-role diagram level (section 3.2.1) and at the sequence diagram level (section 3.2.2) show the ways to derive a new product variant from the old SPL products.

- We can *completely inherit the behavioural pattern of an old product.* A new product inherits full functionality of the previous one by means of inheritance of all roles of the old product interface suite. A new product can *extend* the functionality adding new roles interacting via new interfaces. The process of the old product is derived from the process of the new product. We have used this mechanism to construct the second and the third variants of *SPL Graph Designer* (Figure 3, 4).

- We can *partially inherit the behavioural pattern of an old product* . There are several ways of correct partial inheritance.

  1. *We inherit all roles and interfaces of the old product, but we use only a subset of sequence diagrams of the old product.*

  2. *We inherit all roles of the old product, but we do not use all the interfaces provided by those roles.*

For example, we want to realize product *Data Register* of the *SPL Graph Designer*. *Data Register* can not receive data series from a user, it takes data series only from a database. In such a case, *New Real-Time Graph Designer* inherits through its parent *New Graph Designer* from role *User* only its facility to require interface *IGetGraph*. Interface *IDataSeries* is not inherited (Figure 5).

3. We inherit only a subset of roles of an old product. If, for example, we have constructed product *Embedded Data Register* as an embedded software in a hardware product for automatic control of a parameter, then we do not inherit role *User*. New role *Bip* should be designed, which starts graph drawings and, maybe, produces a 'bip'-signal when the graph moves out of the given boundaries.

- We can *completely and partially inherit behavioural patterns of several old products from one SPL and several products from different SPLs.* In such a case, some roles of a new IS inherit roles from one old product, some roles - from another, some roles - from both products. Our *Embedded Data Register*, for example, definitely needs a piece of software to provide a database with real-time data from a sensor. This software piece belongs to another SPL. Multiple inheritance of interface suites allows to combine different software product lines to a new software product line.

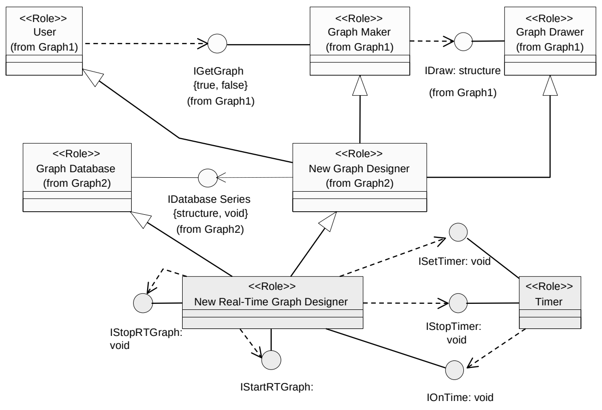


Figure 5. Interface-role diagram for variant ***Data Register***

Using our approach, it is possible to collect useful functionality specified during the SPL evolution in the form of a single interface suite. Let us name it *SPL interface suite, SPL-IS for short*. Thus, for each software product line we have

- one *SPL-IS;*
- a *variant-IS collection* which contains all implemented *variant-ISs*. The variants of a *variant-IS collection* can be used in the SPL-IS design and in the implementation of reusable components.

4.1. Interface suites and feature graphs

An *SPL-IS* represents features of an SPL. So, the feature graph presented in Figure 1 can be set out in detail in Figure 6.

- If a feature specified as an interface suite is inherited by all *variant-ISs* of an SPL, then the feature is *mandatory*. Mandatory features are drawn by boxes.
- If there are implemented *variant-ISs* which do not inherit a feature, then the feature is *optional*. Optional features are drawn by boxes with little white circles. For example, RECEIVE DATA FROM DATABASE is an optional feature.
- A depend relation on features is drawn by a dashed line with an arrow. An example of dependency between features is shown in Figure 4. We can see that role *New Real-Time Graph Designer* inherits not only role *New Graph Designer*, but also role *Graph Data Source*. So, feature DRAW REAL-TIME GRAPH depends on feature RECEIVE DATA FROM DATABASE. In all variants, where we need to draw real-time graphs, both features have to be presented. This constraint is directly derived from the *SPL-IS* model - we cannot obtain any variant with real-time drawing without inheritance of role *GraphData Source*, because its interface *IDatabase Series* acts in the sequence diagrams of the inheritor (Figure 4).
- An exclude relation on features is drawn by a dotted line with arrows in both directions. To illustrate a possible exclude relation between features, consider two features DRAW REAL-TIME GRAPH and RECEIVE DATA FROM USER. Those features exclude each other. Receiving data from *User* is not feasible for real-time graphs. So, we ought to exclude interface *IDataSeries* of role *User for all variants* of real-time drawing (*IS Graph3* in Figure 4).
- An OR-relation on features is depicted by a black arrow directed from a set of features to the parent feature. An OR-specialization of features means, that there are some products, which have all possible variant features. We can derive a *variant-IS* representing these features. For example, we can construct a variant which allows drawing shapshots as well as real-time graphs, or a variant in which data are provided by a

user or may be received from a database. So, both pairs of features may be declared as OR-specializations of their variation points.

- An XOR-relation on features is presented by a white arrow directed from a set of features to the parent feature. An XOR-specialization means, that two or more variant features must not exist in one product variant, i.e. in a single *variant-IS*. For example, our decision to reduce interface *IDataSeries* for all variants of real-time drawing converts OR-specialization RECEIVE DATA FROM USER and RECEIVE DATA FROM DATABASE into XOR-specialization.

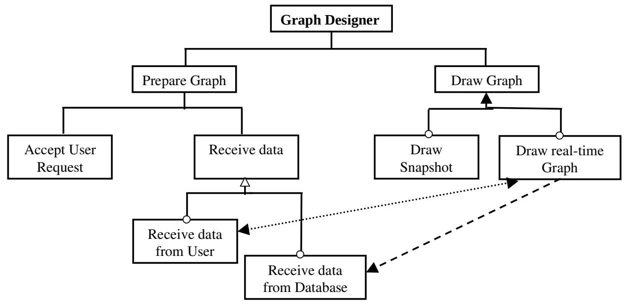


Figure 6. Feature tree of the product line *Graph Designer*

The final feature tree for our example shown in Figure 6 has four optional features, one dependency between features, one exclude relation, one variation point with OR-specialization and one variation point with XOR-specialization.

4.2. Interface suites and the software development process

Let us consider how an *SPL interface-role model* corresponds to the the software product line development process.

Figure 7 shows a standard development process of an SPL [8]. Our approach corresponds to this standard process, but we turn the process to be top down. This way we emphasize the significance of the SPL evolution. We have drawn also a zone in the standard development process where we use our interface-role models. In Figure 7 we can see places of IS-model instances (*variant-IS, SPL-IS, variant -IS collections*) in the development process.

A *variant-IS* is used as a starting point for detailed design of a product variant. A software designer is free to combine several roles in one component and put the same role in several components. An implementator can use dif-

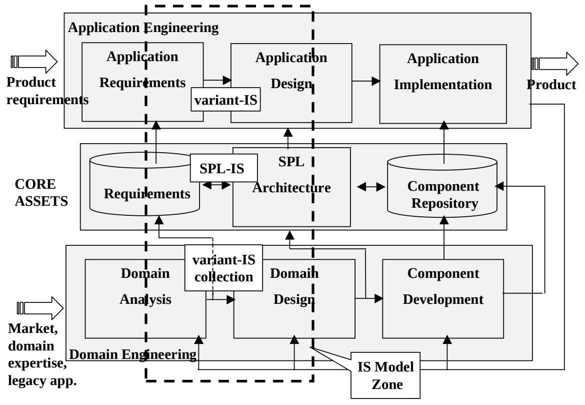


Figure 7. SPL development model and *IS* model

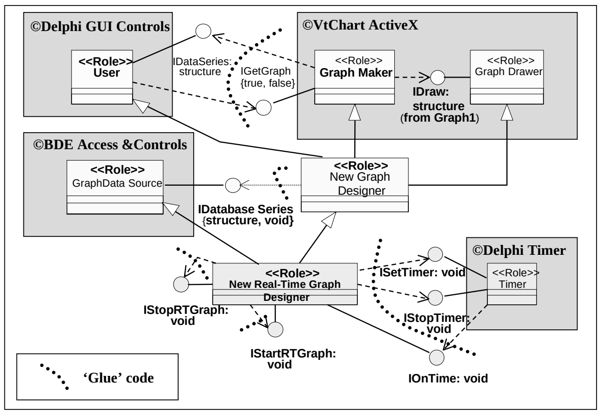


Figure 8. Mapping of the *Graph Designer IS-SPL* on component architecture

ferent implementation techniques if only they are wrapped into interface specifications.

The domain engineering process feedback provides us with *variant-IS collections* that we use during the domain analysis phase to catch commonality and variability between *variant-ISs* in the form of a *SPL-IS* model. In the analysis phase some roles and interfaces of new *variant-ISs* may be accepted as feasible for the entire SPL and saved in the *SPL-IS* model being a part of core assets.

The *SPL-IS* model evolves in the domain analysis phase and is used in the design phase. We suppose the *IS* modeling to be a bridge between these two phases. We believe that the robust design can provide such a mapping of an *SPL-IS* model to an SPL component system that *component boundaries should come across required relations between roles and interfaces*. So, we can avoid ”crosscutting roles”. The similar situation with crosscutting features is not rare in feature modeling [7]. Our confidence is based on the fact that interacting roles are abstractions of software components and, therefore, can be mapped directly onto component architecture.

To illustrate such a successful mapping of an *SPL-IS* to components, we have mapped our ”toy” product line *SPL Graph Designer* to components from the repository of Borland Delphi 4 [17].

In Figure 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.

Figure 8 shows how boundaries between Delphi components come though *SPL-IS* required relations. In coding phase we needed only some tiny pieces of ”glue” code to materialize this relations. We used the condition on constant variability realization technique [18] to implement several product line members.

5. Conclusions

Software product line engineering is a complex problem uniting customers and domain analysts, software designers and programmers.

In this paper, we have defined inheritance of interface-role models to present evolution and variability of an SPL. This approach may be useful for all professionals working on product lines. Customers and domain analysts can specify requirements in terms of roles and interfaces. Software designers and programmers can model new products via inheritance of the old SPL products. On the basis of an SPL model, software designers and programmers can plan new product variants, choose components that should be reused, realize component relations. Inheritance of interface-role models guarantees that SPL transformations do not affect the old SPL products.

References

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

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

[3] OMG, *Unified Modeling Language Specification v.1.4*, http://www.omg. org/mda/specs.htm, 2001.

[4] Fowler M., K. Scott, *UML Distilled. Applying the standard object Modeling Language* , Addison-Wesley, 1997.

[5] M. L. Griss, J. Favaro, M. d'Alessandro, “Integrating feature modeling with the RSEB,” in *Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, Los Alamitos, CA, USA, 1998, pp. 76–85, IEEE Comput. Soc.

[6] Czarnecki K. and U.W. Eisenecker, *Generative Programming. Methods, Tools and Applications*, Addison-Wesley, 2000.

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

[8] J. MacGregor, “Requirements Engineering in Industrial Product Lines,” in *International Workshop on Requirements Engineering for Product Lines, REPL'02*, Essen, Germany, 2002, pp. 5–11.

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

[10] T. Reenskaug, *Working with objects*, Manning Publications, 1995.

[11] Riehle D., *Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509*, Zrich, Switzerland, ETH Zrich, 2000.

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

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

[14] E.E. Roubtsova, R. Kuiper, “Process semantics for UML component specifications to assess inheritance,” *Elsevier Journal, Editors Paolo Bottoni, Mark Minas, Electronic Notes in Theoretical Computer Science, http://www.elsevier.nl/locate/entcs/volume72.html*, vol. 72, no. 4, 2002.

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

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

[17] Imprise Corp. Delphi Studio, *http://www.borland.com/ delphi*.

[18] M. Svahnberg, J. van Gurp, J. Bosch, “A Taxonomy of Variability Realization Techniques,” *Technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden*, 2002.

Variability management with feature models

Danilo Beuche
University Magdeburg
Universitätsplatz 2
D-39106 Magdeburg
danilo@ivs.cs.uni-magdeburg.de

Holger Papajewski
pure-systems GmbH
Agnetenstr. 14
D-39106 Magdeburg
holger.papajewski@pure-systems.com

Wolfgang Schröder-Preikschat
University Erlangen
Martenstr. 1
D-91058 Erlangen
wolfgang.schroeder-preikschat@informatik.uni-erlangen.de

Abstract

Variability management in software systems requires adequate tool support to cope with the ever increasing complexity of software systems. The paper presents a tool chain which can be used for variability management within almost all software development processes. The presented tools use extended feature models as the main model to describe variability and commonality, and provide user changeable customization of the software artifacts to be managed.

1 Introduction

While the development of single-system software is not a completely understood process yet, the need to develop sets of related software systems in parallel already exists and increases. The growing interest in concepts like software product lines and software families by industry and research groups substantiate this need. The first ideas and solution proposals of software families go back a long time in terms of computer science history. Widely known are the works of Parnas [17], Habermann [10] and Neighbors [16] from the 70s and early 80s. However, most of the work was done in the 90s, especially in the second half. Much of this work was related to organizational aspects, i.e. how to make developers in an organization efficiently develop software so that it can be used in several different products instead of just in a single one. Methods like ODM [19], FAST [22] or PuLSE mainly focus on this topic. The more technical aspects of the implementation of such systems are mostly left open in these approaches. Yet there are several techniques which cover these aspects. Examples are (static) meta-programming [6], GenVoca [3] and many others.

Common to all methods is that they use models to represent the differences and commonalities between the various resulting products or implementation fragments. The first model is a result of the domain analysis process and the latter the result of the domain design and implementation process. However, in most cases tool support for the transition from the high-level models of the domain analysis process to the product line implementation is missing. Some of the methods (e.g. FAST) propose the use of generators which accept a problem domain specific language as input and generate the implementations according to the input specification. However, even with generator-generators like in GenVoca this process is not easy and often too heavyweight for many software development projects.

In this paper we present a set of models and related tools that can be used in conjunction with almost any product line process that uses feature models[1] as representation for commonalities and variabilities. The goal was to develop a complete tool supported chain of variability management techniques which cover all phases from domain analysis to the deployment of the developed software in applications (products).

This paper is structured as follows: Section 2 discusses some problems of variability management and tool support. Section 3 introduces the basic concept of the approach. A more detailed explanation of some aspects of this approach is given in the fourth section. Section 5 demonstrates the extensibility of the approach using a case study. A brief introduction of CONSUL based tools is presented in the sixth

[1]Or any model which can be transformed into a feature model

section. Section 7 discusses some related work. The last section contains some concluding remarks and gives an outlook on future work.

2 Rationale for an open variability management tool chain

The definition of software variability as given in the workshop's CfP is:

> "Software variability is the ability of a software system or artifact to be changed, customized or configured for use in a particular context."

This definition is very open and broad. The openness is a key point. Variability management is a cross-cutting problem, which affects almost all more complex software projects to various degrees.

Variability in software systems can be found in the functional and non-functional attributes of the systems. Functional variability means that the system can provide different functionalities in different contexts. E.g. a variable HTML viewer component supports the configuration of the sets of HTML dialects it is able to render. Non-functional variability includes system properties such as memory consumption, execution speed or QoS of system functionalities.

These different aspects of variability can be realized in many different ways. The following list is an attempt to categorize where and how variability is expressed:

Programming language level: the variability is expressed using the programming language which is used to implement the system, for instance Java, C++ or C. This involves language features like conditional execution, function parameters and constants. Some of the variability is resolved at compile time[2], the remaining variability is resolved at runtime.

Meta language level: a meta language is used to describe variable aspects of the software artifacts. Examples are aspect oriented languages like AspectJ or AspectC++, meta programming systems like COMPOST [1], or BETA [15]. Even the C/C++ preprocessor language is an albeit simple example but nevertheless probably most widely known meta language for variability representation. The binding time of variability depends on the language concepts. In most cases the actual result of the binding is expressed in a basic (non-meta) programming language, which is then compiled or executed.

[2]If the compiler is able to optimize the resulting code based on partial evaluation, i.e. replacement of constant expressions with theirs results etc.

Transformation process level: almost every software is transformed from higher level language(s) into an executing system through several steps of transformations. For instance a C program is compiled by a compiler into an intermediate representation (.o files) which in turn is linked against a set of libraries by the linker, and is finally loaded into the memory of a particular computer system by the operating system's program loader. Most of the involved transformation tools can be parameterized so that the resulting system changes. I.e. the compiler has several levels of optimization, which may influences the memory footprint and/or execution speed of the compiled system. The transformation process is usually controlled by a tool like make [20] or ant [2] that interprets a transformation process description.

In most software systems, several levels of variability expressions are used together or independently. The small example shown in Figure 1 demonstrates such a mix of levels. It shows a small C source file and a makefile which is used to produce two different executables from the same source code. The point of variability is the second argument of the `printf` function. The preprocessor macro defines this value if the value of `HW_TEXT` is not already set by other means. The makefile includes two different transformation rules for the same source, the second uses a compile option to set the value of `HW_TEXT`.

```
#include <stdio.h>

#ifndef HW_TEXT
#define HW_TEXT "Hello, world!"
#endif

int main(int argc, char* argv[])
{
  printf("%s\n",HW_TEXT);
}
```

```
all: hw_en hw_de
hw_en: hw.c
   $(CC) -o $@ $<
hw_de: hw.c
   $(CC) -o $@ \
   "-DHW_TEXT=\"Hallo, Welt!\"" $<
```

Figure 1. A very simple example of variability management with C and make

In most cases, such a mixing of levels is needed to accomplish the goals of the software development in terms of efficiency, organization, reuse etc. However, tool support for controlling these highly complex mixes is very limited. Especially an automated coupling of high level models of variability and commonalities (VC) with the "low-level" implementations of the variability is rarely to be found.

Several important issues have to be considered when developing a tool chain to support the complete process of variability management:

- Easy, yet universal model(s) for expressing variability and commonalities should be supported.
- Variability at all levels must be manageable.
- Introduction of new variability expression techniques should be possible and easy.

The CONSUL (CONfiguration SUpport Library) tool chain presented in the next section tries to meet all these requirements.

3 CONSUL overview

The CONSUL tool chain has been designed for development and deployment of software program families. The core of CONSUL are the different models which are used to represent the problem domain of the family, the solution domain(s) and finally to specify the requirements for a specific representative (member) of the family.

The central role is played by *feature models* which are used to represent the problem domain in terms of commonalities and variabilities. CONSUL uses an enhanced version of feature models compared to the original feature models as proposed in the FODA method [12]. A detailed description of those enhancements is given in Section 3.1.

The solution domain(s) (i.e. the implementations) are described using the *CONSUL Component Family Model* (CCFM). It allows to describe the mapping of user requirements onto variable component implementations, i.e. the customization of a set of components for a particular context. As the name suggests, this model has been newly developed for CONSUL. The CCFM is presented in detail in Section 3.2.

The *feature sets* are used at deployment time and describe a particular context in terms of features and associated feature values.

Figure 2 illustrates the basic process of customization with CONSUL. Most steps can be performed automatically once the various models have been created. The developers of variable components have to provide the feature models, the component family models, and the implementations itself. A user[3] provides the required features, the tools analyze the various models and generate the customized component(s).

[3]Here a user can be either human or also a tool which is able to derive the set of required features automatically from some input

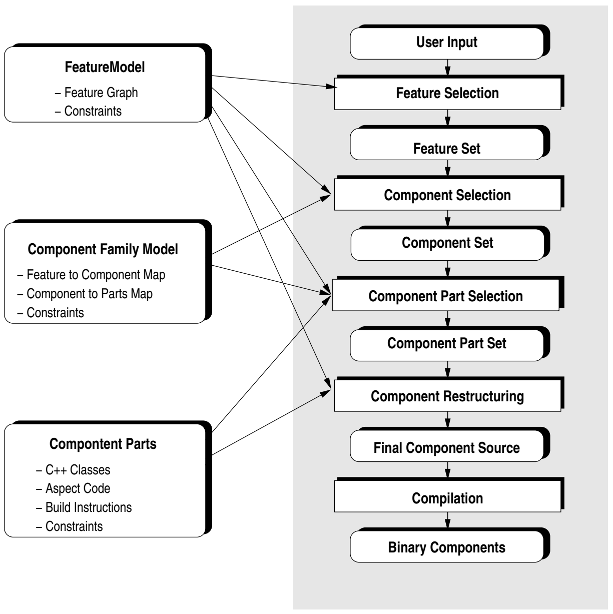


Figure 2. Overview of CONSUL process

The key difference between CONSUL and other similar approaches is, that CONSUL models in most cases only describe *what* has to be done, but not *how* it should be done. CONSUL provides only basic mechanisms which can be extend according to the needs of the CONSUL user. This flexibility is achieved by combining two powerful languages inside CONSUL and allowing the user to extend this system.

The first language is Prolog, a widely known language for logic programming. Prolog is used for constraint checking, i.e. for expressing relations between different features. The same logic engine is used for component selection and customization.

The second language is a XML-based language called XMLTrans which allows to describe the way customization (transformation) actions are to be executed. The most simple transformation is the verbatim inclusion of a file into the final customized source set. Even for this simple transformation different solutions are possible. On systems where file system links are possible, the inclusion action can be described differently in a different way than on systems without such file system capabilities. XMLTrans allows the tool users to describe similar and more complex transformations in a special XML language. Due to its modular structure, it can be extended with user supplied transformation modules. This can be used to provide seamless access to special generators or other tools seamlessly from within the tool chain.

3.1 CONSUL feature models

Feature modeling is a relatively simple approach for modeling the capabilities of a software system introduced by Kang et al. [12]. A feature model represents the commonalities and variabilities of the domain. A feature in FODA[4] is defined as an *end-user visible characteristic of a system*.

CONSUL uses feature models because on one hand they are easy to understand, but on the other hand are able to express relatively complex relations in a very compact manner. To enable modeling of more complex scenarios, CONSUL uses a slightly enhanced version of feature models compared to the original concept. The enhanced versions allows to attach typed values to features to represent non-boolean feature informations and additional relation rules called restrictions.

Features are organized in form of *feature models*. A feature model of a domain consists of the following items:

Feature description: each feature description in turn consists of a feature definition and a rationale.

The definition explains which characteristic of the domain is described by the feature, so that an end-user is able to understand what this feature is about. This definition may be given as informal text only or in a defined structure with predefined fields and values for some information like the binding of the feature, i.e. the time a feature is introduced in the system (configuration time, compile time, etc.).

The rationale gives an explanation when to choose a feature, or when not to choose it.

Feature value: each feature can have an attached type/value pair. This allows to describe non-boolean features more easily.[5]

Feature relations: the feature relations define valid selections of features in a domain. The main representation of these relations is the *feature diagram*. Such a diagram is a directed acyclic graph where the nodes are features and the connections between features indicate whether they are optional, alternative or mandatory. Table 1 gives an explanation of these terms and shows its representation in feature diagrams.[6] Additional relations can be attached to a feature. CONSUL provides a flexible mechanism called *restrictions* to enable the description of arbitrary feature relations.

[4]Feature-oriented Domain Analysis

[5]Typed features with values are not part of the original feature model proposal. However, this extension is required to describe many domains and has been proven to be very useful.

[6]The graphical notation differs from the original FODA style to allow easier drawing/generation of feature diagrams.

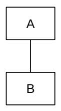
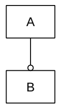
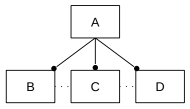
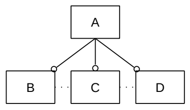
Feature Type	Graphical Representation
mandatory Mandatory feature B has to be included if its parent feature A is selected	
optional Optional feature B may be included if its parent feature A is selected.	
alternative Alternative features are organized in *alternative groups*. Exactly one feature of such the group B,C,D has to be selected if the group's parent feature A is selected.	
or Or features are organized in *or groups*. At least one feature of such the group B,C,D has to be selected if the group's parent feature A is selected.	

Table 1. Explanation of feature diagram elements

From the characteristics of the problem, a domain analyst derives the features relevant for the problem domain.

For example for a domain which requires a variable realization of cosine calculation functions for embedded real-time applications, the model could contain a feature that allows to specify the precision required for the results (`Precision`)[7], a feature that represents whether discrete angle values are used (`ValueDistribution`), a feature to express that fixed calculation time is required (`FixedTime`) and so on. The complete feature model is shown in Figure 3. A more detailed discussion of this example can be found in [5].

The feature model of a problem domain (in our case the cosine world) can be used by an application engineer, and she or he should be able to select the feature the application requires and if necessary to specify feature values.

[7]The names in parentheses are the feature names used in the resulting feature model, see figure 3.

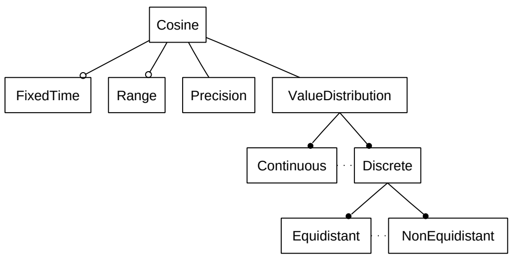


Figure 3. Feature model of cosine domain

3.2 CONSUL component family model

The component family model of CONSUL is not yet another component model in the spirit of CORBA or COM component models. CONSUL uses a very open definition of components. A component encapsulates a configurable set of functionalities. As a consequence, CONSUL cannot check interfaces of connected components itself, but allows to introduce user-definable checks appropriate for the intended framework/architecture. Figure 4 illustrates the hierarchical structure of the component based family model supported by CONSUL.

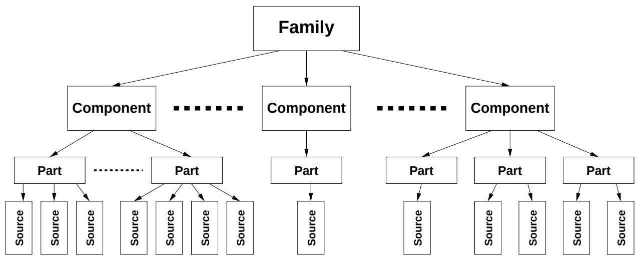


Figure 4. Structure of the CONSUL family models

This approach is reflected in the CONSUL family description language (CFDL) which mainly describes the internal component structure of a family and its configuration dependencies. The language is complementary to languages like OMG's CORBA IDL or Microsoft's COM IDL which focus on the external view of a component. The external interface of a component is merely another (possibly) configurable part of a component for CONSUL.

An small example of the language is given in Figure 5. It shows a simple component realizing the cosine example domain with just three different implementation files. Depending on the selected features one of the `cosine_?.cc` is used to implement the cosine function.

The CONSUL family model represents a family as a set of related components. The inter-component relation of these components is not fixed. I.e. both hierarchical component structures like the OpenComponent model [8] or ordinary independent components can be part of a family model. The CONSUL family description language (CFDL) is the textual representation of the model.

The following paragraphs briefly introduce the three elements of the CONSUL family model.

Components: a component is a named entity. Each component is hierarchically structured in *parts* which in turn consist of *sources*.

Parts: parts are named and typed entities. Each part belongs to exactly one component and contains any number of *sources*.

A part can be an element of a programming language like a class or an object, but also any other key element of the inner and external structure of an component, i.e. an interface description. CONSUL provides a number of predefined part types, like `class`, `object`, `flag`, `classalias` or `variable`. The introduction of new part types according to the needs of the tool users is also possible.

Section 4 gives a small demonstration of this. Table 2 gives a short description of the currently available part types in the current CFDL version.

Sources: a part as a logical element needs some physical representation(s) which are described by the *sources*. A source element is an unnamed but typed entity. The type is used by the transformation backends to determine the way to generate the source code for the specified element. Different predefined types of source elements are supported, like the `file` which simply copies a file from one place into the specified destination of the component's source code. Some source elements are more sophisticated, like `classalias` or `flagfile`, and require generation of new source code by the backends. Table 3 lists the currently available source element representations.

The actual interpretation of these source elements is handed over to the CONSUL component generator backends. To enable the introduction of custom source elements and generator rules, CONSUL allows to plug in different generators. At the moment, two different generators exist. One is implemented in Prolog and operates directly on the Prolog CONSUL knowledge database representation. The second which uses a modular transformation based approach.

The advantage of the Prolog based approach is its speed and the ability to use the power of Prolog everywhere. However, it requires a decent knowledge of Prolog to change or add source element generators. The other approach [18] uses XML to describe the transformations and allows users

```
Component("Cosine")
{
  Description("Efficient cosine implementations")
  Parts {
     function("Cosine") {
       Sources {
         file("include", "cosine.h",def)

         file("src", "cosine_1.cc",impl) {
           Restrictions { Prolog("not(has_feature('FixedTime',_NT))")}}

         file("src", "cosine_2.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('NonEquidistant',_NT")}}

         file("src", "cosine_3.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('Equidistant',_NT")}}
       }
     }
  }
  Restrictions { Prolog("has_feature('Cosine',_NT)") }
}
```

Figure 5. (Simplified) component description for cosine component

to integrate own special-purpose modules into the systems via an easy-to-use module concept. This enables users to introduce their own family specific generators without any need to change the core CONSUL tools.

Using restrictions in CFDL a key difference of the CFDL from other component description languages is the support for flexible rules for inclusion of components, parts and sources. Inclusion constraints, called restrictions, can be attached to each CFDL element.

Each element may have any number of restrictions. At least one of them has to be true to include the element into the system. If there is no restriction specified an element is always included. The CFDL itself does not specify a language for restriction description, it passes the restriction description to an external module. Currently, there is just one language model which uses Prolog as description language and allows direct access to the CONSUL knowledge database[8].

The code of restrictions can access the complete CONSUL model set (feature model, component model, feature set) to make a decision. This allows the customization of components according to the specified needs of the applications on a structural base. In combination with the ability of the backend transformation to produce specialized source elements based on arbitrary parameters and structural informations, this permits almost any customization concept to be used in conjunction with CONSUL.

[8]Although this direct access is very powerful, it has its drawbacks, since it is very easy to make mistakes in Prolog statements, without breaking the syntax. For most statements, an easier, more problem-oriented language would be sufficient. It will be included in a new release of the CFDL.

4 Closing the gap: family variation vs. family member flexibility

One of the main problems of family based software designs is that there are two levels of flexibility or variation in the design. On the one hand there is the "usual" flexibility a family member or a single application has to provide and on the other hand there is the variation inside the family to provide different family members. Both levels cannot be completely separated in a design, often the same design can represent both, family variation and member flexibility.

The following example will illustrate this problem and give an idea how CONSUL can be used to deal with it.

A very important service of any operating system is to provide access to the hardware connected to the processor. Depending on the hardware configuration and/or the needs of the software the operating system has to provide software components and interfaces to different sets of devices. Even if there is a hard disk controller device available in a system, if the software does not require disk access, a disk driver does not have to be included in the system.

The example is based on a fictitious hardware which has three different types of analog/digital converters (ADC) available. The goal is to provide a software design and implementation which adapts easily to different hardware configurations without having to implement different versions of the device drivers. The scalability shall be achieved by

Part Type	Description
`interface` (X)	represents an external component interface X.
`class` (X)	represents a class X with its interface(s), attributes and source code.
`object` (X)	represents an object X.
`classalias` (X)	represents a type-based variation point in a component. A classalias is an abstract type name which is bound to a concrete class during configuration.
`flag` (X)	represents a configuration decision. X is bound to a concrete value during configuration. Depending on the physical representation chosen for the flag, it can be represented as a makefile variable, a variable inside a class or even a preprocessor flag.
`variable` (X)	similar to a flag, but a variable should not be used for configuration purposes.
`project`(X)	represents anything which cannot be described by the part types given above.

Table 2. Overview of CFDL part types

Source element	Description
`file`	represents a file which is used unmodified.
`flagfile`	represents a C++ preprocessor flag.
`makefile`	represents a makefile variable.
`classalias`	represents a C++ typedef variable.

Table 3. Overview of CFDL source element representations

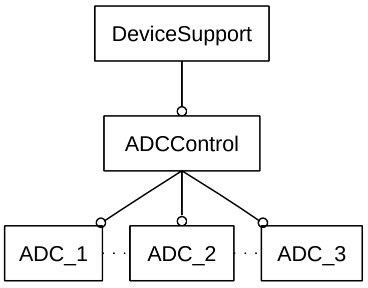


Figure 6. Partial feature model for the ADC example

using the services of CONSUL.

Figure 6 shows the relevant part of the feature model. When ADCSupport is selected, any combination of support for the three different ADC types can be requested. Thus there are seven (three single, three double, one triple) combinations of functional support for ADCs possible. In some application it is known in advance which ADC(s) are going to be used, so compile-time binding should be possible. But there could be applications which will bind an ADC at load-time, and some will defer the decision until run-time and may request access to different ADC over the time.

The drivers shall be realized within a single component. All ADC must provide the same interface to enable switching between different ADCs.

This setting seems to be a classical example for the use of an abstract base class, defining the common interface and three different subclasses which are the concrete realizations of the interface. However, in many configurations, as shown in Figure 7, the base class is not necessary since there is only one class derived from it in use. While the use of abstract base classes is appropriate for modeling and communicating interfaces to users and developers, it requires additional resources during runtime. To implement the run-time variability, C++ as well as other object-oriented languages rely on tables associated with each object derived from abstract base classes. Each table stores the location of the method implementations for the common interface of the abstract base. In C++ these tables are usually called *virtual method tables*. Use of such tables consumes memory for storing the table, and run-time since for each call to an abstract method the corresponding table is consulted.

The measurements for an abstract/concrete class pair

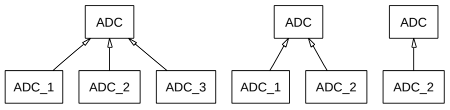


Figure 7. Class hierarchies for 3 different members

with just one virtual method (see Table 4[9]) clearly show that there is an increased memory use for the abstract class version. Especially critical is the use of data memory. Without virtual methods, no data memory is used. Many embedded microcontrollers have separate code and data memories, and often the data memory is quite small (few bytes to some kBytes) so wasting a few dozen bytes of data memory can be a real problem. A skilled embedded programmer would avoid using virtual method whenever possible[10]. To achieve the same resource usage as a hand-coded solution, the variable implementation of drive component should avoid using virtual methods whenever possible.

Hierarchy	Processor	Code	Data
non-virtual	x86	32	0
virtual	x86	206	140
non-virtual	AVR90Sxxxx	80	0
virtual	AVR90Sxxxx	284	42

Table 4. Memory consumption of abstract and non-abstract classes

To solve this problem the `classalias` of CONSUL can be used. The `classalias` part type allows description of flexible, statically changeable class relations. Figure 8 shows a new class hierarchy where the external component interface `ADC` can be mapped to any of the `ADC_?` classes.

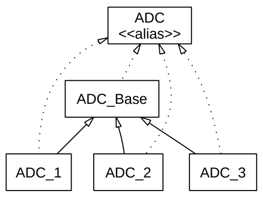


Figure 8. Variable class hierarchy for ADC component

The corresponding component description is shown in Figure 9. The concrete class to which the alias should be set is determined by the four `Value` statements given inside the classalias definition. The evaluation of the second argument of each statement is done top-down. The first argument of the first statement which evaluates to true is used to calculate the class name. In the example, one of the predefined clauses of CONSUL is used. The clause `is_single(X,_NT)` is true when only feature `X` is selected from its corresponding or-feature group. The last statement ensures that if there is more than one feature selected from the group, the abstract base class is used.

To solve the problem of having an abstract base class or not for the `ADC_{1,2,3}` class, the class `ADC_Base` has two different declarations, one as abstract class, and the other as just an empty class definition.

The description of class `ADC_1` is straightforward, it is included in the component whenever support for `ADC_1` is requested. For the other two classes, the descriptions look alike.

It is obvious that the mechanisms for variability used in this example could be used without CONSUL. Changing a class hierarchy could be accomplished using a conditional `#include` resolved by the C++ preprocessor according to a compiler argument which is defined in a makefile. However, with the CONSUL and the CFDL there is one single place to manage the customization process. The information what and how to configure is not spread out over different files in different languages. CONSUL and CFDL separate the structure of systems and components from the source files they are implemented in.

Using AOP to do the trick: the extensibility of the CFDL through its customizable backend makes the introduction of new high-level description elements very easy. Going back to the example given above, there has been some tricking around with the base class of `ADC_{1,2,3}`. It was necessary to provide a fake (empty) base class when the abstract base class should not be used.

The aspect language AspectC++ [7] allows to write aspects for the C++ language which are able to introduce new base classes to arbitrary classes. The use of that feature makes the solution for the ADC example much easier, if the CONSUL would allow a statement to set the base class similar to a class alias.

To make this available in the CFDL, it is necessary to define a new part source type named `baseclass` which takes two arguments, the name of the intended base class and the privilege level (`private, public, protected` for C++).

The addition of a new source element requires only the addition of a new transformation rule to the CONSUL generator backend library. When the XML based backend is used, this requires writing an XML transformation description. With the Prolog backend, the same can be accomplished with appropriate Prolog rules.

Figure 10 shows the modified component description and Figure 11 the generated aspect code.

Using this extension mechanisms, CONSUL can be used to control and combine arbitrarly complex tools to produce the intended customized system. It can be even used to im-

[9]Compiler: gcc 2.96 for x86, gcc 2.95.2 for avr, size values in bytes

[10]Today, most programmers avoid this problem by not even using object-oriented languages for embedded systems programming

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
     classalias("ADC") {
       Sources {
         classaliasfile("include", "ADC.h","ADC") }
       Value("ADC_1",Prolog("is_single('ADC_1',_NT)"))
       Value("ADC_2",Prolog("is_single('ADC_2',_NT)"))
       Value("ADC_3",Prolog("is_single('ADC_3',_NT)"))
       Value("ADC_Base",Prolog("true"))
     }
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT))")
           }
         file("include", "ADC_Base.h",def,"include/ADC_Base_empty.h") {
           Restrictions {
     Prolog("selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT)")
         } } } }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
           { Restrictions { Prolog("has_feature('ADC_1',_NT)") } }
       } }
     ...
    } }
  Restrictions { Prolog("has_feature('ADCControl',_NT)") }
}
```

Figure 9. CFDL for ADC component

```
aspect consul_ADC_1_ADC_Base {
  advice classes("ADC_1"):
     baseclass("public ADC_Base");
};
```

Figure 11. Aspect code generated for the CFDL `baseclass` source element

plement simple source code generators directly, as shown above.

5 CONSUL case study: Pure

To evaluate the CONSUL ideas, it was necessary to use it in a larger project. The Pure operating system family for deeply embedded systems [4] developed at the University Magdeburg, was an ideal target.

The Pure operating system family consists of about 321 classes implemented in some 990 files. Pure runs on nine different processor types from 8 bit to 64 bit processors and is almost entirely written in C++. Prior to the use of CONSUL, the configuration was done by modifying/setting several C++ preprocessor `#define` statements (about 64) and also some makefile variables. Due to its application area Pure is trimmed to use hardware resource as efficiently as possible. For every application it tries to provide exactly the features an application needs, not more.

The result of the domain modeling using feature models was a model of the PURE problem domain with some 250 features. The model allows approx. 2^{105} different valid feature combinations. The component family model representing the implementation consists of 57 components.

A feature set for a typical configuration has some 20 features. The smallest possible set contains just three features (describing the used compiler, the target cpu model and the target hardware platform), selecting 20 classes. A typical configuration supporting preemptive multitasking with time slices has 94 classes[11]

Using CONSUL reduced the risk of misconfiguration, because the feature model and the CFDL allows to express dependencies and these can be checked automatically. Prior

[11] Both configurations are for a x86 PC based target platform and the GNU Compiler, values for other target platforms may differ slightly.

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
    ....
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT))")
           } } } }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
 // introduce new base class when not single
 baseclass("ADC_Base","public")
           { Restrictions { Prolog("not(is_single('ADC_1',_NT))") } }
       } }
     ...
    } }
  Restrictions { Prolog("has_feature('ADCControl',_NT)") }
}
```

Figure 10. CFDL for ADC component using the baseclass() source element

to the availability of CONSUL tools for Pure configuration most Pure developers used only two or three well known configurations, because finding a new working configuration was very complicated. Today, the test directory contains some 120 different base configurations. A new working configuration is typically created in a few minutes.

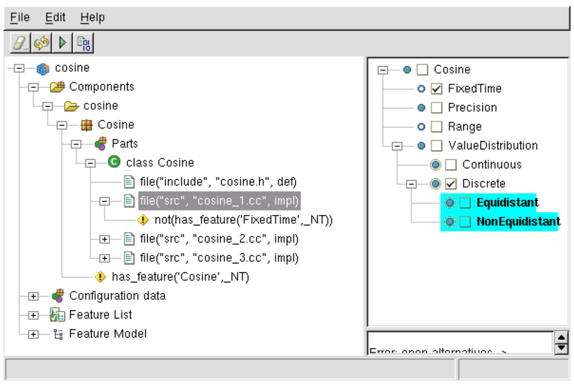
6 CONSUL based tools

Variability management tools have to be used by two different classes of users. The first class is formed by the developers who develop variable software artifacts, the second class by the deployers of these variable artifacts. As a complete tool chain, CONSUL supports both classes.

The modular implementation of CONSUL allows flexible combination of the required services and user interfaces to build different tools. The current application family consists of following three different tools:

Consul@GUI The main application for developers is `Consul@GUI`. Consul@GUI is an interactive modeling tool for CONSUL models. It allows to create and edit the models but can also be used in the deployment of the developed software for generating the customized software.

Figure 12 shows a screenshot of a configuration session. It shows the feature model for the cosine domain with several features selected. The configuration is not valid, since there is still an open alternative. This is indicated by the different background colors of the two features `Equidistant` and `NonEquidistant`.



```
cosine
  Components
    cosine
      Cosine
        Parts
          class Cosine
            file("include", "cosine.h", def)
            file("src", "cosine_1.cc", impl)
              not(has_feature('FixedTime',_NT))
            file("src", "cosine_2.cc", impl)
            file("src", "cosine_3.cc", impl)
        has_feature('Cosine',_NT)
  Configuration data
  Feature List
  Feature Model
```

```
Cosine
  ✓ FixedTime
  Precision
  Range
  ValueDistribution
    Continuous
    ✓ Discrete
      Equidistant
      NonEquidistant
```

Figure 12. Consul@GUI

Once a valid configuration has been found, the generation process can be started.

Consul@CLI Based on CONSUL a customization tool with a command line interface has been built as well. This tool can be used e.g. together with make to provide automated customization when (re)building a software system.

Consul@Web It is also possible to make software customization available via web browsers. A demonstration based on a Java applet can be found at `http://www.`

pure-systems.com/consulat/. It allows the configuration, building and downloading of Pure via an Java-enabled web browser.

7 Related works

There are not many tools for language-independent, cross-level management of software variability available. The company BigLever with their product GEARS [14] is one of the few. GEARS operates on the file system level to manage variability. It allows to specify conditions for the inclusion of a specific file into a resulting system. However, there is no complete domain model, but several independent sets of parameters are used to describe the conditions. Although this might enhance the reusability, this restricts the description of cross-component dependencies.

Several other approaches use feature models for domain modeling [9, 13]. However, most of them do not use an explicit feature modeling tool which effectively limits the size of the models. In [21] a tool is described which operates on a feature model and is able to generate java class skeletons from feature models.

The transformation process in CONSUL, which produces the customized implementation from component descriptions has some similarities to frame-based source generators like COMPOST [1] or XVCL [11]. The idea of frames blends perfectly with the ideas of CONSUL. The open model of the CONSUL tools allows the integration of such a generator into the transformation process, and the parameterization of the generator is controlled vi the feature model and the component family model constraints.

8 Conclusions

This paper presented an extensible tool chain for variability management. The main model types are an enhanced feature model and a flexible component based family model which enable language independent representation of variability in software systems.

Compared to other tools for variability management CONSUL is more flexible through its extension mechanisms. The use of feature models as the model for communication between the developers of variable software and the deployers has been proven to be an effective solution.

One of the problems of CONSUL is that Prolog is not very well suited as a description language for users. Its syntax rules are to weak to detect typical typos in user defined rules, and the Prolog language system tends too produce very unpredictable results in these cases. A new language for expressing the basic restrictions is in development and will replace the use of “native” Prolog in many places.

Among the future projects based on CONSUL are an integration of CONSUL technology into integrated development environments like Eclipse or VisualStudio. To enhance interoperability with other tools the component family model will be mapped to an XMI representation, allowing direct use inside UML tools like Rational Rose or ARGO/UML.

References

[1] U. Aßmann. Beyond Generic Component Parameters. In J. Bishop, editor, *Proc. of the Component Deployment IFIP/ACM Working Conference*, volume 2370 of *Lecture Notes in Computer Science*, pages 141–154, Berlin, Germany, June 2002. Springer.

[2] Ant Project Homepage. see http://jakarta.apache.org/ant/.

[3] D. Batory, J. Thomas, and M. Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *Proceedings of ACM SIGSOFT*, 1994.

[4] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The Pure Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *IEEE Proceedings ISORC’99*, 1999.

[5] D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Fine-grain Application Specific Customization for Embedded Software. In *Proceedings of the International IFIP TC10 Workshop on Distributed and Parallel Embedded Systems (DIPES 2002)*, Montreal, Canada, Aug. 2002. Kluwer Academic Publishers.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[7] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, Oct. 2001.

[8] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. Open Components. In *Proc. of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa, Florida, Oct. 2001.

[9] M. L. Griss, J. Favaro, and M. d’Alessandro. Integrating Feature Modeling with the RSEB. In *Proc. of the 5th International Conference on Software Reuse*, pages 76–85, Victoria, Canada, June 1998.

[10] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[11] S. Jarzabek and H. Zhang. XML-based Method and Tool for Handling Variant Requirements in Domain Models. In *Proc. of 5th IEEE International Symposium on Requirements Engineering RE01*, pages 116–173, Toronto, Canada, Aug. 2001. IEEE Press.

[12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Nov. 1990.

[13] K. C. Kang, K. Lee, J. Lee, and S. Kim. Feature Oriented Product Lines Software Engineering Principles. In *Domain*

Oriented Systems Development — Practices and Perspectives, UK, 2002. Gordon Breach Science Publishers. to appear.

[14] C. Krueger. Variation Management for Software Production Lines. In *Proc. of the 2nd International Software Product Line Conference*, volume 2379 of *LNCS*, pages 37–48, San Diego, USA, Aug. 2002. ACM Press. ISBN 3-540-43985-4.

[15] M. Löfgren, J. L. Knudsen, B. Magnusson, and O. L. Madsen. *Object-Oriented Environments - The Mjolner Approach*. Prentice-Hall, 1994.

[16] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–573, Sept. 1984.

[17] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[18] S. Roemke. XML-Based Modular Transformation System. Master's thesis, Computer Science Faculty, University Magdeburg, Magdeburg, Germany, 2002. In German.

[19] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. STARS Organizational Domain Modeling (ODM) Version 2.0. Technical report, Lockheed Martin Tactical Defense Systems, Manassas, VA, USA, 1996.

[20] R. Stallman and R. McGrath. *GNU Make Documentation Version 0.51 for make Version 3.75 Beta*, May 1996.

[21] A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, pages 1–17, 2002.

[22] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Approach*. Addison-Wesley, 1999. ISBN 0-201-69438-7.

Towards Managing Variability using Software Product Family Architecture Models and Product Configurators[1]

Timo Asikainen, Timo Soininen, Tomi Männistö
Helsinki University of Technology, Software Business and Engineering Institute (SoberIT)
PL 9600, FIN-02015 HUT
{timo.asikainen, timo.soininen, tomi.mannisto}@hut.fi

Abstract

In this paper, we study the possibility of applying configurator tools developed for configuring mechanical and electronics products to representing and managing the variation points in a software product family. We compare at the conceptual level software architecture description languages and configuration modelling. Based on the analysis we are able to define a way of representing much of the architectural knowledge using the configuration modelling concepts. Thus, it seems feasible to provide software configuration support using configurators if the software is represented through architectural descriptions. However, there are also some differences that require extending the current conceptualisations of configuration knowledge and tools to capture software products adequately.

1. Introduction

Software product families (or lines) have been proposed as a means for increasing the efficiency of software development and controlling complexity and variability of software products. They have become increasingly important in the software industry [1,2]. A *software product family* can be roughly defined to consist of a common architecture, a set of reusable assets used in systematically producing, i.e. deploying, products belonging to the family, and the set of products thus produced.

In this paper we pursue one approach to modelling and managing the variability of software product families, based on viewing them as configurable software product families. A configurable product is such that each product individual is adapted to the requirements of a particular customer order on the basis of a predefined configuration model [3]. Such a model explicitly and declaratively describes the set of legal product variants by defining the components out of which it can be constructed and their dependencies on each other. A specification of a product individual, i.e., a configuration, is produced based on the configuration model and particular customer requirements in a configuration task. Efficient knowledge based systems for configuration tasks, product configurators, have recently become an important application of artificial intelligence techniques for companies selling products adapted to customer needs [4,5]. Stated roughly, a configurator supports the deployment process by preventing combinations of incompatible components, by making sure that all the necessary components are included, and by deducing the consequences of already made selections of components based on the dependencies in a configuration model. They are also capable of automatically generating an entire correct configuration based on requirements posed by a user. The tools are based on declarative, unambiguous modelling methods and sound inference algorithms for these. Thus, only the configuration model needs to be created to support a new product family. There are also several reports on successful use of product configurators in practice [6,7,8].

The most systematic software product families closely resemble configurable products in that they are composed of standard re-usable assets and have a predefined architecture [1]. Our approach is based on the assumption that, although customer-specific programming may be required, a significant portion of the assets can be developed and systematically modelled in advance of their deployment.

A major effort has been spent on developing architecture description languages (ADLs) that can be used for representing these re-usable assets and software architectures. Thus, product families and ADLs seem natural counterparts in the software domain for configurable products and configuration modelling languages. There are many ADLs and large differences between them [9,10].

We study the possibility of applying methods and tools developed for modelling and configuring mechanical and electronics products to configuring software. A prerequisite for coming up with a general solution to this problem is to define a mapping from the conceptualisation of software systems to a conceptualisation of configuration knowledge. Towards this end, we analyse three prominent ADLs at the conceptual level and compare them with the major concepts used for modelling configuration knowledge. Based on the analysis and comparison, we show how to represent main concepts of ADLs using the configuration modelling concepts. In addition, we identify several

[1] This is an extended and revised version of a paper presented in the Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002).

potential needs for extending the configuration modelling concepts with ADL derived concepts.

For the purposes of this paper we concentrate on three important ADLs: Acme [11,12,13], Wright [14,15] and Koala [16,17,18]. Out of these, Acme has been designed to include features of other ADLs that its designers considered central. The relevance of Acme is further promoted by the fact that one of the goals of Acme is to serve as an interchange language for other ADLs. Wright is a widely cited ADL that has a rigorous semantics and describes behavioural aspects of software. Both the use of formal methods and description of behaviour make Wright important among ADLs. Koala is not precisely an ADL, but it is in commercial use at Philips Consumer Electronics for documenting products in a product population. Therefore, being one of the few ADLs in industrial use, Koala is an important example of the practical aspects of ADLs.

As the reference point in the comparison, we employ a configuration ontology presented by Soininen et al. [19]. This ontology synthesises prior conceptualisations of configuration knowledge [6,7,8,20]. Moreover, it is very similar to another recognised configuration ontology presented by Felfernig et al. [21]. Thus, as it seems to cover most approaches to configuration modelling, it is a natural reference point for conceptual level analysis.

The remainder of this paper is organised as follows: An overview of the central concepts in product configuration and the modelling concepts in the configuration ontology is given in Section 2. An overview of software architecture and ADLs is provided in Section 3. Section 4 introduces our framework for analysing and comparing ADLs along with the most important characteristics of three ADLs. In Section 5, a comparison between the ADLs and the concepts of the configuration ontology is presented. A mapping from the most important concepts of ADLs to the concepts of the configuration ontology is given in Section 6 and potentially needed extensions of the ontology are discussed in Section 7. We discuss our findings and previous work in Section 8 and finally give our conclusions and topics for further research in Section 9.

2. Product configuration

In this section, we first define the fundamental concepts of product configuration. Thereafter, we introduce a configuration ontology, i.e., a conceptualisation for modeling configuration knowledge, which conceptualises these concepts.

2.1. Fundamental concepts

We define a *product* as an abstract specification of an entity that a company sells. A *product instance* is a product that is to be delivered to a customer or a design of a product, which is concrete enough to serve as a specification for producing it. [22]

In the domain of configurable mechanical and electronics products, a *configurable product* (or a *product family*) is defined as a product that comprises a large number of variants and serves the specific needs of the individual customer by allowing customer-specific adaptation of the product.

Configurable products are specified through *configuration models*, which define the basic product properties and the possibilities for tailoring them. Product instances, in turn, are specified through *configurations*, which result from completing the *configuration task.* Information systems are used to support the configuration task; such information systems are called *configurators*. As mentioned in the introduction, configurators can provide a wide range of functionality, including, e.g., making deductions based on choices a user has already made, and preventing incompatible combinations of components.

During the configuration task, *user requirements* are used to constrain the set of possible configurations represented by the configuration model until there is only one correct (with respect to the configuration model) configuration that satisfies the user requirements left.

Putting pieces together, configurable products form an approach to satisfying variable customer needs. The approach involves two separate processes: first, a product (family) is defined through a configuration model; second, configurations of the product matching specific needs of a customer are formed. Product configurators can support both of the processes. What is essential is that the effort required to create a configuration of the product is moderate compared to designing a product from scratch.

2.2. Configuration Ontology

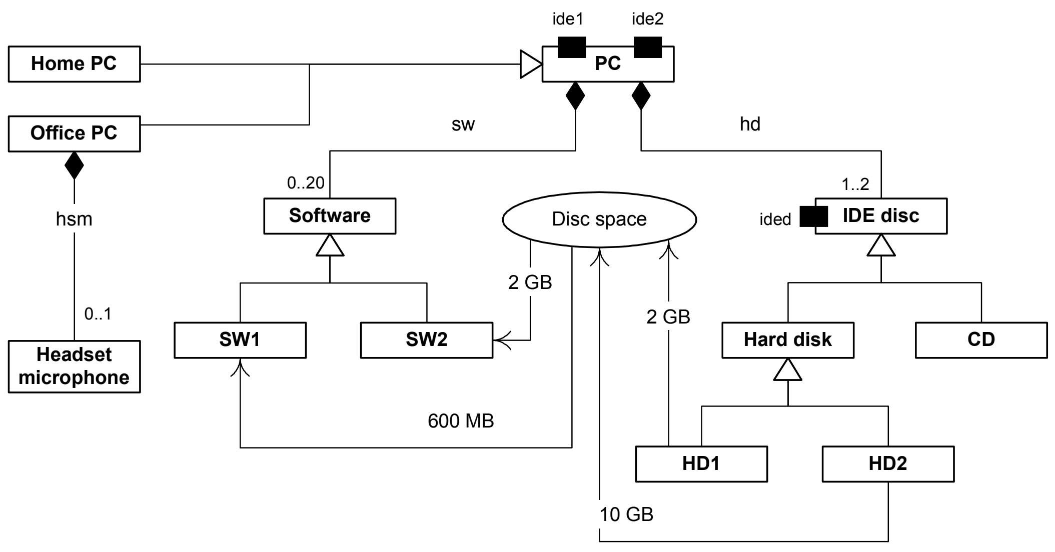
Next, we will give a short overview of a de facto standard configuration ontology used for modelling configurable products. For full details, the reader should refer to [19].

The configuration model consists of a set of *types*. More specifically, a configuration model consists of: a set of component types, a set of port types, a set of resource types, a set of function types - all the above-mentioned sets of types are organised in *is-a hierarchies*, and all types can be given *attributes* that present relevant information about the types. Additionally, component and function types are organised in composition hierarchy. Finally, a configuration model includes a set of constraints. A configuration is defined as a set of instances of the types. These instances are called *individuals*.

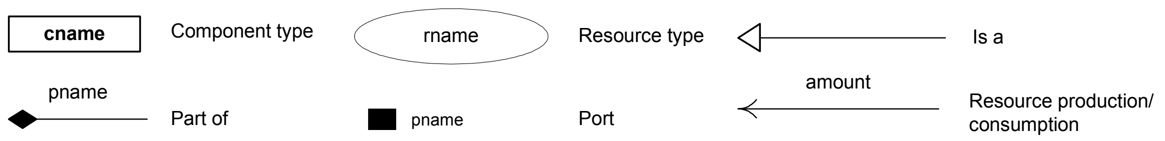
Component types represent distinguishable wholes in products.

Example. Figure 1 (a) depicts a simple configuration model of a computer, and will be used as a running example to illustrate the concepts of the ontology. Figure 1 (b) contains the legend of the notation used. The notation is no standard notation, but UML extended with some additional symbols. Type names are typeset using Arial, and instance names using `Courier`. In the figure, there are a number of component types: Home PC, Office PC, PC etc. Further, there are a number of is-a relationships between the component types: Home PC is-a PC, SW1 is-a Software etc. ■

2.2.1. Structure. The structural composition of component types is modelled by means of *part definitions*. Each part



(a)



(b)

Figure 1 (a) A simple configuration model (b) Legend of the notation used

definition includes a *part name*, a *set of possible part types*, and a *cardinality*. The set of possible part types consists of the component types the individuals of which can occur as a part of the type containing the part definition (whole type). Further, cardinality specifies the amount of individuals that must occur as part of the whole type.

Example. Component types PC and Office PC include parts: the former has two parts, `sw` of type Software, and `hd` of type IDE disc; the latter has part `hsm` with cardinality of 0..1, which implies that the part is optional. In common terms, this means that an Office PC may, but is not required, to include a headset microphone, whereas a Home PC never includes one. ■

Alternative parts can be specified by defining multiple possible part types.

2.2.2. Topology. Connections between component individuals, i.e., the topology of a configuration, can be modelled by supplying component types with *port definitions*. In common terms, ports define the connection points, i.e., interfaces, components have. A port definition includes a *port name*, a *set of possible port types*, and a *cardinality*.

The semantics of a connection are that there is a physical or logical connection between the two port individuals and the two component individuals containing them.

Example. In Figure 1 (a), there are three ports: component type PC defines ports `ide1` and `ide2`, and component type IDE disc defines port `ided`. ■

2.2.3. Resources. Resources are used in the ontology to model the production and use of entities, or the flow of such entities from one component individual to another.

Resource type defines the properties of a resource. A resource type defines whether the resource production or use must be *satisfied* or *balanced*. If the production or use is to be balanced, the production must exactly match the use; in the case they must be satisfied it is enough that production is greater than or equal to use.

The production and use of resources is specified by *production definitions* and *use definitions* in component types. These definitions specify the resource types and quantities produced or used.

Example. There is a single resource type, namely Disc space in our sample model. This resource type is consumed by individuals of component types SW1 and SW2, and produced by individuals of HD1 and HD2. ■

2.2.4. Functions. All the above-presented properties of configuration models are related to technical aspects of products. However, in many cases it is necessary to communicate more abstract characterisations of the features or functionality of configurable products to salespersons and customers.

The configuration ontology includes *functions* as a means for communicating non-technical information. A *function type* is an abstract characterisation of the product. As an example, the configuration model of Figure 1 could be supplemented with the function type MusicCD with the semantics of being able to play music from CDs.

To be of any use, the function concept must be somehow related to the technical concepts. This is achieved through special constraints (see section 2.2.5 below), *implementation constraints*. They specify that a certain function be implemented by one or more technical concepts, e.g., component or port individuals. Returning to the example, MusicCD type could be implemented by individuals of component types CD and Speaker (not in the model) together.

Similarly as for component types, function types can be defined a compositional structure through part definitions, with the natural exception that the possible part types must be function types.

2.2.5. Constraints. *Constraints* can be used to capture aspects of products that cannot be reasonably modelled using the concepts presented above. A constraint is a formal, mathematical or logical, rule specifying a condition that must hold in a correct configuration. They can be used to specify arbitrarily complex interactions of types, individuals, and their properties. Typical conditions used in constraints include *require* and *incompatible*: the semantics of the afore-mentioned are that a certain component individual in a configuration implies that another component individual must (require) or must not (incompatible) be in the configuration.

3. Software architectures and architecture description languages

Software architecture of a system purports to describe the high-level structure of a software system. The significance of considering architecture when designing software systems is well understood. There is, however, no single, generally accepted method for describing software architecture: UML is a good candidate for such a method, but as noted e.g. in [23], it is by no means an optimal tool for documenting all aspects of architecture. Simple methods, such as referring to an existing architectural style or using box-and-line diagrams with no or vague semantics, have been recognised to be inadequate for the task [24]. Hence, there is a need for better methods.

Architecture description languages (ADLs) are a promising candidate solution for the architecture description problem. Loosely defined, ADLs are formal notations with well-defined semantics, whose primary purpose is to represent the architecture of software systems. A large number of ADLs have been proposed. ADLs have in common the concept of component, although different ADLs have different names for the same concept [10]. But in their other characteristics, ADLs differ from each other radically. Some of them address a special application domain and others are dedicated to a specific architectural style [10]. ADLs also employ different formalisms for specifying semantics, and there is variety in how rigorously the syntax and semantics are defined.

The most fundamental elements of architectural descriptions include *components*, *connectors* and their *configurations* [10,11,24].

Components represent the main computational elements and data stores of the system. Intuitively, they correspond to the boxes in the box-and-line diagrams. Clients, servers and filters are examples of components. In a working system, a component might manifest itself as an executable file or a dynamic link library. [11]

Unlike components, connectors are not loci of application specific computation in software systems. Instead, they represent interactions between components. In a box-and-line diagram, connectors are depicted as lines between the boxes. Examples of connectors include method invocation, pipes and event broadcast. [11]

Components can be connected to each other to form configurations. They are sometimes referred to as systems [11] or architectural configurations [10]. In many ADLs, components can only be connected through connectors; explicit use of connectors has even been proposed a defining characteristic of an ADL [10]. Typically, components are connected to each other through *connection points*. Different ADLs call these connection points with different names, e.g. port, role or interface.

In some ADLs, components can also have an inner structure. Such components are called *compound components* and they represent a subsystem that has an architecture of its own. With composite components it is important to be able to specify how the inner parts of the component are linked to the component itself. Usually, the linkage is defined by binding connection points of the compound component with connection points of its parts. Intuitively, binding means that the connection point of the compound component is in fact a connection point of some other component inside the compound component.

A practical concern with ADLs is the tool support available for them. Tool support is out of the scope of this paper, since the goal is to analyse the modelling languages. However, it should be noted that support for generating executable systems out of architectural descriptions is one of the goals of research on ADLs [10]. This is a goal shared by research on configuration modelling.

4. Analysis of three architecture description languages

In this section, we first define a framework for analysing and comparing the concepts of ADLs with those of configuration. Thereafter, we use the framework to study three ADLs: Acme [11,12,13], Wright [14,15] and Koala [16,17,18]. A more detailed analysis can be found in [25].

4.1. Framework for analysis and comparison

The fundamental phenomena described by the configuration ontology and that presented in [21] are: taxonomies, structure, topology, resources, functions, and constraints.

In the following three subsections, we will analyse the above-mentioned ADLs using a comparison framework composed of three parts. The first part includes the key concepts of ADLs and the configuration ontology, and the relations between them. The concepts include *components*, *connectors*, *configurations*, *connection points*, *attributes*, *resources*, *functions*, and *constraints*. The relations include *topology*, *taxonomy*, and *structure*. The second part considers the existence of different concepts for types and instances. The last part of the framework is the variation mechanisms provided by ADLs and the configuration ontology.

4.2. Acme

The basic concepts of Acme are *components*, *connectors*, and *systems*. System is the Acme term for configuration. On the other hand, there are no constructs for resources or functions in Acme. Both components and connectors have connection points that are called *ports* for components and *roles* for connectors. *Design elements* include component, connector, port, and role. Components are connected to connectors by defining an *attachment* between the port of a component and the role of a connector. One connector may connect multiple components. Components cannot be connected directly to each other and neither can a connector to another connector. [11]

Components and connectors can have attributes that are called properties in Acme. Properties are uninterpreted values, i.e. they do not have any semantics defined.

In Acme, design constraints can be defined using first order predicate logic. They can be either *invariant* or *heuristic*: invariant constraints must hold, whereas heuristic constraints are merely hints of what should be true for an Acme system. Constraints can be used to express various aspects of Acme systems: e.g. the existence and values of properties and the connections present in a system. [12]

In addition, Acme includes a structure called *representations* that can be used for describing an alternative view of a component or a connector. *Rep-maps*, or in other words, *representation maps*, can be used to specify the correspondences between different representations of a design element. There is, however, no semantics defined for either representations or rep-maps. One possible use of these constructs is representing the compositional structure of a component and the correspondences between the ports and roles of the compound component and those of the contained components. [11].

Although types are not first class entities in Acme, it has two type systems: one for design element types and, and another for systems. Types in the design element type system are sets of required structure, i.e. design element declarations, and values. New types can be formed from existing types through subtyping. System types are called *families*. A family consists of design element type definitions. Subtypes of families can be formed through single or multiple inheritance. Also, a system can be declared to be a member of many family types. [13]

What makes types a secondary concept in Acme is that design elements and systems need not have a type or be a member of a family, respectively. A design element being of a given type merely implies that the design element has the structure and values specified by that type. Similarly with families, a system being a member of a family signals that the type definitions of the family are type definitions of the system, too. Therefore, type systems of Acme can be considered a sort of macro expansion mechanism.

The syntax and semantics of Acme are formally defined, the latter in terms of a mapping to first order predicate logic.

There seem to be no constructs in Acme for modelling variety. What seems to come closest to modelling variability is the family construct. It can be used to specify a set of type definitions shared by a set of systems. Furthermore, constraints can be used to enforce the instantiation of certain design elements. Hence, the family definitions complemented with constraints seem to provide a mechanism for specifying product families with certain properties.

4.3. Wright

As in Acme, there are *components*, *connectors*, *systems*, *ports*, *roles* and *attachments* in Wright and their semantics are the same in both languages. There are no attributes, resources or functions. What distinguishes Wright from Acme and makes it special among ADLs is its way of specifying the behaviour of ports, roles, connectors and components, and the possibilities for analysis based on these specifications. Wright uses CSP (Communication Sequential Processes) [26], a formal approach for two purposes: (1) specifying processes that reside in Wright elements and (2) defining semantics of non-CSP parts of the language. CSP is a formal method for specifying and analysing the behaviour of objects in terms of sequences of events in which they engage. The pattern of events that is possible for an object is termed a *process*. [14,15]

Each port and role is associated with a CSP process. In addition, each connector and component includes a separate glue and computation process, respectively. The glue of a connector defines the operation of the connector as an entity. That is, the glue coordinates the operations of the other processes in the connector. Ports are attached to roles to form systems. Which ports can be attached to which roles, is determined by their process descriptions: a port can be attached to a role if the port will behave well in all situations enabled by the role, i.e., CSP defines a compatibility relation between ports and roles.

The second usage of CSP in Wright, defining semantics of non-CSP parts of the language, allows using tools operating on CSP to reason about properties, most notably about dead-lock freedom, of a Wright connector. This is an important class of tool support enabled by the rigorously defined semantics of Wright.

Wright allows describing hierarchical structure of both components and connectors. This is done by enclosing a system into the place of a process. In addition to the normal system specification, bindings between the port and role names in the enclosing element and those specified in the enclosed system need to be specified.

Wright distinguishes between component and connector types and instances. Each connector and component is of exactly one type. There is, however, no taxonomy of types.

In addition to component and connector types, Wright includes a construct called *style*. Styles are collections of type definitions and *constraints*. They are expressed in first order predicate logic and they can be used in a manner similar to that in Acme described above. In addition to component and connector type definitions, a style can include *interface type* definitions. They are process descriptions that can be used in port and role definitions.

Type definitions in styles can be parameterised. That is, parts of the type definition can be left open and a value can be filled in when the type is instantiated. New styles can be defined in terms of existing ones through subtyping: the new style has the same type definitions and constraints as the old one plus some additional type definition or constraints.

Variation mechanisms of Wright are similarly limited as for Acme, although Acme uses the term family where Wright uses style. In short, styles supplemented with constraints seem to be able to express variability.

4.4. Koala

As the languages described above, the Koala model has *components* as a main design element. But in other respects, Koala differs greatly from its peers. In Koala, there is no notion of connectors, resources, functions or constraints. Components are connected, or *bound*, as it is said in Koala, to each other through *interfaces* that are the connection points in Koala. The connection between components is not symmetric: a distinction is made between *provided* and *required interfaces*. Loosely defined, a component having a provided interface means that the component offers some service for other components to use. Similarly, a required interface signals a service being required by the component from some other component. Koala interfaces are similar to those in COM or Java.

Compound components can be used to express compositional structure in Koala, i.e. other components can be contained within a component. An interface of a compound component can be bound to an interface defined by a contained component. A *configuration* is defined as a component that is not contained in another component and defines no interfaces.

In addition to binding interfaces to each other, it is possible to bind constituent parts of interfaces directly. These parts are called *functions*. Hence, interfaces in Koala are not atomic even when considered as connection points.

There are some limitations on how interfaces can be connected to each other: These limitations are best illustrated with the aid of Figure 2. In the figure, components are depicted as boxes and interfaces as squares containing triangles inside them. A required interface is depicted as a triangle pointing outwards from the component, and a provided interface with a triangle pointing inwards. The binding rules are that must be bound by its tip to exactly one base of an interface, and any number of interfaces can be bound to the base of an interface. Notice that these rules cover both bindings between interfaces in contained components and interfaces in independent component, i.e., components such that neither is contained within another.

Another rule concerning the bindings is that the type of the tip interface must be a *supertype* of the type of the base interface: interface type A is a supertype of B exactly when B contains all the functions of A.

Koala has a type system: a distinction is made between both interface and component types and instances. There is, however, no taxonomy of component or interface types beyond the supertype relation mentioned above.

Koala includes a construct, *module*, which is a component without an interface of its own. Modules are used inside compound components for gluing interfaces. Suppose, for example, that each component contained in a compound component has an initialisation interface to be called before using the component. Due to binding rules, it would not be possible to bind all these interfaces to any single interface of the compound component. Therefore, a new configuration specific module is added: when the initialisation function for the compound component is called, the call is routed to the module, which in turn calls the initialisation functions of all necessary components in the order desired.

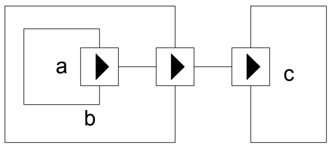


Figure 2 Koala components and interfaces

In addition to the constructs already mentioned, Koala provides mechanisms for handling both the *internal diversity* of components and the *structural diversity* in a configuration. Internal variety is manifested as variation of component parameters. There may be dependencies between parameters: a parameter value may imply that another parameter has a certain value. Structural diversity pertains to alternative provided interfaces for a required interface: e.g. there may be multiple components that provide the same interface required by a certain component. The choice between the interfaces is made by a construct called *switch* either statically, that is at compile time, if the information required for the selection is available, or, otherwise, dynamically at runtime.

5. Comparison of concepts of the adls with the configuration ontology

In this section, we use our framework defined in the previous section for comparing the concepts and constructs found in the ADLs with those of the configuration ontology.

5.1. Key concepts and relations between them

Component is the central concept of Acme, Wright and Koala. It is also present in the configuration ontology with that same name. The semantics are as well similar: components represent the defining parts of a system in configuration modelling, too. In addition, systems as defined in Acme and Wright and configurations as defined in Koala have a counterpart in the configuration ontology, namely configuration.

The notion of connection points is also common to all the studied modelling methods. In Acme and Wright they are called ports and roles in components and connectors, respectively. In Koala connection points are termed interfaces and in the ontology ports. The semantics of connection points are also similar in all the disciplines: they denote the mechanism for connecting other entities.

Connectors are first-class citizens in Acme and Wright., but there are no connectors in the configuration ontology. In Koala, modules can be considered as a form of connectors. Thus, there is a major difference in how the disciplines handle architectural connection – an important issue in both ADLs and in the configuration ontology.

What then is the reason for this disagreement in architectural connection? We believe that at least a partial reason for the importance of connectors in Acme and Wright can be found in the underlying assumptions of them and several of the ADLs not studied in this paper: a major issue in software architecture has been reusing existing components. Furthermore, there has been considerable effort in the software engineering community to reuse heterogeneous components, which cannot be connected directly to each other due to different communication mechanisms and various other reasons. Therefore, connectors have been introduced in ADLs as a vehicle for connecting heterogeneous components.

In Koala, the situation is rather different: components are generally rather homogenous, and it seems that there is no need to require that connectors be used whenever connecting components: modules are used as needed. Hence, in this respect, Koala is closer to the configuration ontology than Wright and Koala.

Resources, a feature present in the configuration ontology but not in any of the ADLs, is similar to the notion of provided and required interfaces present in Koala in the sense that they are both anti-symmetric. What is more, resources are produced and consumed by components, just as interfaces are provided and required. However, resources are produced and consumed in certain quantities, which gives them more expressive power compared with the notion of provided and required interfaces.

In addition to simulating provided and required interfaces, resources can be used to model other relevant quantities. Such quantities include memory, power, output capacity and throughput. The software engineering community has considered similar issues important [27]. Hence, resources could very well be an important feature of the configuration ontology when used to model software architecture.

Modelling functions is another feature of the configuration ontology that all three ADLs presented in this paper lack. Functions are an important aspect of software engineering usually termed features in the domain [28]. We believe that also functions could be very useful when modelling software with the ontology.

All the ADLs have some mechanisms for modelling structure. However, the configuration ontology provides much stronger mechanisms: the configuration ontology provides a wide range of variation mechanisms. Furthermore, in the configuration ontology a component can be a part of many components simultaneously, which is not possible in any of the ADLs.

All the disciplines except Koala have explicit mechanisms for expressing constraints. Further, in all disciplines where constraints exist, they are logical expressions about the non-behavioural properties of a system modelled in that discipline. A difference is that in the configuration ontology, there is no direct support for heuristic constraints as defined in Acme. Support for modelling preferences and optimisation criteria have been identified as important and developed in other research on configuration.

5.2. Distinction between types and instances

All the three ADLs have some distinction between types and instances. In Acme, the distinction is rather weak, as the type systems can be seen as a simple macro expansion mechanism. Nevertheless, there is taxonomy between the Acme types. The situation is rather similar in Wright: types bear a little meaning as such. The only function of types seems to be facilitating in defining and altering recurring patterns. In Koala, the supertype relation constraints bindings between interfaces. This relation is not, however, declared explicitly, but implicitly based on interface types. The component types seem to have no function beyond defining the structure of a set of components. Hence, component types seem to be as a construct as weak as types in Acme and Wright.

In the configuration ontology, strong distinction between types and instances is one of the basic assumptions and is made for all kinds of entities. Types are organised in taxonomies.

5.3. Variation mechanisms

A question closely related with the distinction of types and instances is: What is being modelled, one product or a product family. The configuration ontology aims at modelling product families. Configuration model knowledge defines the common properties of the family members. A lot of variation mechanisms are provided.

As stated in the analysis of Acme and Wright, both of these languages can be seen to provide some support for modelling variability: there are no explicit variation mechanisms, but the combination of system types and constraints seem to be able to express common structure shared by a set of products.

In Koala, there is some knowledge about the common properties of all the products: component and interfaces definitions are stored in a component repository and they are common to different systems to be constructed [17]. In fact, type definitions shared by a set of products is exactly the same phenomenon be have already seen in family construct Acme and in the style construct in Wright. As there are no constraints to complement the shared type definition in Koala, the support provided by Koala for variability is weaker than that Acme and Wright.

In the previous section, it was stated that Koala could model both internal and structural variety. How does this statement relate to the above observation that Koala provides a weaker support for variability than Acme and Wright? We claim that we are dealing with two distinct forms of variability. The variability in Acme and Wright can be used to span a set of products with many similarities, or a product family. On the other hand, the variation mechanisms in Koala seem to model behavioural variety of software embedded in a physical product instance: e.g. a television set can behave differently depending on some parameters. Of course, it could be argued that the television set in our example is, in fact, a product family. However, we consider the variation mechanisms discussed above examples of different phenomena.

6. Modelling software architecture with the configuration ontology

In this section, we strive to synthesise the configuration ontology with the domain of software architecture. We do this by mapping the concepts in the ADLs to some concept or concepts in the configuration ontology. Components, ports, properties, and constraints are represented in the obvious manner using their direct counterparts, whereas the representation of connectors and roles is more problematic. Hence, we will present a mapping of connectors to components, and provided and required interfaces to ports with the aid of type specifications.

Due to limited space, we will only present the main ideas of the mapping. For full details, please refer to [25].

6.1. Modelling connectors as a type of component

In translating the semantics of connectors in Acme and Wright into concepts in the configuration ontology, it helps to observe that components and connectors have structures very similar to each other. Therefore, it is natural to view connector as a subtype of component with special semantic constraints. Indeed, defining connector to be a subtype of component will enable us to express part of the semantics associated with connectors. Furthermore, we can define roles in connectors to be ports in the connector-type components. To enforce the right use of connectors, we define suitable constraints that enforce the right use of connectors: e.g. in Wright, the only class of allowed connections is that between a component and a connector.

Subtyping can also be used for distinguishing provided and required interfaces from one other. By defining common supertypes for provided and required interfaces it is possible through multiple inheritance to have two versions of each port type, a provided and a required. By using constraints it is possible to assert that invariants concerning provided and required interface types hold. For instance, the fact that in Koala a required interface must be connected to exactly one provided interface of the same interface type can be easily captured using constraints.

6.2. Capturing diversity

Internal diversity of Koala can be captured with attributes defined by components and constraints. Dependencies between different parameters can be captured using constraints between attribute values of component types.

In the configuration ontology, cardinality of a port defines the amount of ports that can be connected to it. Cardinality can be used to capture some aspects of structural diversity in Koala. By defining cardinality greater than one for a port representing a required interface, multiple provided interfaces represented as ports could be connected to that port. This is only a partial solution as it says nothing about deciding which ports should actually be connected; constraints can be used to model this.

7. Extensions needed for modelling software architecture

Albeit the ontology captures a major part of aspects of all the studied ADLs, each of them has some features the modelling of which would require extending the ontology.

Capturing all of the idea behind **heuristic constraints** of Acme may require adding some method of representing optimisation criteria and preferences in the ontology.

There is no mechanism in the configuration ontology for **modelling behaviour** similar to the way how CSP is used in Wright. In fact, the configuration ontology ignores behavioural aspects entirely. In case considering behaviour should be required in the configuration ontology, it would be natural to extend the constraint language to cover behavioural aspects, as the constraint language can be seen as the extension mechanism of the ontology.

Koala includes the method of **function binding**, in which the constituent functions of interfaces are connected directly to each other instead of connecting interfaces [17]. This construct gives an internal structure to Koala interfaces. Given that interfaces of Koala are modelled with ports in the configuration ontology, this contradicts with the underlying assumption of ports being undividable connection points. As a result, there is a mismatch between interfaces in Koala and ports in the configuration ontology.

There is a number of possible ways to capture ports with internal structure. The first one is to make Koala functions the basic level of connection. Unfortunately, this approach introduces major problems. Firstly, interfaces would lose their counterpart in the configuration ontology. Secondly, following the approach would likely lead to increased complexity in models of software products: that an interface can contain several functions implies this.

The second approach would be to introduce compositional structure for ports of the configuration ontology. Applied to the problem at hand, interface types correspond to port types that have ports corresponding to functions as their parts. This approach is appealing: it models the relation between interfaces and functions in a way corresponding to the intuitive understanding of the issue. This approach would require major changes to the ontology.

Binding of interfaces of a compound component with the interfaces of the inner parts is another feature of Koala lacking a counterpart in the ontology. It seems that the ontology would need to be extended in order for it to model this phenomenon.

8. Discussion, comparison to previous work

There is an apparent difference in the natures of the sets of product variations modelled in different disciplines. In the configuration domain, this set is typically termed as configurable product or a product family. One of the defining characteristics of this concept is a pre-designed general structure with a lot of variation in the configurations [29]. On the other hand, it was found that Koala supports no common structure for a set of products. In fact, Koala is not targeted at modelling a product family or a set of them, but product populations, defined as a sets of products with many commonalities but also with many differences [16]. Hence, the underlying aims of the ADL modelling and configuration modelling are not totally similar.

In the previous section, it was stated that no satisfactory mapping could be found for function binding of Koala. One possibility to respond to this and similar problems is to ignore the problematic feature. Even though we do not light-heartedly ignore aspects of ADL that are of practical or theoretical importance, we still believe that doing so in some cases will increase the usefulness of the configuration ontology in modelling software products. Therefore, the question is: which features of ADLs should be modelled. This question can only be fully answered by empirically studying software product families.

Research closely related to this paper has been conducted earlier. We do not know of earlier attempts of comparing the concepts of software architecture description to those of configuration modelling. This is the main contribution of our paper: studying the feasibility of configuration techniques to software variability management.

In their work, Männistö et al. have pointed out the existence of the research area of configurable software and identified some key concerns in the area [30]. They have not, however, studied the concepts of ADLs in detail or proposed any mapping from these concepts to those of configuration modelling domain.

On the other hand, [31] presents a formalised software configuration management (SCM) ontology. The concepts of the SCM ontology are, however, different from those of the configuration ontology. They are aimed at representing the modules, files, or packages, their versions and the dependencies between these. The ontology does not take into account the connections and interfaces between components of a system.

Felfernig et al. have proposed a scheme for constructing configurators based on UML descriptions of configuration knowledge [21]. Their approach could be used for creating configurators for software products as well. Their approach is, however, different from our approach: theirs is based on presenting configuration knowledge in UML, while our approach is based on modelling software with the concepts of product configuration.

In [32], Kühn has presented an approach to software configuration based on structure and behaviour. He uses statecharts, a method similar to finite state machines, for specifying the behaviour of a module. This approach is similar to Wright in that it describes both structure and behaviour. With its focus on using behavioural constraints for making decisions during the configuration process, this approach is different from ours.

Feature models have been suggested as a modelling method for software product lines (see, e.g., [33]). Apparently, feature models share much with the configuration modelling concepts presented in this paper: features, and both components and functions in the configuration ontology are organised in composition hierarchies. What seems to be different in the two approaches is that the configuration ontology distinguishes between technical and non-technical aspects of a product, whereas in features models, both aspects are contained in the same hierarchy.

Lars Geyer et al. have identified the need and enumerated a number of requirements for a configuration technique for software product families [34]. Of the requirements, the configuration ontology supports hierarchical structures for both technical and non-technical entities, i.e., components and functions, respectively. Further, the ontology incorporates a constraint mechanism. Finally, a prototype tool for configuring mechanical products that supports a subset of the configuration concepts presented in this paper corresponding to the requirements posed by Geyer et al., visualises the aggregation hierarchy and is able to assist in configuring the product, as required by them [35]. Therefore, although Geyer et al. have deemed knowledge-based configuration techniques rather useless in the context of software product families, we feel that knowledge-based configuration techniques provide considerable potential in domain of software product families.

9. Conclusions and further work

We have presented an analysis and comparison of three ADLs and a conceptualisation of configuration knowledge. We defined a mapping from the concepts of ADLs to those of the conceptualisation of configuration knowledge. Our goal is to use the configuration concepts and their supporting tools for configuring software product families.

We found counterparts and close correspondences in the configuration ontology for the main elements of the ADLs we have studied and were able to propose a mapping between them that shows that configuration languages can be used for representing architectural knowledge. For instance, both share the notion of components. Furthermore, compositional structure, systems formed of connected components and constraints are phenomena present in both disciplines. Hence, it seems that the concepts of the configuration ontology can be used for modelling software products.

Capturing some aspects of ADLs seems to require extending the configuration ontology. These aspects include function binding and binding the connection points of compound components with connection points in its inner parts. Another important aspect is modelling behaviour. Of the ADLs, Wright models behaviour. Additionally, the approach presented by Kühn also emphasises behaviour [32]. The question whether behavioural aspects really are important and should be modelled when configuring software product families, should be resolved through empiric studies with real products. The existence of Koala, a commercial ADL with no behaviour modelling, suggests that modelling behaviour is not absolutely necessary.

There are still open questions and a need for further work. An ontology and a configuration language for software products should be defined, and a configurator supporting this language constructed. This work is currently in progress. This will probably require investigating more thoroughly the current ADLs and the conceptualisations of disciplines such as SCM, generative and feature based programming [28,36], and, of course, the developments in the UML community, as well as case studies of real software product families. After completing this, case studies are needed to verify the applicability of the configuration language to modelling software. Another issue to be concerned is the computational complexity of configuring software products. Theoretical complexity analysis can provide insight into this issue, but only experiments with real products will give relevant information on the practical feasibility from this point of view. When moving towards empirical studies, it is also necessary to consider which of the existing configurators and their modelling languages best support software configuration at a more detailed level than in this study.

Finally, the economics of our approach should be studied. Particularly for simpler products, the overhead from developing a configuration model of a software product family using a configurator is higher than the advantages that can be gained. However, we believe that there are cases where the product family is complex enough, including even thousands of variation points, that the support for deployment process would outweigh the costs.

Acknowledgements

We gratefully acknowledge the financial support from Academy of Finland (project 51394) and National Technology Agency of Finland (Tekes).

References

[1] J. Bosch, *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[2] P. C. Clements and L. Northrop, *Software Product Lines - Practices and Patterns*, Addison-Wesley, 2001.

[3] T. Soininen, *An approach to knowledge representation and reasoning for product configuration tasks*. PhD thesis, Helsinki University of Technology, 2000.

[4] B. Faltings and E. C. Freuder, eds., Special Issue on Configuration. *IEEE Intelligent Systems* 13(4), 1998.

[5] T. Darr, M. Klein, and D. L. McGuinness, eds., Special Issue on Configuration Design. *AI EDAM* 12(4), 1998.

[6] D. Mailharro, 'A Classification and Constraint-Based Framework for Configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4), 383-397, 1998.

[7] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, 'Configuring Large Systems Using Generative Constraint Satisfaction', *IEEE Intelligent Systems*, 13(4), 59-68, 1998.

[8] B. Yu and J. Skovgaard, 'A Configuration Tool to Increase Product Competitiveness', *IEEE Intelligent Systems*, 13(4), 34-41, 1998.

[9] S. Vestal, A Cursory Overview and Comparison of Four Architecture Description Languages. Technical report, Honeywell Systems & Research Center, 1993

[10] N. Medvidovic and R. M. Taylor, 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Transactions on Software Engineering*, 26(1), 70-93, 2000.

[11] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: An Architecture Description Interchange Language', in: *Proceedings of CASCON'97*, 1997.

[12] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: Architectural Description of Component-Based Systems', in: *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, eds. Cambridge University Press, 47-68, 2000.

[13] R. T. Monroe, D. Garlan, and D. Wile. Acme Reference Manual. Available at http://www-2.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/ACME%20StrawManual.html, 2002. Cited January 10th, 2003.

[14] R. Allen and D. Garlan, 'A Formal Basis for Architectural Connection', *ACM Transactions on Software Engineering and Methodology*, 6(3), 213-249, 1997).

[15] R. Allen, *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[16] R. van Ommering, 'Configuration Management in Component Based Product Populations', in: *Proceedings of Tenth Intl Workshop on Software Configuration Management (SCM-10)*, 2001.

[17] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, 'The Koala Component Model for Consumer Electronics Software', *IEEE Computer*, 33(3), 78-85, 2000.

[18] R. van Ommering, 'Building Product Populations with Software Components', in: *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 255-265, 2002.

[19] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM*, 12(4), 357-372, 1998.

[20] L. Ardissono, A. Felfernig, G. Friedrich, et al, 'Customer-Adaptive and distributed online product configuration in the CAWICOMS project', in: *Proceedings of IJCAI-01 Workshop on Configuration*, 2001.

[21] A. Felfernig, G. Friedrich, and D. Jannach, 'UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems', *International Journal of Software Engineering and Knowledge Engineering*, 10(4), 449-469, 2000.

[22] J. Tiihonen and T. Soininen, *Product configurators - information system support for configurable products*. Technical report TKO-B137, Helsinki University of Technology, 1997

[23] P.Clements, F.Bachmann, L.Bass, et al, Documenting Software Architecture, Addison Wesley, 2002.

[24] D. Garlan, 'Software Architecture', in: *Encyclopedia of Software Engineering*, J. J. Marciniak, ed. John Wiley & Sons, 2001.

[25] T. Asikainen, *Representing Software Product Line Architectures Using a Configuration Ontology*. Master's thesis, Helsinki University of Technology, 2002.

[26] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[27] D. Garlan and D. E. Perry, 'Introduction to the Special Issue on Software Architecture', *IEEE Transactions on Software Engineering*, 21(4), 1995.

[28] K. Czarnecki and U. W. Eisenecker, *Generative Programming*, Addison-Wesley, 2000.

[29] J. Tiihonen, T. Lehtonen, T. Soininen, et al, 'Modeling Configurable Product Families', in: *Proceedings of the 12th International Conference on Engineering Design (ICED'99)*, U. Lindemann, H. Birkhofer, H. Meer-kamm and S. Vajna, eds. 1139-1142, 1998.

[30] T. Männistö, T. Soininen, and R. Sulonen, 'Product Configuration View to Software Product Families', in: *Proceedings of the Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

[31] T. Syrjänen, 'Including Diagnostic Information in Configuration Models', in: *Proceedings of the First International Conference on Computational Logic*, 2000.

[32] K. Kühn, 'Modeling Structure and Behavior for Knowledge-Based Software Configuration', in: *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling, and Design*, J.Sauer and J. Köhler, eds., 2000.

[33] K. Kang, J. Lee, and P. Donohoe, 'Feature-oriented Product Line Engineering', *IEEE Software*, 19(4), 58-65, 2003.

[34] L. Geyer and M. Becker, 'On the Influence of Variabilities on the Application-Engineering Process of a Product Family', in: *Proceedings of the Second International Conference on Software Product Lines, SPLC2.*, 2002.

[35] J.Tiihonen, T.Soininen, I.Niemelä, and R.Sulonen, "Empirical Testing of a Weight Constraint Rule Based Configurator", in: *Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, 2002.

[36] C. Prehofer, 'Feature-Oriented Programming: A Fresh Look at Objects', in: Proceedings of ECOOP'97, 1997.

Product Line Derivation with UML [1]

Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Tewfik.Ziadi, Jean-Marc.Jezequel, frederic.fondement}@irisa.fr

Abstract

Handling the various derivations of a product can be a daunting (and costly) task. To tackle this problem, we propose a method based on the use of a creational design pattern to uncouple the variations from the selection process. This makes it possible to automatically derive a given product from the set of all possible ones, and to specialize its model accordingly. The contribution of this paper is to provide a set of patterns for modeling variability issues of a Product Line Architecture to define architectural constraints for Product Line expressed in UML as meta-level OCL constraints and to propose an approach to automate the derivation process.

1. Introduction

Software Product Line (SPL) captures "commonality" and “variability” between a set of software products in the same domain. Commonality designates elements that are common to all products while variability designates elements that may vary from a product to another one.

Software Product Line engineering aims at improving productivity and decrease realization times by gathering the analysis, design and implementation activities of a family of systems. It is based on the reuse of assets instead of working from scratch. A Software Product Line Architecture also called a reference architecture is a generic architecture from which the model of each product can be derived. The role of software product line architecture is to describe commonalities and variabilities of the products contained in the Product Line (PL) and, as such, to provide a common overall structure.

To model SPL with the UML (Unified Modeling Language) [19], we need mechanisms to specify variabilities and commonalities, and techniques to derive products. We also need to manage a set of constraints that specify variation point dependencies in the PL.

This work focuses on the PL derivation activity and proposes an approach based on a creational design pattern to derive product models from a PL architecture modeled by the UML. The derivation process should preserve PL coherence, so we have defined and specified a set of PL constraints as OCL (Object Constraint Language) meta-model constraints. To illustrate our approach, we use a Mercure PL.

The paper is organized as follows: Section 2 briefly presents the Software Product Line Engineering approach and the Mercure PL. In section 3, we propose some mechanisms to specify variability in the UML class diagrams. Section 4 presents PL constraints and their specification with the OCL, and the section 5 illustrates the derivation process. Finally section 6 concludes this work.

2. Background in Product Line Engineering

2.1. The Software Product Line approach

The general process of Product Line Engineering, as found in the literature [4,5,18], is illustrated in the figure 1. We distinguish two main activities:

Domain Engineering. The domain engineering activity is twofold:

- Collecting, organizing, and storing past experiences in building systems in the form of reusable assets (i.e. reusable work products) in a particular domain,
- providing an adequate means for reusing these assets when building new systems [4].

The term *Developing for reuse* is often used to characterize the Domain Engineering. It can be divided in three main processes: *Domain Analysis, Domain Design,* and *Domain Implementation*. The domain analysis consists in capturing information and organizing it as a model. Some methods, such as FODA (Feature-Oriented Domain Analysis) [13] propose a set of notations for the domain modeling using the notion of "features" to refer to products properties. The domain design consists in establishing the product line architecture. The domain implementation consists in implementing the architecture defined during the domain design as software components.

Application Engineering. The application engineering activity consists in building systems based on the results

[1] This work has been partially supported by the CAFE European project. Eureka Σ! 2023 Programme, ITEA project ip 0004

of Domain Engineering. During application requirements of a new system, we select the requirements from the existing domain model, which matches the customer's needs. We assemble applications from the existing reusable components. The term *Developing by reuse* is used to characterize the application engineering activity.

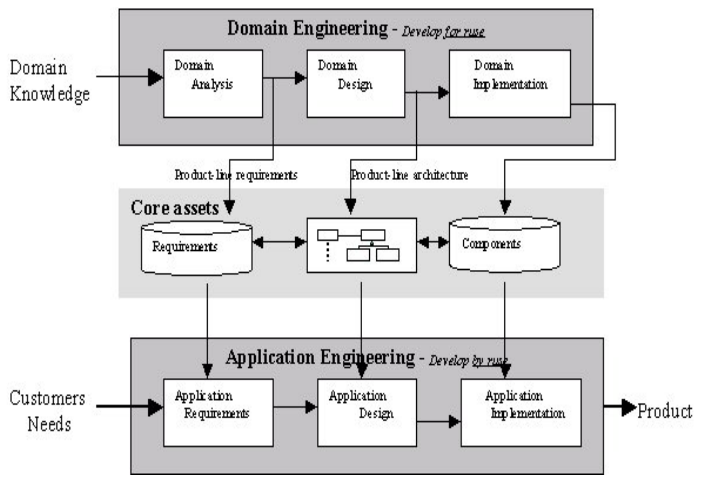


Figure 1. The general process for Product Lines Engineering

2.2. The Software Variability Management

The main challenge in the context of software product lines is to model and implement the variability. Even if the product line approach is a new paradigm, managing variability in software systems is not a new problem, and it can be solved by some existing approaches. [14,16] study how existing techniques can be used for the variability management. We briefly list some of these techniques:

Compilation techniques: it is used to derive products at the compilation time by the inclusion or the exclusion of code segments during program compilation. For example, the conditional compilation can be used to manage variability at the compilation time.

Programming languages properties: Object Oriented Languages offer some techniques such as inheritance, overloading, and dynamic binding that can be used to implement variability. Variation points are defined as abstract properties in the Product Line and each product defines these points in a specific way. Variability can also be implemented using class templates if the variants differ by a set of parameters.

Design patterns: Design Patterns [8] can be used to model variability in software product line architectures. Patterns provide reusable solutions to certain types of problems and support the reuse of underlying implementations. In [12], the Abstract Factory pattern is proposed for reifying variants (we will present in more detail this solution in section 5). [2] proposes a set of patterns to model variability in product line architectures based on the notion of "Discriminants".

Programming approaches: some recent approaches of Software Engineering can be used for the variability management. Aspect-Oriented paradigm [6] is an engineering principle that aims at reducing systems complexity: it decomposes problems into a set of functional components and a set of aspects that crosscut functional components. Then it composes these components and aspects to obtain a system implementation. Some work [9,14,17] say that this approach can be used to implement variability. Aspects can be viewed as variation points, and product line members are specified by the aspects they contain. Generative Programming [4] is a software engineering paradigm based on the notion of "generator" for system families. Viability in Product Line can be managed by implementing components and generators as generic artifacts. A specific instantiation can be used to generate the implementation of a product.

The techniques presented above are generally related to programming languages. We also find some work [3,5,15] about the modeling of variability in the UML. These work mainly are based on the UML extensions mechanisms such as stereotypes and tagged values. We will present in the next section mechanisms that we have used to specify variability in UML class diagrams.

2.3. The Variability in the Mercure PL

As a case study for evaluating our approach, we consider the Mercure PL, which is a family of SMDS (Switched Multi-Megabits Data Service) servers whose design and implementation have been described in [10,11]. It can abstractly be described as a communication software delivering, forwarding, and relaying "messages" from and to a set of network interfaces connected into an heterogeneous distributed system.

Mercure PL must handle variants for five variation points: any number of specialized processors (Engines), network interface boards (NetDriver), levels of functionality (Manager), user interface (GUI) and support for languages (Language). To identify variabilities in the Mercure PL, we specify its domain model using FODA notations, slightly modified and extended by [4]. We use a set of feature kinds to specify variability and commonality:

Mandatory features: to specify features that are common to all products, we use mandatory features whose ancestors are also mandatory. Mandatory features are shown in the FODA diagram by nodes with black circles.

Optional features: it represents features that can be omitted in some products; it is shown by nodes with an empty circle.
Or-features: a feature may have one or more sets of direct or-features. If the parent of a set of or-features is included in the description of a specific product, then any nonempty subset from the set of or-features is included. The nodes of a set of or-features are pointed to by edges connected by a filled arc.

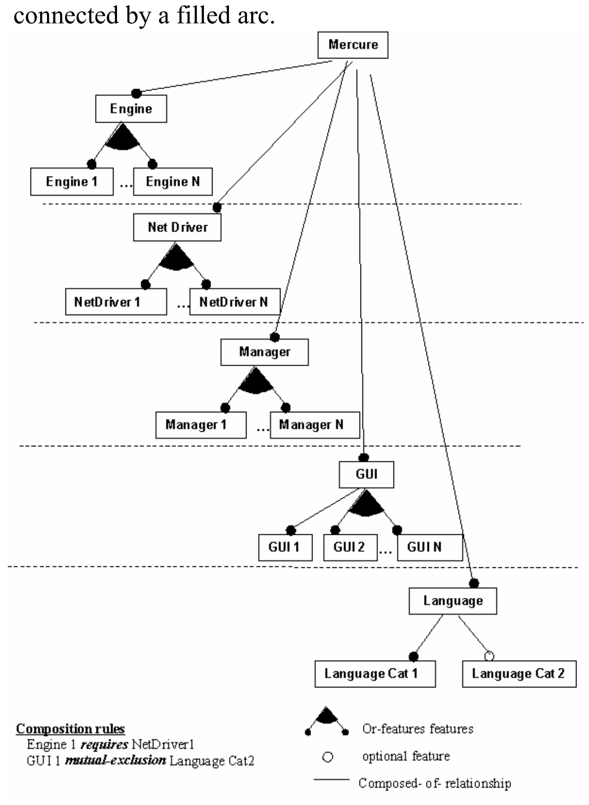


Figure 2. The FODA diagram for the Mercure PL

Figure 2. shows a feature diagram of the Mercure PL. The Mercure consists of Engine, Net Driver, Manager, GUI, and Language; all these features are mandatory. The Mercure product may support one or more of Engine 1,..Engine N, we use FODA or-features to represent it. In the same way, we define all NetDrivers and Managers dimensions. However all Mercure products should support one GUI, which is GUI 1, so it is defined mandatory. Other GUIs are defined as FODA or-features. We distinguish two categories of languages: Language Cat1 and Language Cat2, all products should support the first one and the second one is optional.
The FODA notations allow us to specify dependencies relationships, called "composition rules", between domain features. FODA supports two types of composition rules: the *requires rule* that expresses the presence implication of two features, and the *mutually-exclusive rule* that captures the mutual exclusion constraint on feature combinations. Two rules are identified in the context of the Mercure PL: a *requires rule* is added between the Engine 1 and the Net Driver 1 while a *mutual-exclusion* rule is added to specify that GUI 1 do not supports Language Cat 2 (see figure 2.)

3. Variability in UML class diagrams

The Unified Modeling Language (UML) [19] is a standard language for the object-oriented analysis and design. It defines a set of notations to describe different aspects of systems. In this section, we present three mechanisms that can be used to specify the variability in the UML class diagram: *Abstraction*, *Parameterization*, and *Optionality*.
Abstraction: Using an object-oriented analysis and design approach, it is natural to model the commonalities between the variants of a variation point in an abstract class (or interface), and expressing the differences in concrete subclasses (each variant implements the interface in its own way).
Parameterization: the UML classes can be defined as generic assets with a set of parameters; each product binds these parameters in a specific way. UML class templates can be used as parameterization classes.
Optionality: the Product Line model includes all elements associated to all products, so in specific products some of these elements called "optional" can be omitted. To show optionality, we use an ad-hoc stereotype «optional» that can be applied to classes, packages, and interfaces.

The UML class diagram in the figure 3 represents the Mercure PL model. It basically says that a Mercure system is an instance of the MERCURE class, aggregating an ENGINE (that encapsulate the work that Mercure has to do on a particular processor of the target distributed system), a collection of NETDRIVERS, a collection of MANAGERS (that represent the range of functionalities available), and the GUI that encapsulates the user preference variability factor. A GUI has itself a collection of supported languages, which are classified into two categories.
A UML class model of a specific derived product of Mercure can include an optional number of Engines, Network Drivers, Managers, GUIs, and Languages; so these features are defined as abstract classes (Abstraction variability mechanism) and we specify variants as concrete subclasses with the optional stereotype. All Mercure products should at least support one mandatory language (LANGUAGE1-1), and one GUI (GUI1), so these subclasses are defined without the optional stereotype.

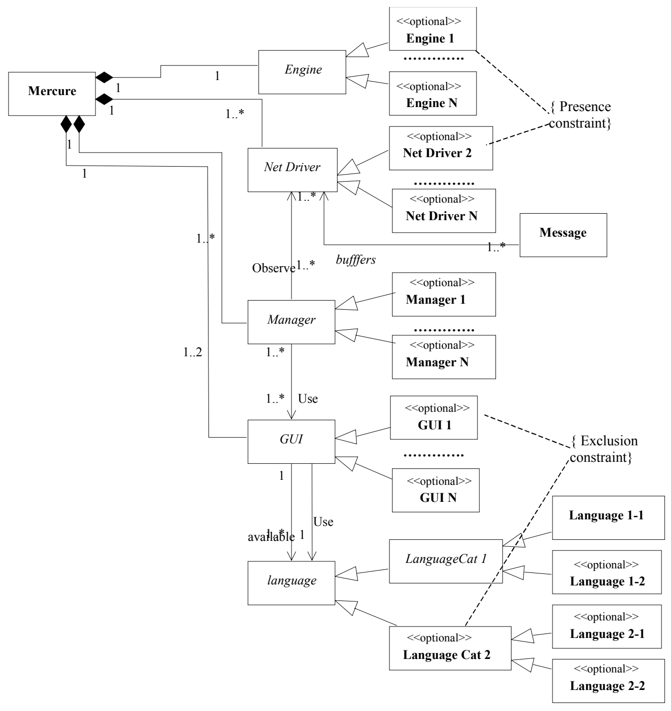


Figure 3. The Mercure Product Line UML class diagram

Defining variation points as abstract classes and each possible variant as subclass with the optional stereotype is what we call the “abstraction variability pattern”.

4. Managing the PL constraints

[1] considers that constraints are parts of PL architectures. Constraints define coherence rules and relationships between elements in the architecture. As shown previously, FODA composition rules allow us to specify relationships between domain features. Using UML, some work such as [15] use UML stereotypes to show dependencies between classes.
The Object Constraints Language (OCL) [23] allows us to attach constraints to UML models. These constraints can be defined at meta-model level as well as model level. In the context of Product Lines, we have identified two types of constraints: generic constraints applying to any PL, and specific constraints associated to a specific Product Line and we propose to define them as OCL meta-model constraints.

4.1 The Generic Constraints

The introduction of variation points, especially the optionality (specified by the «optional» stereotype), in the PL model allows us to improve genericity but it can generate some incoherence. For example, if a non-optional element depends on an optional one, we risk deriving an incomplete product model. So the first type of product line constraints defines structural properties of any product line model to preserve its coherence. UML can be extended by defining a set of stereotypes and a set of meta-level constraints that are often related to these stereotypes. So the idea for defining generic constraints is to associate a set of constraints to the relevant stereotypes, this solution was already used in [7] to define design pattern occurrences in the UML. These constraints are represented as OCL meta-model level constraints and they will be evaluated on any product line model, see figure 4.
The generic constraints may be seen as well-formedness rules for the UML modeled product lines.

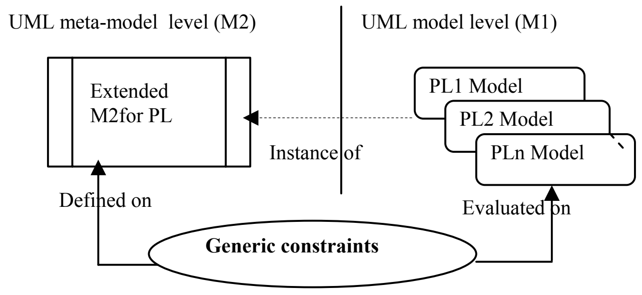


Figure 4. Generic constraints as OCL meta-level constraints

Examples of the generic constraints
Generic constraints aim to preserve the PL model coherence. In the case of the static model represented by the UML class diagram, we have defined the dependency and the inheritance constraints:
The dependency constraint. A dependency in the UML specify a require relationship between two or more elements. It is represented in the UML meta-model [19 p 2.15] by the meta-class *Dependency* (see appendix), it represents the relationship between a set of suppliers and clients. An example of the UML Dependency is the "Usage", which appears when a package uses another one. If a non-optional element is depending on an optional one, there’s incoherence in the model. To specify this rule, we add the following constraint as an invariant to the *Dependency* meta-class in the UML meta-model [19 p 2.15], where *isStereotyped(S)* is an auxiliary

primitive indicating if an element is stereotyped by a string S (see appendix):

```
context Foundation::Core::Dependency
-- For each Dependency: if the supplier is
optional the client should be  optional too
inv:
  self.supplier→ exists(S:ModelElement |
         S.isStereotyped ('optional')) implies
  self.client → forAll( C:ModelElement |
         C.isStereotyped('optional') )
```

The inheritance constraint. Optional classes in Product Line model can be omitted in some products then, if a non-optional class inherits from an optional one, perhaps there is incoherence in the product model. However, in some cases, in particular when the product line model includes the multiple inheritance, it can be correct. But it is more advisable to generate a warning if the static model includes a non-optional class which inherits from an optional one. The inheritance is represented in the UML by the meta-class *Generalization* [19 p 2.14] (see appendix). The inheritance constraint is added as an invariant to the *Generalization* meta-class:

```
context Foundation::Core::Generalization
-- For each generalization: if the parent is
optional the child should be optional too
inv:
  self.parent.isStereotyped ('optional') implies
  self.child.isStereotyped('optional')
```

Applying this to the Mercure PL model, LANGUAGE2-1 and LANGUAGE2-2 classes appear to be defined as optional because their parent (LANGUAGE_CAT2) is optional and there is not a multiple inheritance.

4.2 The Specific Constraints

A fundamental characteristic of product lines is that not all elements are compatible. That is, the selection of one element may disable (or enable) the selection of others. The set of constraints that define variation points dependencies in the specific product line are called "Specific Constraints". As generic constraints, we propose to specify specific constraints as OCL meta-level constraints. The aim of these constraints is to add dependency relationships between model elements, they are associated to a specific product line and will be evaluated on all products, derived from this PL, see figure 5.
The specific constraints are parts of the PL model definition.

Examples of specific constraints
A PL class diagram is defined to be as generic as possible and it should include elements related to all products. We have defined the presence and the mutual exclusion constraint as examples of specific constraints and we propose to define them as *Model* meta-class invariants [19 p 2.189]. A Model is a namespace that contains a set of *ModelElement* whose names designate a unique element within the namespace.

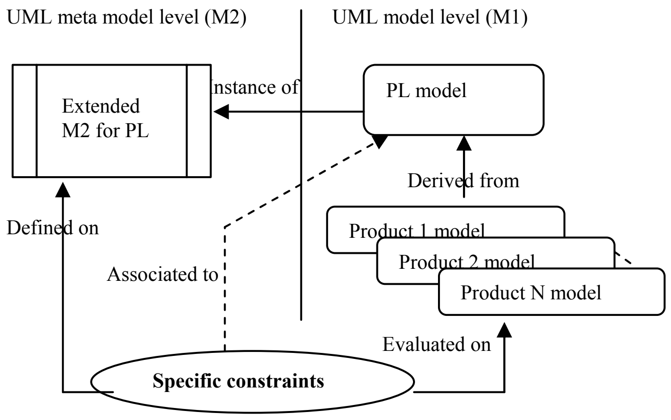


Figure 5. Specific constraints for PL model as OCL meta-level constraints

The presence constraint. This constraint is close to the *requires rule* in FODA, it expresses in a specific PL model that the presence of an optional class requires the presence of another optional class. To specify a require relationship between ENGINE1 and NETDRIVER2 classes in the class diagram of the Mercure PL, we add the following OCL meta-model constraint as a Model meta-class invariant, where the *presenceClass(C)* is an auxiliary operation indicating if a specific class called C is present in the namespace (see appendix):

```
context Model_Management::Model
--The presence in the model of the class called
'ENGINE1' requires the presence in the same
model of the class called 'NETDRIVER2'
inv:
 self.presenceClass('ENGINE1') implies
 self.presenceClass('NETDRIVER2')
```

The mutual exclusion constraint. This constraint expresses in a specific PL model that two optional classes cannot be present in the same product. As shown previously, GUI1 does not support LANGUGE_CAT2, so the mutual exclusion constraint between their associated UML classes is added as an invariant to the Model meta-class:

```
context Model_Management::Model
-- A class called GUI1 and a class called
LANGUGE_CAT2 cannot be present in the same model
inv:
(self.presenceClass('GUI1') implies not
self.presenceClass('LANGUGE_CAT2'))and
(self.presenceClass('LANGUGE_CAT2') implies not
self.presenceClass('GUI1'))
```

In the UML class diagram (see figure 3.), we use graphical shorthands to show the above constraints.

5. From the Product Line to Products

Once we have analyzed the Product Line and produced the corresponding UML Model, enriched with constraints, we still need to handle the various derivations of products. The PL derivation consists in generating from the PL model the UML class diagram of each product. As shown previously, the PL model is defined by a set of variation points and to derive a specific product model, some decisions (or choices) associated to these variation points are needed. For example, each Mercure product model should choice among the presence or non-presence of all optional classes. So another challenge in the context of PL engineering is to specify a “decision model”.
A decision model represents the set of relevant decisions and their impacts that are needed to identify one single product of the product line [5]. In this section, we propose to use the design pattern *abstract factory* as a model decision and we propose an algorithm for the product model derivation.
To illustrate the derivation process, we have defined three products of the Mercure PL:
FullMercure: it is the product that includes all optional elements. Thus, all combinations can be dynamically bound.
CustomMercure: it is a restricted product that supports only two different network drivers (NETDRIVER1 and NETDRIVER2), two languages (LANGUAGE 1-1, which is mandatory and LANGUAGE 2-1) and two GUIs (GUI1, GUI2).
MiniMercure: is a lightest product that supports only ENGINE1, GUI1, LANGUAGE 1-1, MANAGER1, and NETDRIVER1.

5.1. The decision model in a Product Line

In [12], the creational design pattern *abstract factory* [8] is used to refine the several variation points. This process is easily customizable by defining an interface for creating variants of Mercure’s five variation points (Engines, Net Drivers, Managers, GUIs and Languages). Obtaining an actual variant of the Mercure PL then consists in implementing the relevant concrete factory. The idea is originally used to simplify the Software Configuration Management by reifying the variant of an object-oriented software system into language-level objects. Our aim in this section is to use this idea as a design of the PL decision model.
The decision model of the Mercure PL is illustrated in the figure 6. Each concrete factory is related to one product in the Mercure PL, and each creational operation in the different concrete factories corresponds to a variation point. We use stereotypes to restrict the returned type of creational operations to the possible one. For example, the product model corresponding to the concrete factory CustomMercure includes only GUI1, and GUI2 classes as GUI variants. So we add two stereotypes <<GUI1>> and <<GUI2>> to the operation new_gui().

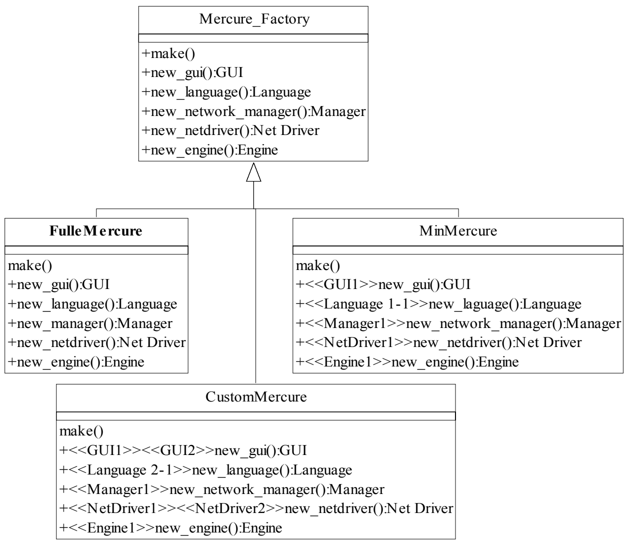

Mercure_Factory
+make()
+new_gui():GUI
+new_language():Language
+new_network_manager():Manager
+new_netdriver():Net Driver
+new_engine():Engine
FulleMercure
make()
+new_gui():GUI
+new_language():Language
+new_manager():Manager
+new_netdriver():Net Driver
+new_engine():Engine
MinMercure
make()
+<<GUI1>>new_gui():GUI
+<<Language 1-1>>new_laguage():Language
+<<Manager1>>new_network_manager():Manager
+<<NetDriver1>>new_netdriver():Net Driver
+<<Engine1>>new_engine():Engine
CustomMercure
make()
+<<GUI1>><<GUI2>>new_gui():GUI
+<<Language 2-1>>new_language():Language
+<<Manager1>>new_network_manager():Manager
+<<NetDriver1>><<NetDriver2>>new_netdriver():Net Driver
+<<Engine1>>new_engine():Engine

Figure 6. The Abstract Factory as a model decision for the Mercure PL

5.2. Product model derivation

At this stage, we have precisely defined the Product Line, now we have to tackle with the automation of the derivation process exploiting the abstraction variability pattern and the decision model. The description of the transformation algorithm used to derive product models is illustrated in the figure 7. The transformation algorithm is decomposed in three steps: variants selection, model specialization, and the model optimization.

1. The variants selection: Variation points are defined by return types of concrete factory operations. The selected variants are defined by their significant stereotypes (as names of variants). When the operation does not define stereotypes (such as in the FullMercure concrete factory operations), all sub classes of its return type is selected,
2. the model specialization: it removes all optional classes from the model, which have not been selected in 1. However, optional ancestors of selected variants are not removed,

3. the model optimization: it deletes unused factories and optimize the model (i.e when there is only one concrete class inheriting from an abstract one).

The product line model should satisfy generic constraints before the derivation and the product model derived should satisfy specific constraints. The generic constraints represent the pre-conditions of the transformation operation and the specific constraints represent the post -conditions:

```
DeriveProductLine(aConcreteFactory:Class,
PL_model:Model)
 pre : -- check Generic Constraints on PL_model
 post :-- check Specific Constraints on the
product model obtained
```

The figure 8 illustrates the CustomMercure product model that we have obtained after derivation of the Mercure PL.

```
DeriveProductLine

Input: PL_model: Model
       aConcreteFactory: Class
Output : Product_model: Model

--Variants selection

 Initiate selectedVariantsList to empty;
for each factory operation in
  aConcreteFactory do
  initiate definedVariantsList to
    significant stereotypes of the operation;
  if definiedVariantsList is empty
    then selectedVariantsList.add(all sub
classes of the returned type of the operation);
    else
selectedVariantsList.add(definedVariantsList) ;
  endif
done

-- Model specialization

for each optional class C in PL_model do
 if (the class name of C not in
 selectedVariantsList) and ( names of all sub
 classes of C not in selectedVariantsList)
 then
  delete the class C from the PL_model;
 endif
done

-- Model optimization

delete all other factories;
optimize inheritance;
Product_model := PL_model;
```

Figure 7. Deriving a product line UML model

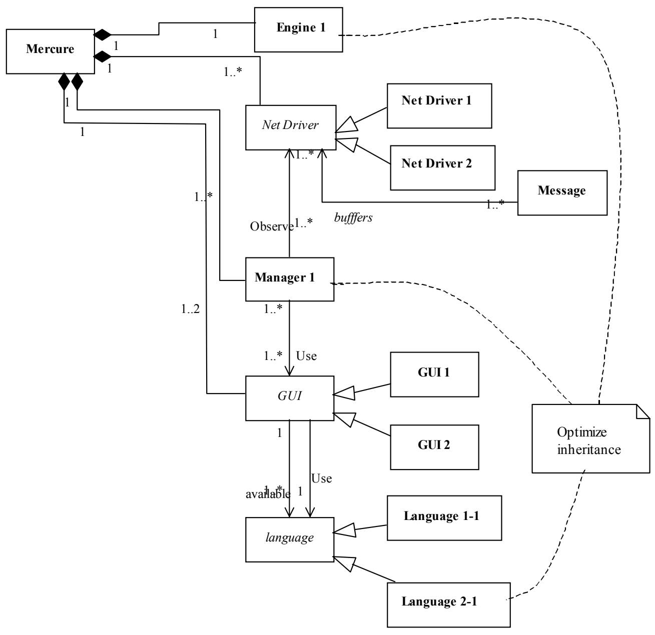


Figure 8. The CustomMercure Product UML model

6. Conclusion

We have proposed an approach based on the UML to model and to derive Product Line models. This approach especially focuses on static models represented by the UML class diagrams. To achieve this, we propose the use of the UMLAUT framework [22] combined to the Model transformation Language (MTL).
UMLAUT is a framework for building tools dedicated to the manipulation of models described using the UML. A specific use is to apply a model transformation to an UML model, automating the derivation process then consists in writing the relevant model transformation.
This transformation retrieves the useful model elements thanks to the selected concrete factory and then builds a specialized UML model corresponding to the selected Product. The challenge of such model manipulation is to be able to transform the model accessing its meta-level and ensuring the integrity of the derived model accordingly to the introduced specific constraints. A new version of the UMLAUT framework is currently under construction in the Triskell[2] team based on the MTL language, which is an extension of OCL with the MOF(Meta-Object Facility) architecture and side effect features, so it permits us to describe the process at the meta-level and to check OCL constraints (the generic

[2] http://www.irisa.fr/triskell/

constrains at first sight and specific constraints once the product model is derived). We present in appendix a detailed description of the derivation process as example of the MTL procedure.

As future work, we want to implement a UML profile for Product Line (including behavior aspects as proposed in [21]). This UML profile defines a set of stereotypes and a set of generic constraints to ensure any PL correctness. The user PL specification includes a set of models enriched by specific constraints, which may guide the derivation process. The derivation consists in applying a transformation algorithm written in MTL.

The abstract factory derivation approach was described here for a specific PL, which is the Mercure project. We think that it's possible to generalize this solution for others product lines that use the same abstraction variability pattern.

7. References

1. Bass, L., Clements, P., and Kazman, R. *Software Architecture in practices*, Addison-Wesley, 1998.
2. B. Keepence, M. Mannion, "Using Patterns to Model Variability in Product Families", IEEE Software, 16(4): pages 102-108, 1999.
3. C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line lopment. the KobrA approach", In Proc. of the 1st Software Product Lines Conference (SPLC1), pages 289–309,2000.
4. Czarnecki K., Eisenecker U.W., *Generative Programming: Methods, Tools, and Applications*, Addison-wesley, 2000.
5. ESAPS project deliverables. http://www.esi.es/esaps/
6. G. Kiczales, et al, "Aspect-Oriented Programming", In ECOOP'97 –Object Oriented Programming 11th European Conference, 1997.
7. G. Sunyé, A. Le~Guennec, and J.M. Jézéquel, "Precise modeling of design patterns", In LNCS, editor, Proceedings of UML 2000, volume 1939 of LNCS, pages 482--496, 2000.
8. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995
9. J. Bayer, "Toward engineering product line using concerns", GCSE 2000, Young Workshop, 2000.
10. Jézéquel, J.-M.. *Object Oriented Software Engineering with Eiffe,*. Addison-Wesley. ISBN 1-201-63381-7, 1996
11. J-M. Jézéquel, "Object-orented design of real-time telecom systems", In IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC'98, Kyoto, Japan (April 1998).
12. J-M. Jézéquel, "Reifying Variants in Configuration Management", ACM Transaction on Software Engineering and Methodology, pages 526-538, 1998.
13. Kang.k. et al Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, November 1990.
14. M. Anastapoulos, C. Gacek,, "Implementing Product Line Variability", Technical report IESE report N°: 089.00/E, Franhofer IESE publication, 2000.
15. M. Clauß, "Modeling variability with UML", In GCSE 2001 Young researchers Workshop. 2001
16. M. Svahnberg, J. Bosch, "Issues Concerning Variability in Software Product lines", in F. van der Linden, editor, Software Architecture for Product Families International Workshop IW-SAPF-3, LNCS 1951, pp. 146-157, Springer 2000.
17. M.L. Griss, "Implementing Product-line Features by Composing Component Aspects", in Proceedings of the First Software Product Line Conference, P. Donohoe, pp. 271-288, 2000.
18. Northrop.L., A Framework for Software Product Line Practice–Version 3.0., http://www.sei.cmu.edu/pLdP/framework.html#framework_toc, Software Engineering Institute (SEI), 2002.
19. OMG. UML specification. Version 1.4, 2001.
20. Pierre America and Steffen Thiel and Stefan and Martin Mergel, Introduction to Domain Analysis, ESAPS project, 2001 web = http://www.esi.es/esaps/.
21. T. Ziadi, L. Hélouët, J-M. Jézéquel, "Modeling Behaviors in Product Lines", International Workshop in Engineering Requirement for Product Line (REPL'02), Essen, 2002.
22. W.-M. Ho, J-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h, "UMLAUT: an extensible UML transformation framework", In Proc. Automated Software Engineering, ASE'99, Florida, October 1999.
23. Warmer, J., and Kleppe, A.. *The Object Constraint Language – Precise Modeling with UML*, Object Technology Series. Addison-Wesley, 1998

Appendix

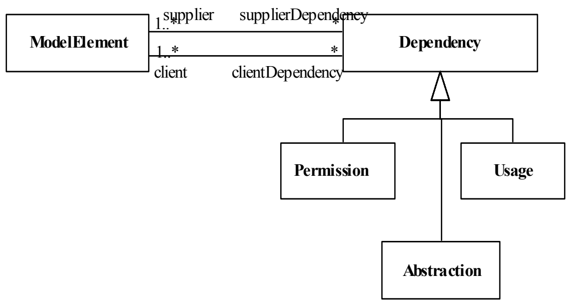
A.1: OCL Auxiliary operations

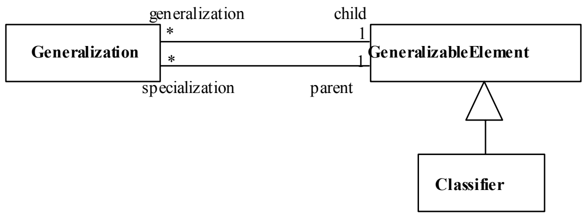
```
context
 ModelElement::isStreotyped(S : String):Boolean
  post :result =
   self.stereotype →exists(s |
       s.name = S)


context
  Namespace::presenceClass(C :String): Boolean
  post :result =
   (self.oclIsKindOf(Class) and self.name = C))
     or
   (self.presenceClass(C))


context Class::AllSubClasses() : Set(Class)
  post: result =
  self.specialization.child → iterate(c:Class;
acc: Set(Class) = Set{} | acc →
including(c)→union(c.AllSubClasses()))

context Namespace::AllClasses() : Set(Class)
 post : result =
 self.ownedElement → select(me: ModelElement|
  me.oclIsKindOf(Class)) → union
(self.ownedElement. AllClasses())
```

A.2: A detailed description of the derivation algorithm

```
--Based on OCL extended with side effect
features

ProductLineDerivation(aConcreteFactory:Class,
pl:Model)
BEGIN

--Variant selection

Set(String) definedVariants
Set(String) selectedVariants
for op in
aConcreteFactory.feature→select( f: Feature
  | f.oclIsKindOf(Operation)
and f.name.startsWith('new_') )
do
 Class opsReturnType :=
  ( op.parameter→select( p:Parameter | p.kind =
                   #return) ).type
 definedVariants:= op.stereotype.name →
     intersection(
             opsReturnType.AllSubClasses().name)
 if definedVariants →isEmpty()
 then selectedVariants :=selectedVariant →
      union(opsReturnType.AllSubClasses().name)
 else selectedVariants :=selectedVariant →
      union(op.stereotype.name)
 endif
done

--Model specialization

for C:Class in  pl.AllClasses()
do
 if (C.isStereotyped('optional')) and
  (selectedVariant→exludes(C.name)) and
  selectedVariant→
       exludesAll(C.AllSubClasses().name)
 then
   deleteElement(C, pl)
 endif
done

-- Model optimization

aConcreteFactory.generalization.parent.specializ
ation.child→
excluding(aConcreteFactory)→collect(cf : Class|
    deleteElement(cf, pl))

optimizeInheritance( pl)

END
```

A.3: The Dependency and the generalization meta-classes in the UML meta-model





Short Papers

Software Variability Management Using a Platform Based Autonomous Agents

Amar RAMDANE-CHERIF, Samir BENARIF and Nicole LEVY
PRISM, Université de Versailles St.-Quentin, 45, Avenue des Etats-Unis,
78035 Versailles Cedex, France
{rca}@prism.uvsq.fr

Abstract
System architectures embody the same kinds of structuring and decomposition decisions that drive software architectures. Moreover, they include hardware/software tradeoffs as well as the selection of computing and communication equipments, all of which are completely beyond the realm of software architecture. The foundation of any software system is its architecture, that is, the way the software is constructed from separately components and the ways in which those components interact and relate to each other. If the requirements include goal for variability management, then the architecture is the design artifact that first expresses how the system will be built to achieves this goal. Some architectures go on to become generic and adopted by the development community at large: three-tier client server, layered, and pipe-and-filter architectures are well known beyond the scope of any single system. In this paper, we use a platform based on multi-agents system in order to test, evaluate component, detect fault and error recovery by dynamical reconfigurations of the architecture. This platform is implemented on pipe-and-filter architecture which is applied for controlling a mobile robot to follow a trajectory towards the desired objective in the presence of obstacles. The hardware/software of this architecture system is completely monitored by the platform in order to evolve quality attribute variability. Some scenarios addressing the variability at architectural level is outlined by both with and without using our platform-based-agents. In this paper, we discuss how our approach supports the variability management of complex software / hardware systems.

1. Introduction

A critical aspect of any complex software system is its architecture. The architecture deals with the structure of the components of a system, their interrelationships and guidelines governing their design and evolution over time [1][2]. The architectural model of a system provides a high level description that enables compositional design and analysis of components-based systems. The architecture then becomes the basis of systematic development and evolution of software systems. It is clear that a new architecture that permits the dynamism reconfiguration, adaptation and evolution while ensuring the variability management of an application is needed. The variability is defined as the ability of a software system or artifact to be changed, customized or configured for use in a particular context [3][4][5]. The architectural level reasoning about the variability quality attribute is only just emerging as an important theme in software engineering. This is due to the fact that the variability concerns are usually left until too late in the process of development. In addition, the complexity of emerging applications and trend of building trustworthy systems from existing, untrustworthy components are urging variability concerns be considered at the architectural level. In [6] the researches focus on the realization of an idealized fault-tolerance architecture component. In this approach the internal structure of an idealized component has two distinct parts: one that implements it's normal behavior, when no exceptions occur, and another that implements it's abnormal behavior, which deals with the exceptional conditions. Software architectural choices have profound influence on the quality attributes supported by system. Therefore, architecture analysis can be used to evaluate the influence of the design decisions on important quality attributes such as variability management [7]. Another axe of research is the study of fault descriptions [8] and the role of event description in architecting dependable system [9]. Software monitoring is a well-know technique for observing and understanding the dynamic behavior of programs when executed, and can provide for many different purposes [10][11]. Besides variability, other purposes for applying monitoring are: testing, debugging, correctness checking, performance evaluation and enhancement, security, control, program understanding and visualization, ubiquitous user interaction and dynamic documentation. Another strategy is used, like a redundant array of independent component (RAIC) which is a technology that uses groups of similar or identical distributed components to provide dependable services [12]. The RAIC allows components in redundant array to be added or removed dynamically during run-time, effectively making software components "hot-swappable" and thus achieves greater overall variability. The RAIC controllers use the just-in-time component testing technique to detect component failures and the component state recovery technique to bring replacement components up-to-date. The approach in [13] advocates the enforcement of variability requirements at the architectural design level of a software system. It provides a guideline of how to design an architectural prescription from a goal oriented requirements specification of a system. To achieve high variability management of software/hardware, the architectures must have the capacity to react to the events (fault) and to carry out architectural changes in an autonomous way. That makes it possible to improve the properties of quality of the software application [14]. The idea is to use the architectural concept of agent to carry out the functionality of reconfiguration, to evaluate and to maintain the quality attributes like variability management of the architecture [15]. Intelligent agents are new paradigm for developing software/hardware applications. More than this, agent-based computing has been hailed as "the next significant break-through in software development" [16], and "the new revolution software" [17]. Currently, agents are the focus intense interest on the part of many sub-fields of computers science and artificial intelligence. An agent is a computer system situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives. Autonomy is a difficult concept to pin down precisely, but we mean it simply in the sense that the system should be able to act without the direct intervention of humans (or other agents), and should have control over its own actions and internal state. It may be helpful to draw an analogy between the notion of autonomy with respect to agents and encapsulation with respect to object-oriented systems. In this paper, we propose a new approach which provide a platform based agents. This platform will monitor the global architecture of a system and improve variability quality attribute. It will achieve its functional and non functional requirements and evaluate and manage changes in such architecture dynamically at the execution time.

This paper is organized as follows. In the next section, we will introduce the platform based multi-agents. Then a strategy to achieve fault tolerance by our platform will be presented. In section four, we describe an example showing the application of our platform on Pipe-and-Filter architecture and its benefits are outlined through some scenarios about the variability management. Finally, the paper concludes with a discussion of future directions for this work.

2. The platform multi-agents

In recent years, agents and Multi-Agent Systems (MAS) have become a highly active area of Artificial Intelligence (AI) research. Agents have been developed and applied successfully in many domains. MAS can offer several advantages in solving complex problems compared to conventional computation techniques. The purpose of traditional Artificial Intelligence is to perform complex tasks, thanks to human expertise. This often assumes assimilation of many competencies to be subject of centralized programming. Moreover, in such monolithic system, the consensus between various expertises is difficult to model; indeed, the structure of communication between the experts is fixed whereas it should depend on the considered problem. Thus, a formalization close to reality where several people work together on a same problem is needed. Such formalism should describe the participants and interactions between them. This approach is the paradigm of the Distributed Artificial Intelligence (DAI). The DAI leads to the realization of systems known as "multi-agent" systems allowing modeling the behavior of all the entities according to some laws of social type. These entities or agents have certain autonomy and are immersed in an environment in which and with which they interact. Their structure is based on three main functions: perceiving, deciding and acting.

The term "agent" is subject to many interpretations. The most used one is: "an agent is an autonomous entity which pursues an individual goal, which is ready to act on the environment of the system to which it belongs and/or to interact with the other agents, which has only one evolutionary representation of this environment and which can perceive the other agents thanks to the communication or the observation".

The modeling of a multi-agents system can be based on four dimensions (figure-4-) which are: Agent (A), Environment (E), Interaction (I), and Organization (O). Facet A indicates the whole of the functionalities of internal reasoning of the agent. The facet E gathers the functionalities related to the capacities of perception and actions of the agent on the environment. Facet I gathers the functionalities of interaction of the agent with the other agents (interpretation of the primitives of the communication language, management of the interaction and the conversation protocols). The facet O east can be most difficult to obtain, it relates to the functions and the representations related to the capacities of structuring and management of the relations of the agents between them.

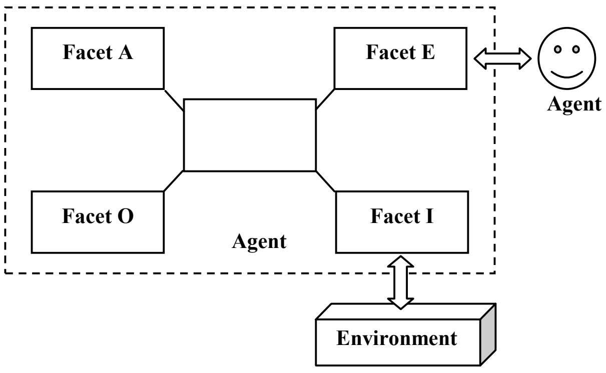


Figure 1: AEIO Facets within an agent

While following a logical reasoning, we thus manage to perceive two layers in our platform, but it is noticed well that we need a link between the two various layers, since the reactive layer answers only to stimulus, and the higher layer is dedicated to management and reasoning. Thus, we need a layer which interacts with the two layers, it must act on the reactive layer by stimulating and coordinating the actions of these agents, but also interact with the higher layer by informing it of the state of the architecture and the agents. This layer acts as links between the decisional and the reactive parts of the platform. This offers to us a division of the tasks and a specialization of the layers. Thus we obtain the speed, flexibility and a weaker cost of communication as well as a greater stability of the all platform, resulting from the cooperation and the coordination of the layers.

The other aspect of our problem is the dynamic nature of our architecture, indeed architecture does not cease to evolve, to reconfigure and to extend. It is inconceivable to create a rigid and static platform which can follow the evolution of this architecture!. We must thus already think of such a dynamic and evolutionary platform so that it can constantly reach and follow the evolution of this architecture. We will consider that our software architecture is a such board cut out in small pieces. We consider that we can extend this board as parts are added. We have also the freedom to modify the parts and to make them move on the board. While considering this example, we will establish specific rules to the platform based multi-agents which we will build. We will consider that the available software architecture is divided into localities, grouped, it forms one or several zones. This strategy will enable us to better control the characteristics of modifiability and extensibility of the available architecture. The architecture of our platform consists of three distinct layers. A layer known as higher equipped with evolved agents able to communicate with the external environment or other agents in order to establish the plans and the adequate strategies to achieve the desired goals. A second layer comes in continuation, which is the intermediary layer, located between two layers, communicates with the higher layer and the lower layer known as a reactive layer. The agents in the intermediary layer are less evolved than the agents of the higher layer (equipped with a less advanced social nature). The last layer is the reactive layer having purely reactive agents to a stimulus, their roles are limited exclusively to the perception/action (figure-2-,-3- and -4-).

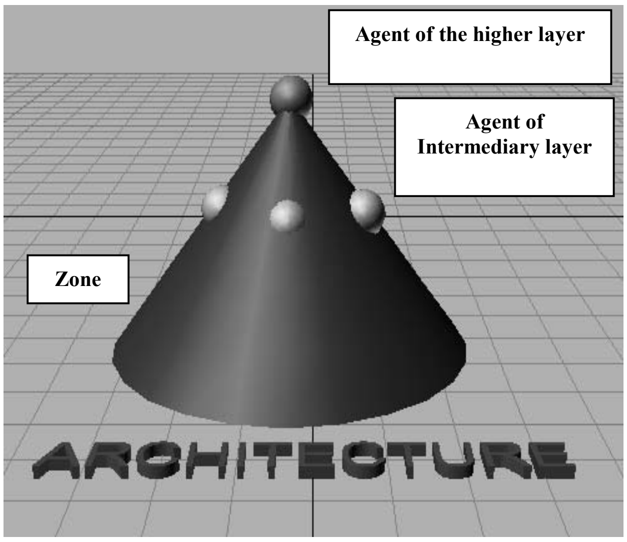


Figure 2: Hierarchy of the platform and representation of a zone

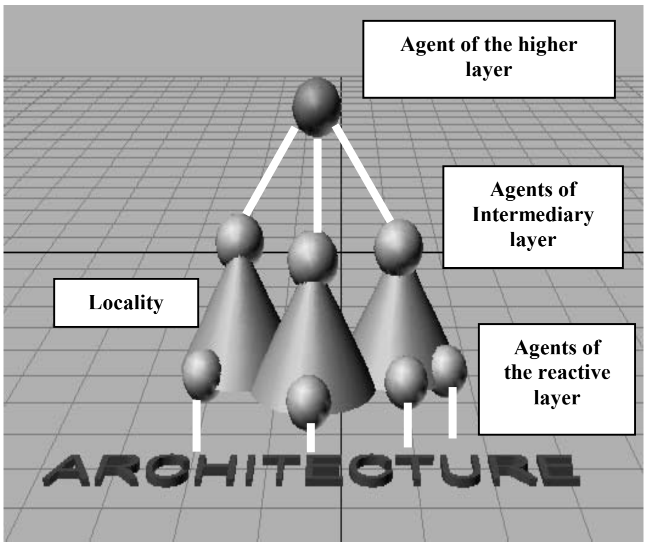

Agent of the higher layer
Agents of Intermediary layer
Locality
Agents of the reactive layer
ARCHITECTURE

Figure 3: Hierarchy of the platform and diagram of a locality

2.1. The higher layer

The higher layer is the highest layer of the platform, it is thus, more evolved than the others. This layer has the capacity to analyze information coming from architecture, thanks to the facet E of its agents. Thus, it can evaluate qualities of architecture constantly and intervene in a targeted way, since the agents have a facet A, implying the reasoning. The facet O and I, of the agents enter in action when the agents of the intermediary layer do not manage to find only a solution to a problem. The agents of the higher layer have the capacity to organize a group of agents in the intermediary layer (implies a cooperation) or to utilize another agent of the higher layer (implies a negotiation) in order to achieve the goal to seek. For example, an agent of the intermediary layer controlling a desired locality can add a component being in another locality. The solution which is offered to him, is to refer to the agent which supervises it, namely the agent of the higher layer, which will put him in direct contact with the agent which controls the locality concerned if this one belonged to its own zone. In the contrary case, the agent starts a negotiation with the agent which supervises the locality concerned. The agents of this layer can constantly exchange information relating to the zone which it controls so that they always have a global and complete architecture vision. Each agent of this layer controls a zone of architecture, it is responsible for a group of agents of the intermediary layer. The planning by analysis of environment is specific to the higher layer. The capacities of perception of the environment and of organization of the agents offer a greater coordination in the platform. Thus, we facilitate the division of the work by directing the agents toward common goals. The agents of the higher layer act according to the received messages from their environments and other agents. By coordinating this information, they establish a work plan, which targets the objective to be reached and which defines the coordinating agents for achieving the goal. In other words, by dividing work according to the agents aptitudes. The agent of the higher layer can perceive signals coming from architecture (system) or from the agents (agent of the higher layer or intermediary layer). The perceived information (by using facets I,E) is sorted, classified and decoded according to the protocol used for each type of message. Thereafter, the agent define the objective to be reached by identifying the place and the type of the desired reconfiguration. Thus, it adopts one of the strategies implemented in its knowledge base, it is the facet reasoning of the agent. Then, the agent establishes a plan according to the information collected by its sensors and the available information on the architecture in its knowledge base. By adopting a specific plan, the agent can act in three manners: A) Negotiation: It can start a negotiation with an agent of the higher layer so that it can complete work, if the desired reconfiguration is apart from its own zone. B) Cooperation: the agent established a plan of cooperation between the agents of the intermediary layer, if the reconfiguration is in its own zone. C) Action: the agent can act of itself, for example the creation of a new agent in the intermediary layer, carrying out a simple test or making a reconfiguration on architecture (this action is very limited). The strong points of this layer are: 1 - Knowledge bases distributed and exchanged constantly between the agents of the higher layer, which avoids the losses of information in the event of breakdown. 2 - A very high social character, thanks to facet O,I of the agent: thus being able to organize agents or to negotiate with agents an application of a task. 3 - A low number of agents: imply a better coordination of the actions and a weak cost of communication.

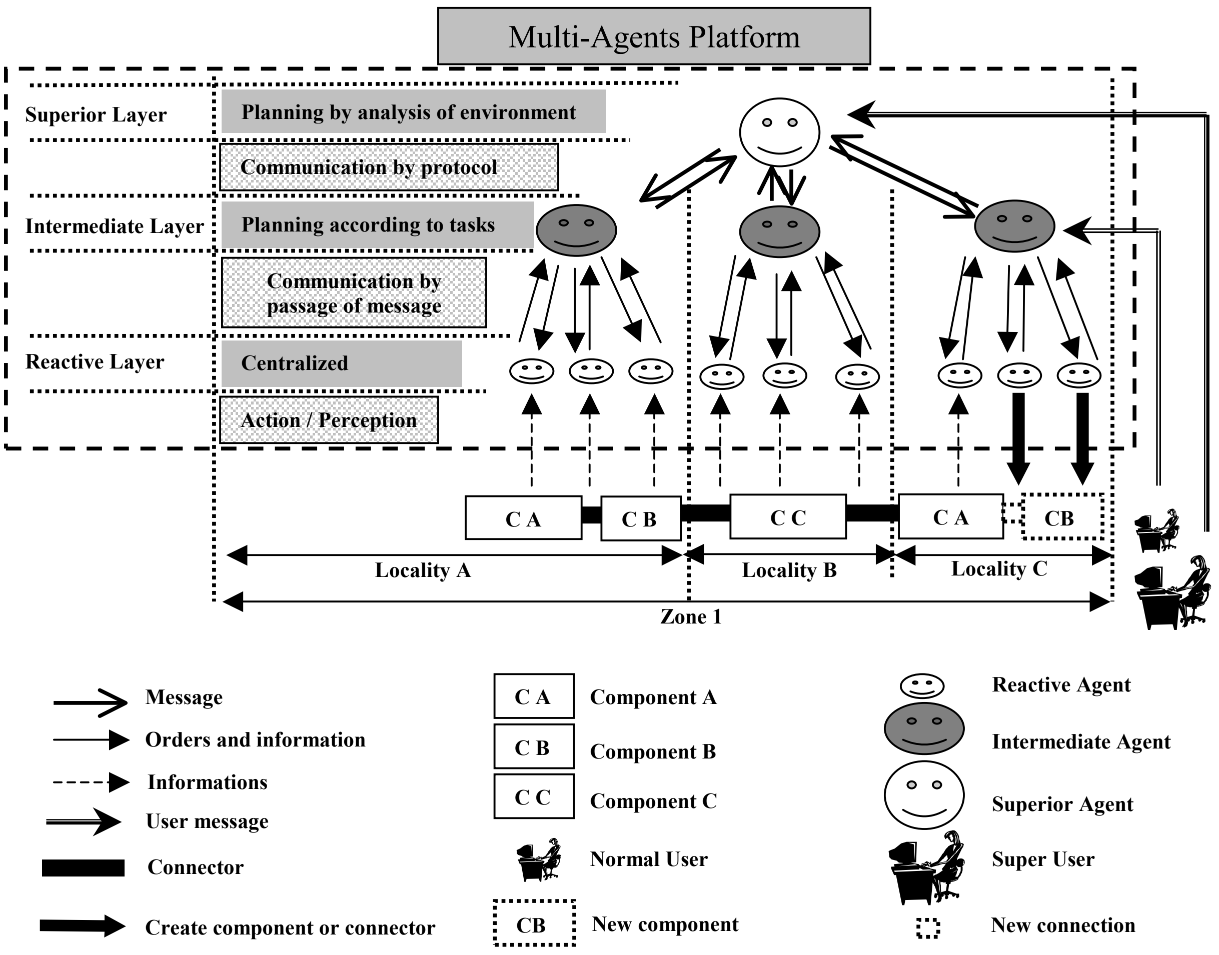


Figure 4: Configuration of the platform

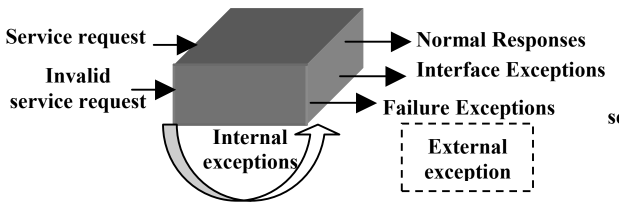


Figure 5: Error on component

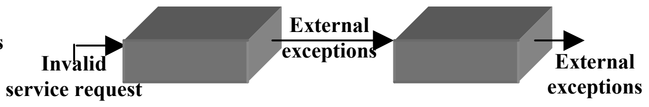


Figure 6: Propagation of the fault

2.2. Intermediary layer

As its name indicates it is a layer which is placed between the higher layer and the reactive layer. Each agent of this layer takes care of several agents of the reactive layer, it is responsible for a quite precise locality. The agent itself is connected to only one agent of the higher layer. A set of agents of the intermediary layer forms what is called a zone. The principal role of this layer is to take care of the good progress of the reconfigurations imposed by the higher layer. It is a question of controlling and coordinating the agents of the reactive layer in order to carry out and to achieve a goal. Another role of this layer is the collection of information coming from the reactive layer in order to forward them to the agent of the higher layer. The agents of the intermediary layer can be confronted with two kinds of problems: queries of reconfiguration in their locality, but also outside. From where the name of planning according to task. The agent establishes two kinds of plans so that it can answer to the requests which they are: a planning centralized with the agents of the reactive layer or a planning distributed in certain case, toward the supervisory agent of the higher layer: A) Distributed planning: In the intermediary layer, the agents use a distributed planning. In the case where they are in the incapacity to solve only the posed problem. They refer to the agents of the higher layer. The agents of this layer break up the problem into sub-problems and elaborate the sub-plans so that they can be carried out by the agents of the intermediary layer. B) Centralized planning: In certain case, the agents are unable to solve only the posed problem. For example, if we ask an agent to reconfigure a locality which it does not control, in this precise case, the plans are generated by the higher layer. This layer has a total sight of architecture and platform. Thus the higher layer put in cooperation mode agents of intermediary layer in order to carry out work requested, by dividing and managing the work of each one. Contrary to the agents of the higher layer, the agents of the intermediary layer do not have advanced social character. The communications between the agents of this layer are simple and indirect, i.e. that they are conveyed by the agents of the higher layer. The agents are thus limited to an interaction with the agents of the higher layer described above, and a communication by passage of asynchronous message with the reactive agents by directing acts primarily.

2.3. Reactive layer

This layer is the body of perception and of action of the platform. It is equipped with purely reactive agents which act with simple stimulus coming from the intermediary layer. The reactive agents belong to a locality depending on only one agent of the intermediary layer whose they receive the plans. These agents answer to a centralized planning and work in cooperation. The exchange between the reactive agents and the agent of intermediary layer is simple. The perception induces sending simple information toward the central agent, the action is the consequence of a stimulus or a simple command.

3. The platform and fault tolerance

3.1. Fault at architectural level

The basic strategy to achieve fault tolerance in a system can be divided into two steps. The first step called error processing is concerned with the system internal state, aiming to detect errors that are caused by activation of faults, the diagnostic of the erroneous states, and recovery to error free states. The second step, called fault treatment, is concerned with the sources of faults that may affect the system and includes: fault Planning and fault removal.

The communication between components is only through request/response messages. Upon receiving a request for a service, the components will react with a normal response if request is successfully processed or an external exception, otherwise. This external exception may be due to the invalid service request, in which case it is called an interface exception, or due to a failure in processing a valid request, in which it is called a failure exception (Figure 5). The error can propagate through connector of software architecture by using the different interactions between the components (Figure 6). Internal exceptions are associated with errors detected within a component that may be corrected, allowing the operation to be completed successfully; otherwise, they are propagated as external exceptions.

3.2. Monitoring system

Software monitoring is a well-know technique for observing and understanding the dynamic behavior of programs when executed and can provide for many different purposes. Besides variability, other purposes for applying monitoring are testing debugging, correctness checking, performance evaluation and enhancement, security, control, program understanding and visualization, ubiquitous user interaction and dynamic documentation. System monitoring consists in collecting information from the system execution, detecting particular events or states using the collected data, analyzing and presenting relevant information to the user, and possibly taking some (preventive or corrective) actions. As the information is collected from the execution of the program implementation, there is inherent gap between the levels of abstraction of the collected events, states of the software architecture. For event monitoring, there are basically two types of monitoring systems based on the information collection: sampling (time-driven) and tracing (event-driven). By sampling, information about the execution state is synchronously (in a specific time rate), or asynchronously (through direct request of the monitoring system). By tracing, on the other hand, information is collected when an event of interest occurs in the system. Tracing allows a better understanding and reasoning of the system behavior than sampling. However, tracing monitoring generates a much larger volume of data than sampling. In order to reduce this data volume problem, some researchers have been working on encoding techniques. A more, common and straightforward way to reduce data volume is to collect interesting events only, and not all events that happen during a program execution. The second approach may limit the analysis of events and conditions unforeseen previously to the program execution. Both state and event information are important to understand and reason about the program execution. Since tracing monitoring collects information when events occur, state information can be maintained by collecting events associated to state change. With a hybrid approach, the sampling monitoring can represent the action of collecting state information into an event for the tracing monitoring. Not all events with state information should be collected, but only the events of interest. Integrating sampling and tracing monitoring and collecting the state information through events reduce the complexity of the monitoring task. The monitoring system needs to know what are the events of interest, what events should be collected.

3.3. Detection of faults with the platform based agents

We will use a monitoring system based on the agents, by implementing our platform, described above, on the top of the architecture. Each component will be supervised by a reactive agent, by sampling or tracing. The reactive agents will use sampling on architecture and collect information on the state of the components with each interval of time predefined or limited by the user. Another type of detection in reactive agent is the tracing, in this case, the component generates an external exception in the form of an event, this event will be collected and will be transmitted towards the intermediate agent, this event will be thereafter analyzed, identified and then sent by this agent towards the agent of the superior layer in order to establish plans to correct the errors. In other words, the signals are collected by the agents of the reactive layer, which transmit them immediately to the intermediate agent of its locality. This agent analyzes this information using its knowledge base containing the description of the errors. Thus, it will sort information coming from the reactive agents and send only the error messages towards the agent of the superior layer of its zone. According to the detected errors the superior agent establishes the plans in order to solve the errors coming from architectural level (Figure-7-).

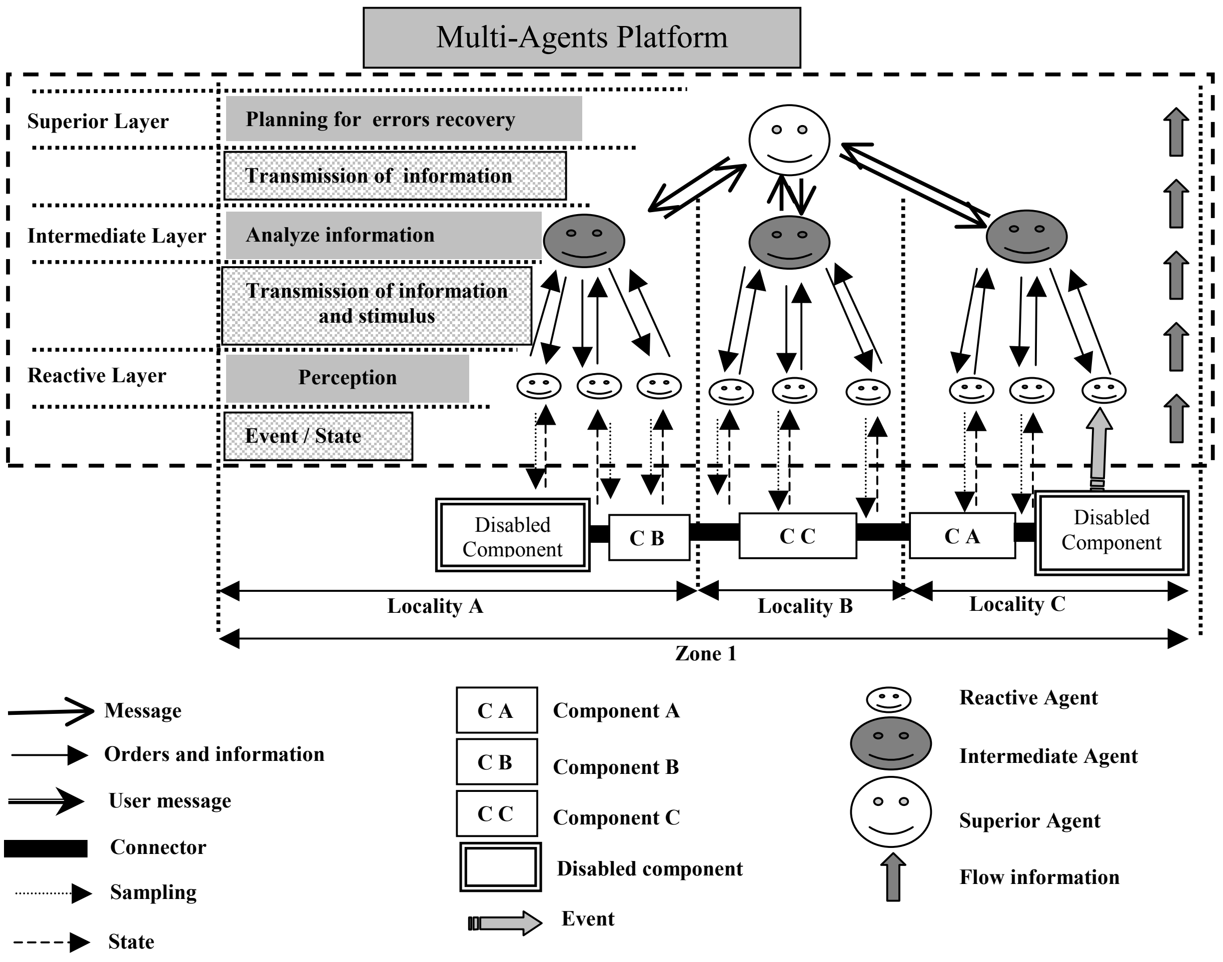


Figure 7: Multi-agents platform for monitoring

3.3. The treatment process

After the phase of detection, the platform identifies the type of error and establishes the plans in order to achieve at architectural level the necessary reconfigurations to correct the faults. This treatment process uses tow types of plans, the first plans consist to reconfigure architecture connections for finding temporary solution of fault (disabled component or connector), the second plans recover errors by addition or changing disabled component or connector.

3.3.1) Reconfiguration of connections :In the detection phase, the information travel up through the layers of the platform in order to arrive to the superior agent, in this decisional layer the treatment process begins by establishing plans. The superior agent chooses the best solution to support evolution and changing requirements of the architecture. The platform can reconfigure connections of architecture to isolate the disabled components (if the platform can't create new components), the superior agent distributes the plans to the intermediate agent on the locality of fault. When the intermediate agent receives the plans, it distributes directives to the reactive agents. The reactive agents delete the connection of disabled component and create new connection to isolate it.

3.3.2) Creation of new component :If the platform has the possibility to create new component in order to recover errors at architectural level, the superior agent distributes plans to the intermediate agent. This agent distributes directives to reactive agents, and the reactive agents work together in order to delete the disabled component and it's connection and create new component and it's new connection (Figure 8).

4. Implementation of the multi-agents platform on Pipe-and-Filter architecture

4.1. The navigation of the mobile robot in an environment without obstacle

We dispose of a mobile robot in a flat environment, it must go from a point initially to parameterize towards a finale point in a plan (environment represented here by a plan), the robot can move in a horizontal way or vertical way, when it is immobile, it can do rotation on itself. The mobile robot moves on a plan (Figure 9) which we divide into six parts by taking the finale position of robot the origin point of Cartesian coordinates (0,0). Thus, we distinguish six possibility approaches, if the robot is on parts 1, 2, 3 or 4, then it manages to reach the finale desired point by deploying a very simple navigation plan which is: an approach on the X axis, then a final approach on the Y axis. In both remaining cases (part 5 and 6), if the robot is on part 6, then it uses an approach on the X axis, or if it is on the part 5, then it starts an approach on the Y axis.

4.2. Pipe-and-Filter Architecture for the navigation of the mobile robot in an environment without obstacle

In an environment without obstacles, we will choose a Pipe-and-Filter architecture which corresponds as well as possible to our navigation strategy.

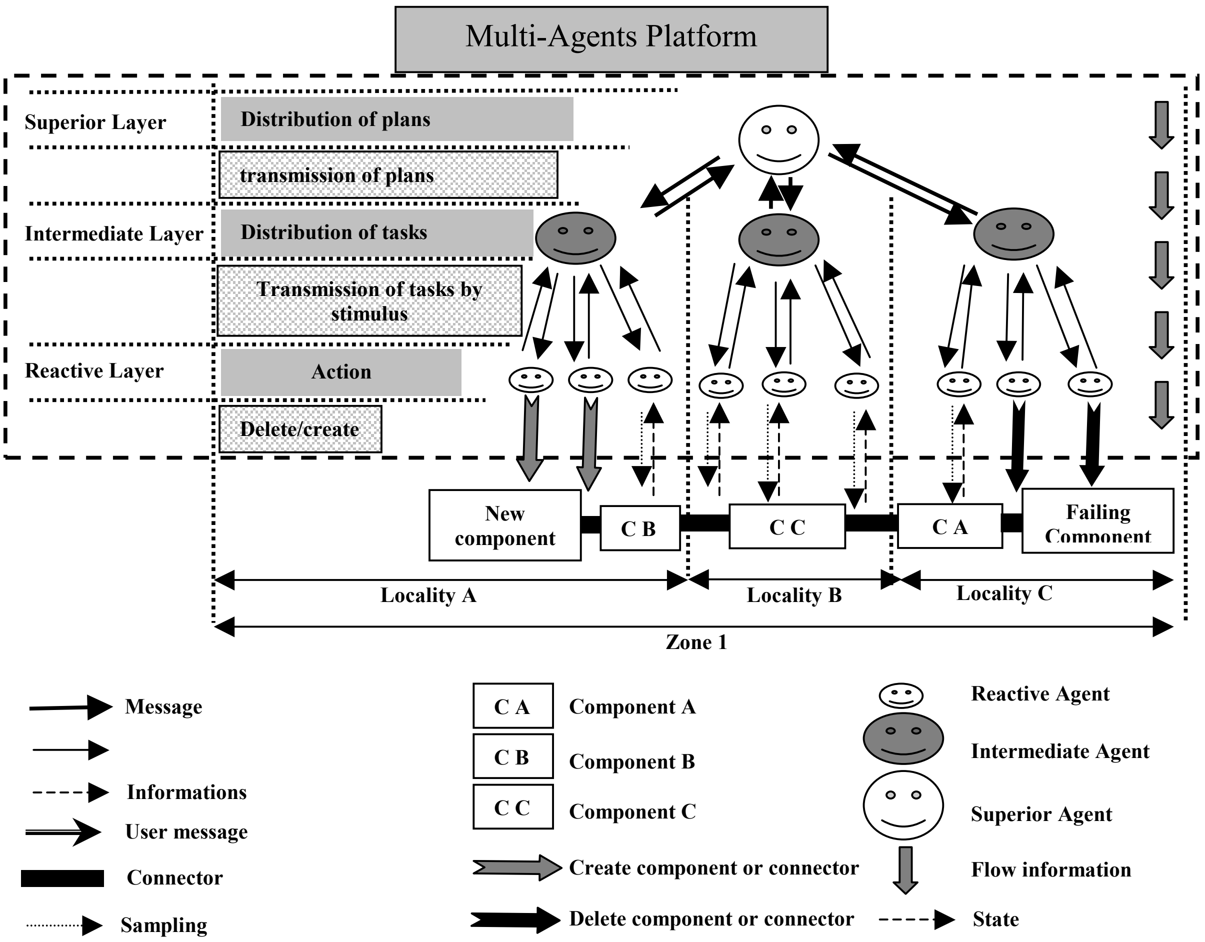


Figure 8: Multi-agents platform treatment process

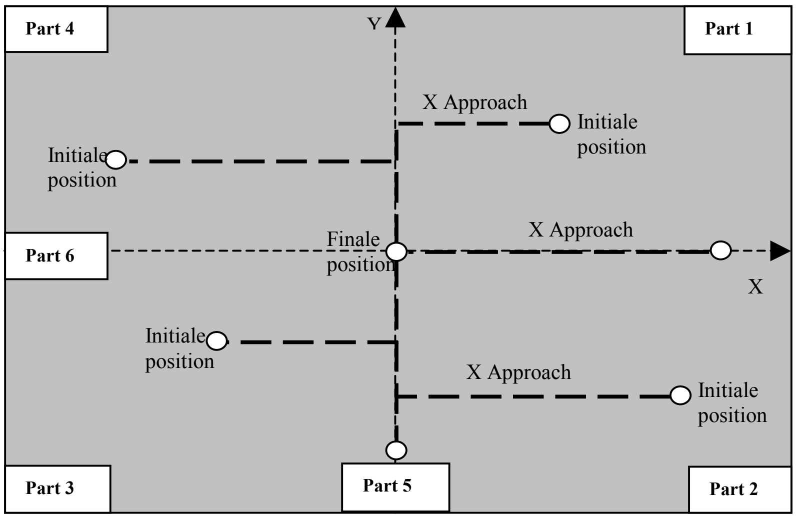


Figure 9: Strategy of navigation of the mobile robot

The first component (Figure 10), “Parameter ” is used to enter the Cartesian coordinates (X,Y) of the initial and finale position of the mobile robot. The component “Planning” defines the position of the robot in the plan in order to establish the ideal planning to reach the finale point. The component “X approach” increments X position of the mobile robot and the component “Y approach” increments position Y. The component “Simulation” is charged for displaying the robot displacement on the screen.

Example : In our example, the mobile robot is positioned on part 1 of the plan. When the user enter the parameters of the mobile robot (finale and initiale positions), the component “Planning” definites the first plan that the robot follows to reach the finale point. Therefore, a first approach on the X axis is activeted by the component “X Aproach”. The component “Simulation” is also actuated at each increment on the X axis in order to display step by step the movement of the robot. When the component “X Approach” finishes its approach, the positions of

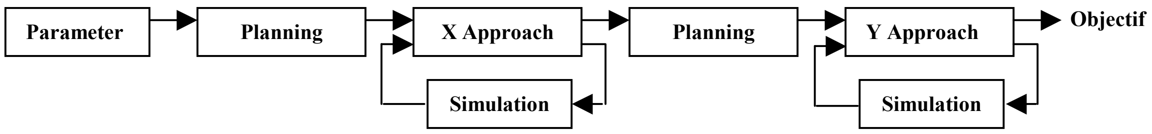


Figure 10: Pipe-and-Filter architecture for the navigation of obil robot

the robot are sent towards the component "Planning" which will define the new plan to be followed by the robot according to its positions. The approach on the Y axis is activated by the component "Y approach" as well as the display of each step of the robot by the component "Simulation". At the end of the incremantation on the Y axis the robot reaches its objective.

4.3. The navigation of the mobile robot in an environment with obstacle

The mobile robot moves in a flat environment (the plan) with obstacles which are positioned randomly (Figure 11). We will install a sensor on the robot which will help the mobile robot to detect the obstacles, when it tries to reach the final position. In order to avoid the obstacle we will use the same basic displacement of the robot, i.e. rotation on itself of 90° and the vertical or horizontal way. If the obstacle is out of the mobile robot trajectory then its origin navigation planning will not be affected. In other case the obstacle is on the trajectory of the mobile robot during its X or Y approach. When the obstacle is detected (the distance from detection of the mobile robot depends on the range of the used sensor). The mobile robot decreases its speed, then stops in order to make a rotation of 90° on itself and starts to avoid the obstacle. When this one is out of the trajectory, the robot carries out a new planning with new X or Y approaches to reach its finale position.

4.4. Pipe-and-Filter architecture for the navigation of the mobile robot in an environment with obstacle

The mobile robot moves in an environment with obstacle, the software architecture proposed previously is retained, but a new hardware component installed on the robot is taking into account, it represents, in our architecture, by a software component called the "Scan" (Figure 12). The mobile robot will use the new architecture which takes into account the possibility of founding obstacles on its trajectory with each incrementing on the Y or X axis.

4.5. The role of the platform to manage the variability in the mobile robot navigation

The multi-agents platform will be placed on the top of our Pipe-and-Filter architecture, and exerts on it a permanent monitoring in order to avoid all processing possible errors. Generally, the multi-agents platform reacts to the events emitted by the architecture using two distinct strategies: the reconfiguration of the component's connections or the creation of the new components able to solve the arise problem.

The sensor is installed on the robot and it sweeps sequentially its environment, in the case the sensor detects an obstacle on its trajectory, it sends a signal towards the component "Scan" of the software architecture, which emits an event towards the platform. On the level of the architecture, the error is collected by the reactive agent which supervises the component "Scan". The error is then transmitted towards its intermediate agent, this error is then identified and sent towards the superior agent. The superior agent establishes the plans in order to correct the errors, in this case, the multi-agents platform will create new components so that the robot avoids the detected obstacle.

When the obstacle is finally out of the trajectory of the mobile robot, the component "Planning" establishes new plans. If these plans require a reconfiguration of the connections, the component "Planning" emits an event towards the platform, which is collected by the reactive agent of the platform related to the component "Planning". The event is transmitted towards the intermediate agent which identifies the event thanks to its knowledge base describing the event which is emitted by the software architecture. The agent of the intermediate layer sends information towards the superior agent, which establishes the plans so that the error is corrected on the level of the architecture, and distributes them to the agent of the intermediate layer. The agent of the intermediate layer orders the reactive agents to create the new connectors necessary to the new navigation plan of the mobile robot.

5. Scenario of navigation of the mobile robot on an environment with obstacle

In this scenario the mobile robot is in part 1 of the plan (Figure 11), the final position is entered by the user. The obstacle will be placed on the first trajectory of the X axis. The mobile robot starts with an approach according to the X axis. After the detection of the obstacle by the sensor, the robot slows down for stopping, it makes a rotation of 90° on itself. Then the obstacle is avoided by choosing a vertical trajectory as soon as the obstacle is not located on the X axis trajectory, the mobile robot begins a new approach on the X axis, then finishes by an approach on the Y axis to achieve its finale goal.

This scenario is produced on the level of the architecture by applying the following steps:

5.1 *The mobile robot will use the starting configuration of the architecture, and starts its approach X.*

5.2 *The detection of obstacle and creation of components:* when the sensor detects the obstacle on its trajectory it emits one signal towards the "Scan" component, which will send an event towards multi-agents platform (Figure 13-a). The event will be detected by its reactive agent which transmits it towards its intermediate agent. The agent of the intermediate layer identifies the event and transmits the information to its superior agent. The superior agent establishes a plan which will be sent towards the intermediate agent. The intermediate agent orders to its reactive agents to create and activate new components and their connections (Figure 13-b). The information on the reconfiguration goes up towards

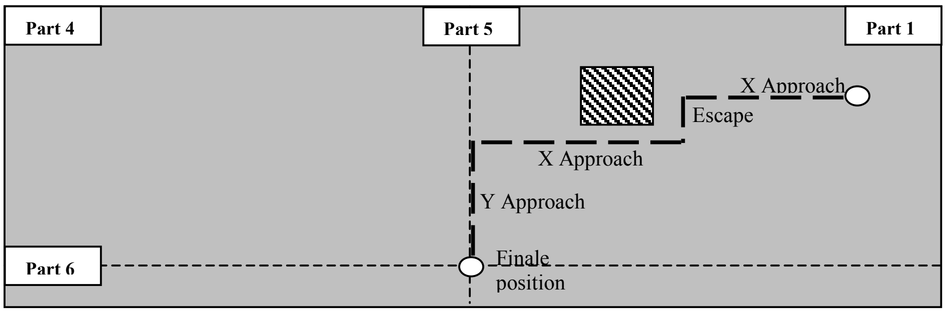


Figure 11: The navigation of the mobile robot in an environment with obstacle

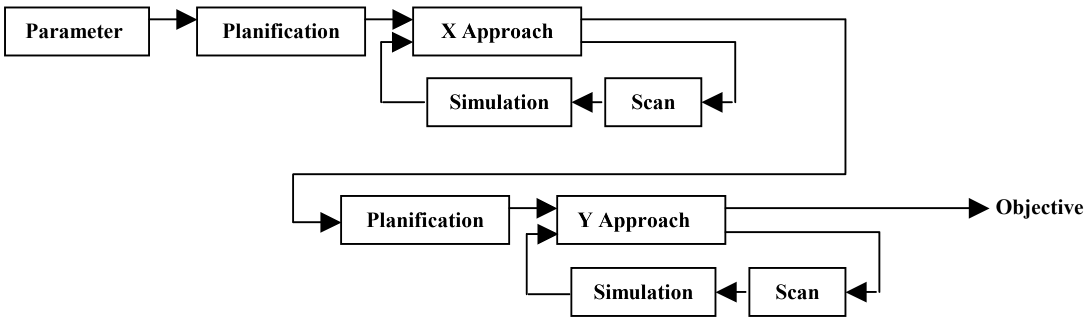


Figure 12 : Architecture of the navigation of the mobile robot in an environment with obstacle

the agent of the superior layer so that it will have a precise sight of the architecture state.

5.3. *The destruction of the useless components for new planning of the navigation:* the component "Analysis" collects information relating to the position of the robot as well as information coming from the "Scan" component. Then, this "Analysis" component activates both the "Escape" component which starts its plan to avoid the obstacle and the "Simulation" component for displaying the movement. If the obstacle is out of the trajectory of the mobile robot, the component "Escape" sends an event towards the platform to restore the original configuration of the architecture (Figure 13-c). This event is detected by the agent of the reactive layer and transmitted to its agent of the intermediate layer so that it can be identified. After the identification, the intermediate agent sends information towards its superior agent. The superior agent will establish again so that the component "Escape" and "Simulation" and all their connections are destroyed. This plan will be sent to the intermediate agent which orders to its reactive agents related to these components and connections to begin the destruction. These agents will be themselves destroyed thereafter (Figure 13-d). The components "Scan" and "planning" will be connected by the reactive agent (Figure 13-e). All of these modifications are transmitted to the superior agent.

5.4. *The creation of new connectors for new planning of navigation:* the "Planning" component defines new plan to reach the finale point. The component "Planning" emits an event towards the platform (Figure 13-f) so that new connector will be created to connect component "X Approach" to component "Planning" (Figure 13-g) with the aim to reactivate the approach on X axis. The event is collected by the reactive agent and is sent towards its intermediate agent which will identify the new event, and send it towards the superior agent. This agent will establish a new plan. In this way the mobile robot will start its movement according to the X approach, then it will reach the finale point by an Y approach.

6. A real application

After we have established a Pipe-and-Filter architecture for the navigation of a mobile robot in an environment with obstacle, we have programmed an application (Figure 14) which shows well how the mobile robot move on the our simulator. The user has a user-friendly and intuitive interface for various simulations. Thus, it can parameter the initial and final position of the robot as well as the position of the obstacle on the screen of our simulator and also the range of the sensor.

During simulation, the user can choose different architectures (with or without multi-agents platform). The importance of our platform in the maintenance of the dependability and performance in any circumstance, is well illustrated in the Figure 15. Without the intervention of our platform the robot crash on the obstacle. In Figure 16, we can see that the initial Pipe-and-Filter architecture is modified by our platform. During the simulation the robot detects the obstacle, and the architecture is dynamically reconfigured, so that the mobile robot avoids the obstacle and reaches the finale point. The user can parameter in the "Scan" component the range of the sensor via the platform. If the user raises the range of the sensor then during the simulation the robot detects earlier the obstacle on its trajectory.

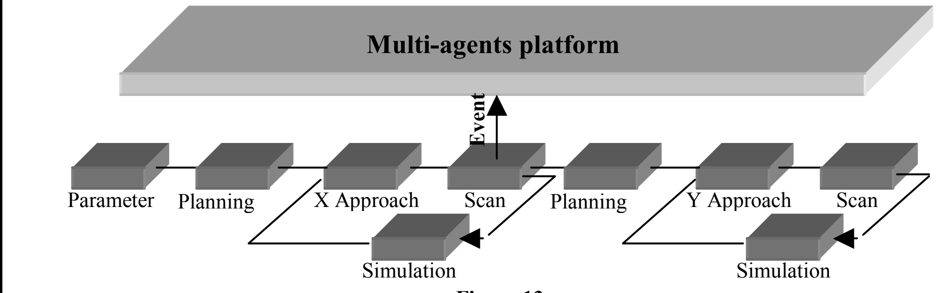


Figure 13-a

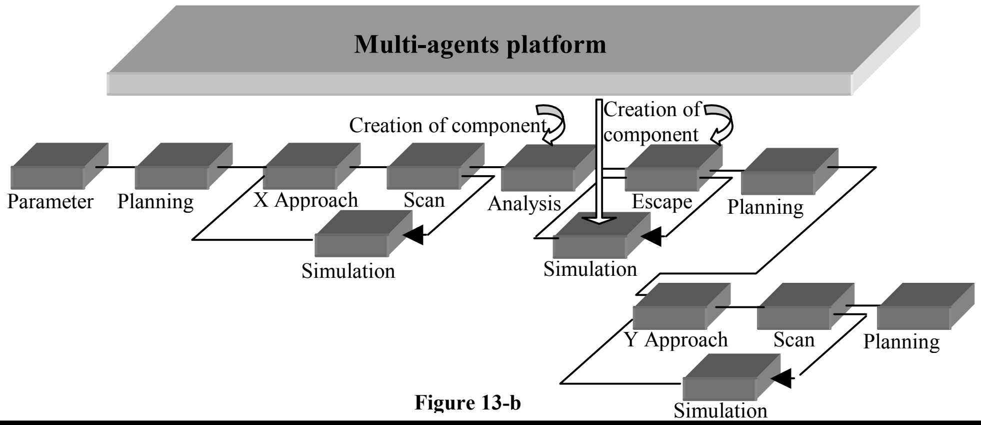


Figure 13-b

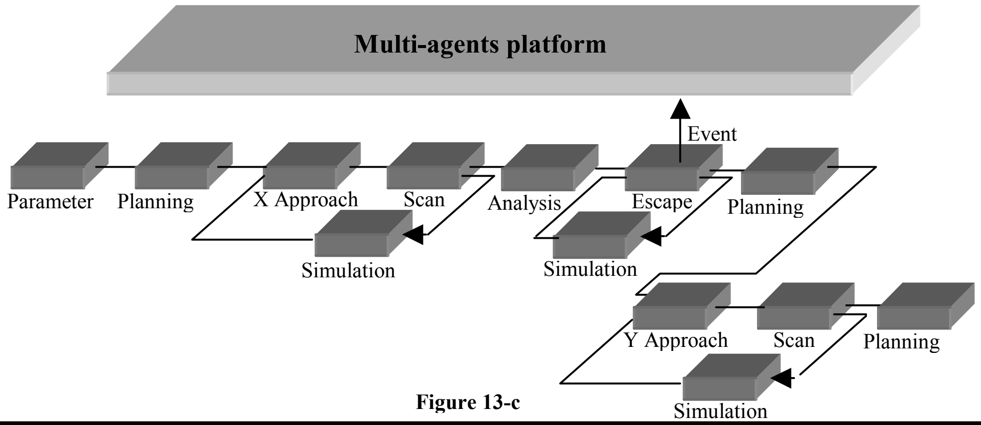


Figure 13-c

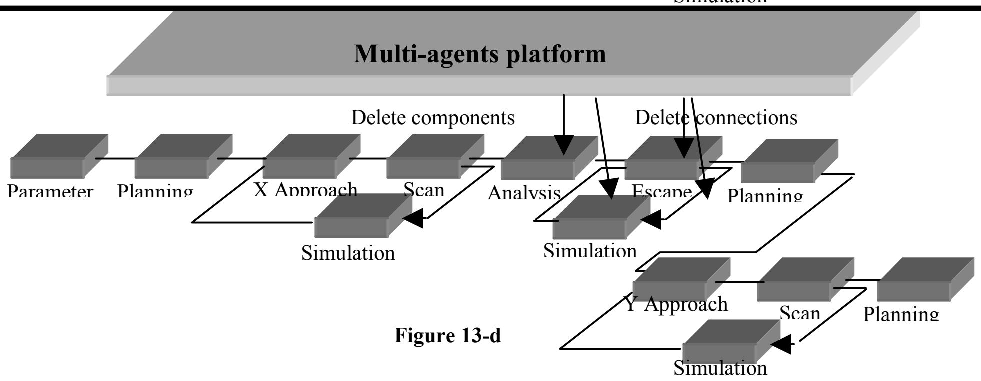


Figure 13-d

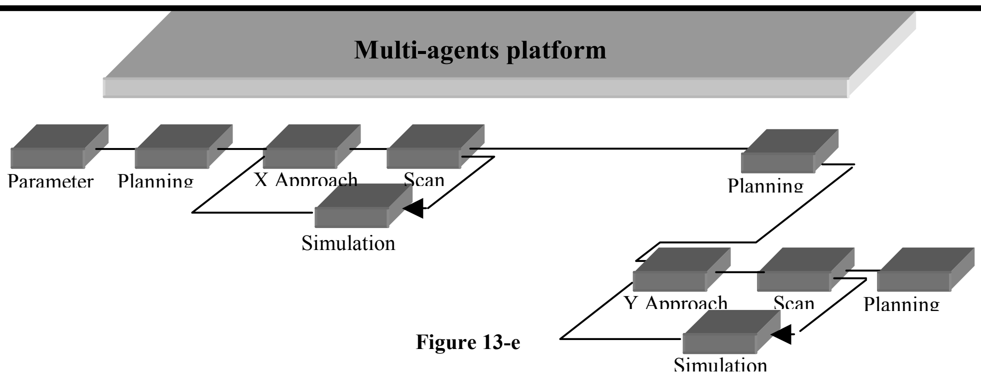


Figure 13-e

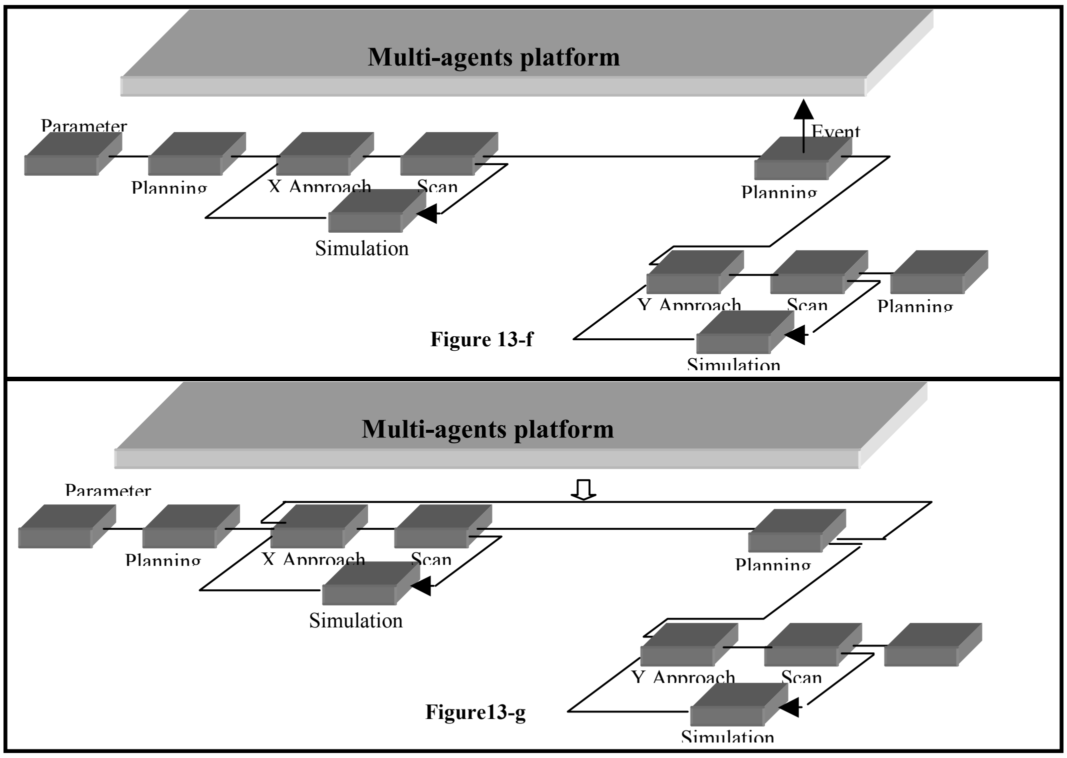


Figure 13 : Scenario of navigation of the mobile robot on an environment with obstacle

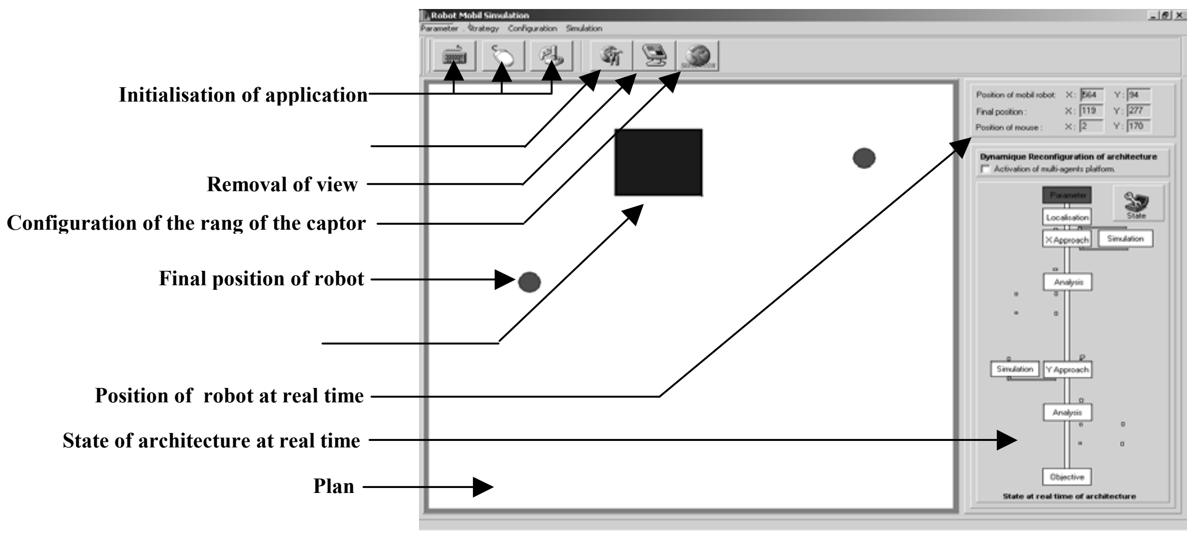
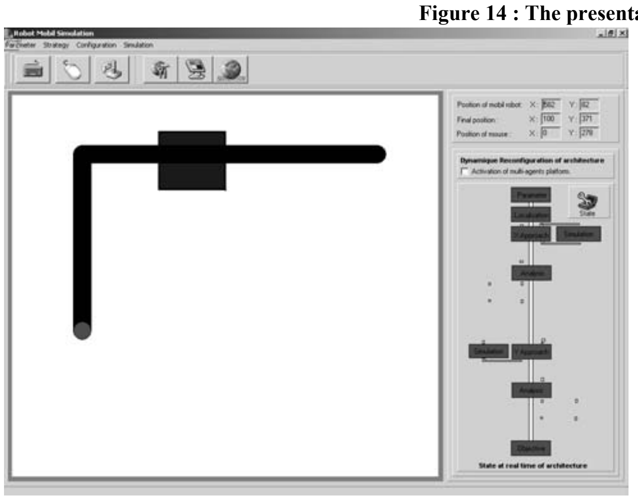


Figure 14 : The presentation of the simulator



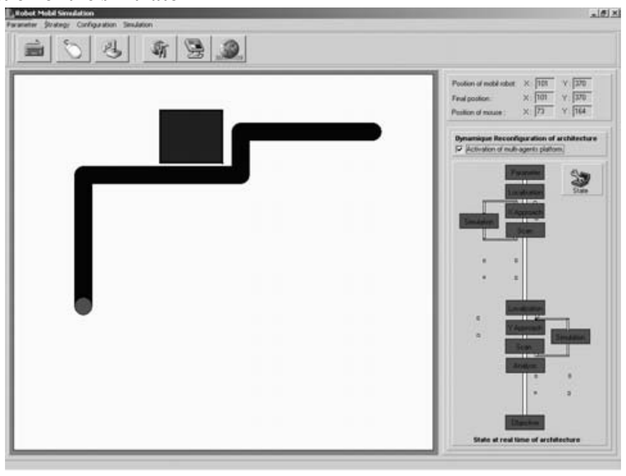


Figure 15: The crash of the robot on the obstacle without using our platform

Figure 16 : The mobile robot avoids dynamically the obstacle by using our platform

7. Conclusion

The right architecture is the first step to success. The wrong architecture will lead to calamity. We can identify causal connections between design decisions made in the architecture and the qualities and properties that result downstream in the system or systems that follow from it. This means that it is possible to evaluate an architecture, to analyze architectural decisions, in the context of the goals and requirements like variability management that is levied on systems that will be built from it. The architecture then becomes the basis of systematic development and evolution of software/hardware systems. It is clear that a new architecture that permits the dynamism reconfiguration while ensuring the use of software in multiple contexts and the ability of software to support evolution and changing requirements in various contexts are needed. This paper presents a new platform based multi-agents which monitors the global architecture of a system and manages the provided variability. It will achieve its functional and non functional requirements and evaluate and manage changes in such architecture dynamically at the execution time. In this paper we have developed our generic platform and we have applied and implemented it on the Pipe-and-Filter architecture. This software/hardware architecture is used for controlling a mobile robot to follow a trajectory towards the desired position in the presence of obstacles. We have showed by some scenarios the dynamic reconfigurations related to the improvement of the variability management through the structuring investigation of fault-tolerant component-based systems at architectural level of Pipe-and-Filter style. Our approach can be extended to deal with other architectural “non-functional” quality attributes in the context of developing complex and reliable systems.

References

1. M. Shaw, D. Garlan, Software Architecture, Perspectives on Emerging Discipline,
2. Prentice-Hall, Inc. , Upper Saddle River, New Jersey, 1996.
3. D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, Software Engineering Notes, 17(4):40, Oct. 1992.
4. B. Randell and J. Xu, The evolution of the recovery block concept, In software fault tolerance, chapter 1. John Wiley sons ltd. 1995
5. M. Sloman and J.Kramer, Distributed systems and computer networks. Prentice hall. 1987
6. D. Sotirovski. Towards fault tolerance software architectures. In R. Kazman, P. Kruchten, C. Verhoef, and H. Van Vliet, editors. Working IEEE/IFIP Conference on software architecture workshop, pages 7-13, Los Alamitos, CA, 2001.
7. P. Asterio de C. Guerra et al. An Idealized Fault-Tolerant Architectural Component, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
8. S. S. Gokhale and al. Integration of Architecture Specification, Testing and Dependability Analysis, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
9. R. De Lemos and al. Tolerating Architecture Mismatches, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
10. M. S. Dias and D. J. Richardson, The role of Event Description in Description in Architecting Dependable Systems. In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
11. B. Shroeder, On-line monitoring, IEEE Computer, vol. 28, n. 6, June 1995. pp. 72-77.
12. R. Snodgrass, “A Relation approach to monitoring complex systems”, ACM Trans. Computer Systems, vol. 6, n. 2, May 1988, pp. 156-196.
13. C. Liu and D. J. Richardson, Architecting dependable systems through redundancy and just-in-time testing. In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
14. M. Brandozzi and D. E. Perry, Architecture prescription for dependable systems, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
15. L. Bass, P. Clements and R. Kazman, “Software architecture in practice” SEI Series, Addison-Wesley. January 1998.
16. A. Ramdane-Cherif, N. Levy and Francisca Losavio. Dynamic Reconfigurable Software Architecture: Analysis and Evaluation.. In WICSA’02: The Third Working IEEE/IFIP Conference on Software Architecture. Montreal, Canada, August 25-31, 2002.
17. P. Sargent. Back to school for a brand new ABC. In: the guardian, 12 March 1992, p28.
18. Ovum Report. Intelligent agents : the new revolution software, 1994.

Software Testing Requires Variability

Henrik Bærbak Christensen
Department of Computer Science
University of Aarhus
DK–8200 Aarhus N
Denmark
hbc@daimi.au.dk

1 Motivation

Software variability is the ability of a software system or artefact to be changed, customized or configured for use in a particular context. Variability in software systems is important from a number of perspectives. Some perspectives rightly receive much attention due to their direct economic impact in software production. As is also apparent from the call for papers these perspectives focus on qualities such as *reuse*, *adaptability*, and *maintainability*.

However, the wish for introducing variability points into software systems can also come from sources that are less directly coupled with economic and end-user aspects but more coupled to the development process itself. One source it the wish for high quality software through test-driven software development as advocated by eXtreme Programming [1]. We will explore this perspective in this position paper.

In test-driven software development, the development of functionality and tests are intertwined in an iterative, short-cycled, development process. The developed test cases are maintained throughout the lifetime of the product and are run very often to ensure that functionality introduced does not invalidate the functionality of the existing code base. Systems developed this way "grow functionality", as functionality is added in an incremental fashion where each addition ideally leads to a "micro release" that is limited but operational and is able to be evaluated by the customer.

The requirement to keep the test cases running at all times puts high demands on the existing code base. Adding functionality often introduce small changes to the code base that invalidates the test cases—often in trivial but still costly ways. As a trivial example, it may be necessary to add a parameter to a method signature of some class in the existing code base. While the cost of this may in itself be low, a higher cost is usually associated with rewriting the test cases—a major reason why tests have a tendency to become invalid and hence useless.

Another problem with testing is instrumentation of the product code base i.e. developers add code that is demanded to make the tests possible rather than demanded by product requirements. As an example, testing algorithms that depend on random number generation are tedious unless it is possible for the test cases to dictate the exact sequence of "random" numbers. The problem with this instrumentation is that product code easily becomes polluted with instrumentation code and even testing code. This enlarges the code base, may introduce faults on its own, and lowers maintainability. We therefore find it important to ensure that product code and testing code is completely decoupled so that no testing code or instrumentation code can be found in the product code.

Both examples demonstrate small but recurring problems. They therefore introduce expenses associated with test-driven approaches that accumulate if not taken care of.

In our view, these problems are best faced by perceiving them not as testing problems but as a question of introducing variability into the code base. This viewpoint has several benefits both for the testing aspect as well as for the resulting product code base. And, it adds yet another perspective to the concept of variability.

2 Variability Points for Testing

You can perceive the problems of maintaining test cases and instrumentation product code as variability problems.

In the first case, the aim is to reduce the number of changes in the product code base that require changes in the testing code. As we implicitly assumes a testing approach biased towards black box testing this means that the contracts of the product code should remain as stable as possible as the code base evolves. Good object-oriented modeling and design techniques are of course essential here, but introducing variability points using well known design patterns into the product code base is a key to further maintain stability. The characteristics of these variability points are that they allow the testing code to configure the product code appropriately for a certain test situation using the same variability points as is introduced to augment ("grow") the product code base with new functionality. Thereby the product code base contracts/APIs are stable. A small example is given below.

In the instrumentation case, the aim is to avoid introducing test specific code into the product code thereby polluting it. Here variability points can be non-intrusively introduced into the product code base that serves as hooks for the testing code. Thus the testing code can create hook instances that when inserted into the product code's variability points can monitor internal state (to aid e.g. white-box testing) and/or force certain conditions by manipulating state information to test special cases that is otherwise difficult to set up. In the product these variability points will contain no-operation hooks, unless they can be used to

3 Central Design Patterns

We have some experience with the sketched approach, primarily from teaching at the university level. While the scale of implementation effort tackled in a teaching context is necessarily much smaller than those faced in an industrial context, we still do not find that this invalidate the premises on which we draw our conclusion. The problems we look into appear at the unit testing level and at the individual variability point level. The problem of scale primarily shows when the number of variability points grows large.

Below is a list of design patterns that are very helpful (all from the Gang of Four book [2] except when cited):

- *Abstract factory* allows us to define a number of hook instances for a set of variability points using one object, the factory. This is important because it allows the product code contract/API to stay intact even though new variability points are introduced. You of course need to add a new creator method in the factory as well as introduce calls to the hook instance from the product code but these are simple operations that only minimally influence the product code base.

- *Mediator* combined with *strategy* or *state* allows us to partition and delegate logical tasks in the product to "handlers" or "managers" that only collaborate through the mediator. Typically the concrete handlers are instantiated by an abstract factory, allowing the testing code to configure handlers.

- *Null object* [3] is important because it allows us to "turn off" unwanted functionality/strategies in the product code from the testing code. For instance adding advanced functionality may influence testing of basic functionality in unwanted ways. If this influence is constrained to be done through invoking methods on appropriate handlers/strategies then the testing code can instantiate a null object strategy for the advanced functionality and thus eliminate the unwanted side effects for the basic testing.

3.1 Example

To quantify some of the above abstract points of view, a small example may help to illustrate. The case is an implementation of the board game Backgammon that has served as compulsory exercise in a university-level programming course.

A domain model of Backgammon is complex but can be grown naturally through a test-driven process. First basic abstractions are made operational: players, dice, points, checkers, bars, and basic movement. Second comes the complex subject of validating moves. Third, one needs logic to control the logical flow of the game. And finally, if a graphical user interface is required an appropriate coupling must be defined and implemented.

At the basic level, you want to test basic movement of checkers on the points—essentially viewing the board as a structured collection of checkers. As we have not yet introduced validation according to the backgammon rules, this may include tests that accept that both red and black player have checkers on the same point. Moving to the phase where the game is augmented with move validation this test must either be removed, rewritten, or (as we propose) view validation as a variability point by designing it as a validation strategy that the move code delegates to. Thus the basic level test code can instantiate a null validation strategy, a small change, and thereby keep all basic level tests intact.

Validation of moves depends on the dice thrown. Die throws are by nature random and systematic testing in a random world is, well, counter-productive. The test code needs to control the outcome of the dice to systematically test the validation code. Again, treating the die as a variability point is beneficial. We introduce a strategy, a die manager that the game utilizes. The test code can instantiate a subclassed die manager where it has the ability to dictate the sequence of die values thrown. Thus no changes are introduced in the product code at all while we still retain all the power of control in the testing setup.

The small examples also show how tests influence the product code design towards modular and compositional design. The "trick" that allows disabling the validation code by making it a variability point has laid the way for introducing alternative sets of rules simply by substituting the validation strategy. While there seems less use for alternative dice it is still an example that the testing perspective promotes good programming practice, namely to isolate specific and well-defined responsibility into separate components.

4 Discussion

One may argue that introducing variability points in the product base as a tool to aid testing quickly leads to a unmaintainable situation as the product code essentially becomes polluted with them. This is partly right, and one of the reasons that variability management is very interesting also from the perspective advocated within this paper. Participation in the workshop will hopefully provide insight that makes the perspective more scalable.

One plausible technique that addresses the scalability problem to some extent is the use of appropriate design patterns, for instance *abstract factory* essentially allows the definition of a large number of hook instances to be grouped and the product codes contract to stay intact, as outlined in the discussion above.

We have found that the testing perspective provides a first benchmark for evaluating adaptability and flexibility of a product code base as it essentially is a first instance of reuse and adaptation.

We have also found that testing via variability promote code designs that are highly modular, compositional, and adaptable.

About the Author

Henrik Bærbak Christensen is assistant professor at Department of Computer Science, University of Aarhus, where he also received his Ph.D. His research interests are software architecture, design patterns and frameworks, software configuration management, object-oriented techniques, and teaching.

References

[1] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[3] B. Woolf. Null Object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18, 1997.

Timeline Variability: The Variability of Binding Time of Variation Points

Eelco Dolstra
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Gert Florijn
SERC, P.O. Box 424,
3500 AK Utrecht, The Netherlands
florijn@serc.nl

Eelco Visser
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
visser@cs.uu.nl

1. Introduction

Timeline variability is the ability of a software system to have variation points bound at different moments of the system's life-cycle.

Virtually every non-trivial software system exhibits *variability*: the property that the set of *features*— characteristics of the system that are relevant to some stakeholder— can be changed at certain points in the system's life-cycle. The parts of the system that implement the ability to make such changes are called *variation points*. Selecting some variant supported by a variation point is called *binding* the variant. Every variation point has at least one associated *binding time*: the moment in the system's life-cycle at which the variation point can be bound. A more detailed exposition of this terminology can be found in, e.g., [7, 2].

For example, the decision to build an operating system kernel with multiprocessor support, or to build a "light" or "professional" version of a word processor, might be implemented at build time. On the other hand, the decision to include support for some brand of hard drive in an operating system, or to use some particular language for spell checking in a word processor, might be made at runtime.

Generally, one would like variation points to be as flexible as possible with regard to binding time. That is, ideally one wants to have the ability to bind a variation point at build time, installation time, runtime, and so on. This leads to the notion of *timeline variability*: that certain features can be bound at *several* stages of the life-cycle. We do not formalise the term *timeline* here. Intuitively, we use if to refer to the set of distinguished moments during the build and deployment process where a user can potentially select variants. For example, the Linux operating system kernel allows functionality, e.g., device drivers, to be included either at build time or at runtime. However, changing features at runtime proceeds through entirely different interfaces than changing them at build time. Similarly, the Apache `httpd` webserver allows server extensions to be included at build time or at load time, but through different configuration mechanisms. Microsoft Office 2000 allows components to be installed either at install time proper or on demand, at runtime.

The concept of timeline variability—that is, *variability of binding time*—should not be confused with the binding time of variation points. In this paper we illustrate timeline variability through two case studies, Apache and the Linux kernel, and show that the two main technical issues in timeline variability are *inconsistent configuration interfaces* and *ad hoc implementation mechanisms*. We also provide some directions for future research.

2. Examples

In this section we show some examples of timeline variability in real systems. As we shall see, implementation of such variability is problematic. Consider, for example, a binary variation point that is bound at runtime, implemented in C. This is not hard to implement:

```
if (feature) f() else g();
```

Moving this variation point to build time is not hard either using conditional compilation:

```
#if FEATURE
  f()
#else
  g()
#endif
```

But suppose we wish to allow for this feature to be bound both at build time and runtime. A possible implementation would be:

```
#if FEATURE_BOUND_AT_BUILD_TIME
#if FEATURE
 f()
#else
  g()
#endif
#else
  if (feature) f() else g()
#endif
```

which is not very elegant. For more complex variation points, the situation becomes even worse.

2.1. The Linux kernel

The Linux kernel provides the basis for several variants of the GNU/Linux operating system. The kernel's job is to virtualise the hardware (e.g., provide multitasking and virtual memory) and abstract from it (e.g., provide a unifying interface to different types of storage devices or file systems).

The Linux kernel was originally implemented as a traditional monolithic kernel. In this situation all device drivers are statically linked into the kernel image file. Conditional defines and makefile manipulation are used to selectively include or exclude drivers and other features.

The disadvantage of this approach is that it closes a large number of variation points at build time. Hence, the kernel was retro-fitted with a *module* system. A set of source files constituting a module can be compiled into an object file and linked statically into the kernel image, or compiled into an object file that is stored separately and may be dynamically linked into a running kernel. Modules may refer to symbols exported by other modules. A tool exists to automatically determine the resulting dependencies to ensure that modules are loaded in the right order.

The implementation of the variation points realised through the module system is for the most part straightforward. For example, operations on block or character device files are implemented through dispatch through a function pointer; this is a feature of standard C. However, these function pointers must at some point be *registered* (i.e., be made known to the system), and this cannot be done in standard C. In particular, every module exports an initialisation function which must be called during kernel initialisation, in the case of statically linked modules, or at module load time, in the case of dynamically loaded modules. The C language, however, does not provide a mechanism to iterate over a set of function *names* that are not statically known. For example, we have no way of calling every function called `init_module()` that is linked into the executable image.

The Linux kernel solves this problem through the technique of emitting certain data in specially designated *sections* of the executable image. An invocation of the macro `__initcall(`f`)` arranges for the address of f to be placed in the special section `.initcall.init`;

```
typedef int (*initcall_t)(void);

#define __initcall(f) \
  static initcall_t __initcall_##f \
    __attribute__((unused,__section__\
      (".initcall.init"))) \
    = f
```

A module can declare some initializer f by invoking the macro `module_init(`f`)`. For statically linked modules, `module_init` expands to an invocation of `__initcall`, and so the address of f is emitted in the `.initcall.init` section. We can then iterate through all initialisers as follows:

```
initcall_t *call = &__initcall_start;
do {
  (*call)();
  call++;
} while (call < &__initcall_end);
```

The symbols `__initcall_start` and `__initcall_end` are emitted at the start and end of the `.initcall.init` section by the linker script that guides the linker.

For dynamically loaded modules, on the other hand, `module_init(`f`)` emits a symbol `init_module` as an alias for f. The module loader will simply look this symbol up and call it.

Hence, we achieve timeline variability of module activation extending to build time and runtime, through a combination of preprocessor, compiler, and linker magic.

Cross-cutting features One problem facing the scheme implementing the module system is that it is closely tied to the structure of source modules; it is therefore difficult to modularise features that are not localisable into one or a few distinct source modules, i.e., cross-cutting features. An example is whether the kernel is built for uniprocessing or for symmetric multiprocessing (SMP). In an SMP configuration, many kernel data structures have to be guarded carefully against concurrent access; this affects a large amount of code. Quantitatively, we can get an indication of the degree to which a feature cross-cuts a system by counting the `ifdef`s conditionalised on the feature variable. In this case, we see that `#ifdef CONFIG_SMP` occurs more that 540 times in 250 source files of version 2.4.10 of the kernel. Because they impact so many source components, cross-cutting features are not very well suited for dynamic loading. Additionally, variation points such as SMP support affect the definition of data structures, which makes it practically impossible to bind them at any time later than build time.

Analysis A problem of the Linux kernel is its monolithic distribution. If a feature is required that is not part of the distribution, either the kernel must be patched (e.g., the JFS file system) or the code must be compiled separately, outside of the kernel source tree (e.g., the ALSA sound system). Note that the latter solution makes static linking into the kernel impossible, the build mechanism is totally different, and it creates more work for users. Dynamic source tree composition [3] can alleviate this problem.

Note that the timeline variability of the module system does not directly extend to startup time, i.e., the loading of the kernel, since the kernel may not have the ability to load kernel modules at boot time. For example, the modules supporting the storage medium and file system on which the modules are stored must be statically linked into the kernel to prevent a chicken-and-egg problem. In essence, the timeline variation point has been closed with respect to startup time by the problem domain. However, an *initial ramdisk* (which is part of the kernel's image) may be used to store the required modules, thus extending the timeline variability to startup time.

2.2. Apache

The Apache `httpd` server is a freely available web server. In order to support various kinds of dynamic content generation, authentication, etc., the server provides a module system. Modules can be linked statically, or dynamically, at startup time. Dynamically loaded modules can be compiled inside or outside the Apache source tree.

Apache faces the same problem as the Linux kernel: how to register a variable set of modules (i.e., how to make statically included modules known to the core system)? The solution used by the Apache developers is to have the configuration script generate a C source file containing a list of pointers to the module definition structures:

```
module *ap_preloaded_modules[] = {
  &core_module,
  &access_module,
  &auth_module,
  ...
};
```

Note that this solution is again, in a sense, outside of the C language; we need to *generate* C code (i.e., externally) in order to deal with these open variation points.

Analysis Note that neither Apache nor the Linux kernel take advantage of static linking beyond the fact that it may be a necessity, e.g., dynamic linking may not be available on some platforms on which Apache is configured, provides simplified runtime characteristics, or, in the case of the Linux kernel, may be perceived as a security feature (the absence of dynamic loading of kernel modules makes it a little bit more difficult to subvert the kernel). Compile time knowledge of the module configuration does not lead to more efficient code, since this requires cross-module optimisation; many C compilers are not capable of this.

2.3. Issues

So what are the issues in timeline variability? First, though some features can be bound at several moments during the life-cycle, the configuration interfaces tend to be different for each moment. For example, in the case of the Linux kernel, a module may be included at build time through the use of an interactive configuration tool that shows variants, dependencies between features, and so on. On the other hand, including a module at runtime happens by running the `modprobe` command; an entirely different interface. Likewise, Apache modules can be added at build time through a Autoconf `configure` script, or at startup time by editing a configuration file.

Second, the techniques used to implement timeline variability are *ad hoc* necessarily because the underlying languages do not offer the required support. Providing a variation point *either* at build time *or* at runtime is not hard, but providing it at both requires quite a bit of "magic".

3. Future Work

We have seen that timeline variability causes difficulties at two different levels, namely, in the *implementation* and in the *configuration* of the system.

Implementation The main implementation issue is that variation points are not first-class citizens in conventional programming languages and development environments, that is, they are not represented explicitly and cannot be manipulated directly. Rather, the implementation of a variation point happens through some mechanism that is specific to the binding time, e.g., conditional compilation or dynamic loading of shared libraries. This means that moving a variation point to a different binding time, or supporting binding at multiple binding times, requires explicit and often non-trivial modification to the system.

A partial solution to this problem is the use of *staged compilation*. For example, partial evaluation may be used to move an apparent runtime variation point to build time.

The converse—moving from build time to runtime—is generally harder. For example, it is not obvious how to deal with conditional data structure definitions.

Configuration The main problem here is that every stage in the life-cycle tends to present a different configuration interface to the user. This is particularly annoying for variation points that have several binding times. In the *Transparent Configuration Environments* (*TraCE*) project we aim at generalising system configuration interfaces. TraCE consists of a generic configuration interface parameterised with a formalised feature model.

In approaches such as FODA [4] feature models are described as graph-like structures, where the edges between features denote certain relationships such as alternatives and exclusion. The model therefore describes a set of *valid* configurations that satisfy all constraints on the feature space. Apart from being used during analysis and design, such models can also be used to drive the configuration process directly. For example, the CML2 [5] language was designed to drive the configuration process of the Linux kernel on the basis of a formal feature model of the system.

However, these models provides a *static* view of the configuration space: a configuration is either valid or it is not; no timeline aspects are taken into account. In order to model timeline aspects, it is necessary to take into account that some feature selections, i.e., bindings of variation points, are valid only on certain points on the configuration timeline. Therefore, the feature model presented in this section does not place constraints on configurations, but rather on transitions between configurations.

Formally, a feature model for a system with a statically fixed set of variation points has the following elements:

- A set of named variation points P and, for each variation point $p \in P$, the set of named states S_p.
- A *configuration* C is a mapping from variation points to states, that is, a function $P \rightarrow \cup_{p \in P} S_p$.
- An initial configuration $c_0 \in C$.
- A relation $T \subseteq C \times C$ expressing valid configuration transitions; i.e., it constrains configurations. As noted above, it is not sufficient merely to describe valid configurations, since not every valid configuration can be transformed into any other valid configuration. However, the set of valid configurations follows by computing the transitive closure of the set $\{c_0\}$ under the T relation.

Note that static feature models such as FODA [4], FDL [6], and CML [5] can be transcoded into this model; they are just different ways of expressing the valid-transition relation T. Indeed, the main problem in making this approach useful is to find a suitable way to specify T. Note that this is just a usability issue; the model is as described above.

It may be argued that implementation restrictions should not appear in the feature model (e.g. in [1], p. 117). However, they are required to generate configuration systems. In addition, we can identify several types of constraints. First, there are constraints that are inherent to the problem domain; these arise from the domain analysis. Second, some constraints result from implementation restrictions. This may well be the largest set in typical systems. Finally, some constraints are not forced by the domain or implementation, but rather are added by some stakeholder. (for example, a system administrator restricting some end-user configurability). The specification language for the feature model should allow these constraints to be specified separately.

4. Conclusion

Timeline variability makes the configuration of software systems more flexible by leaving open the decision about the binding time of a feature. However, the implementation of timeline variability is often ad hoc and presented through inconsistent configuration interfaces. Better support for timeline variability requires features models to describe the variability of a system *including* its timeline variability, and *transparent configuration environment* which provide an abstract interface to the details of configuration mechanisms required

References

[1] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[2] L. Geyer and M. Becker. On the influence of variabilities on the application-engineering process of a product family. In G. J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Science*, August 2002.

[3] M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[5] E. S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *9th International Python Conference*, March 2001.

[6] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. Submitted.

[7] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceeedings of WICSA 2001*, August 2001.

Enabling Reconfiguration of Component-Based Systems at Runtime

Jasminka Matevska-Meyer and Wilhelm Hasselbring
Department of Computing Science, Software Engineering Group
University of Oldenburg, Germany
{matevska-meyer, hasselbring}@informatik.uni-oldenburg.de

1 Introduction

The development of software systems iterates over analysis, design, implementation, and deployment. Subsequent iterations require refactoring [2] of design and reconfiguration of deployed systems. At least three software engineering disciplines are involved when dealing with runtime reconfiguration of component-based software systems:

- software architecture,
- software configuration management, and
- software component deployment

These disciplines contribute in various ways. Software architectures play a central role at design, describing a system model and specifying it in a formal way using some architecture description language [7]. Configuration management focuses on implementation, defining a configuration from various component versions and building a system from this configuration [6]. Component deployment addresses the deployment phase, managing all dependencies among the involved components and eventually producing a running system [1, 12].

Although these three activities may evolve independently and provide their own models of the system, they are all involved when reconfiguration is required (roundtrip engineering). Applying planed changes to a deployed system usually triggers changes in all those system models to obtain a consistent system after reconfiguration. A major problem to be solved here is managing (run-time) dependencies among the components. Therefore, we need a formal system model, which covers components, their interconnection, communication, and run-time behavior, integrating all the system models of software architecture, configuration management and component deployment [13].

2 An Approach to Enabling Reconfiguration of Component-Based Systems at Runtime

We aim at Reconfiguration of Component-Based Systems at Runtime. Our proposed approach employs:

- Parameterised Contracts [8] as a method for formal component specification, adding a formal run-time component description technique,
- using graphs [5] to describe dependencies among components and considering run-time concerns,
- extending C2-ADL [11] with a concept of containers to establish modelling of a deployment and runtime properties of a system,

This combination shall be the way to provide a foundation for achieving our goals. Figure 1 displays our suggested system configuration. A system configuration is designed as a hierarchy using three GoF design patterns [3]: *composite, decorator and adapter*.

- Composite is required to build a system configuration,
- The Decorator pattern allows functional changes to components,
- The Adapter pattern (wrapper) allows changes of their interfaces

Furthermore the concept of containers allows us to manage the process of run-time reconfiguration as *run-time re-deployment* of components.

Our Reconfiguration Manager (a special type of connector) is activated on every *reconfiguration request*. It consists of:

- Reconfiguration Analyzer
- Dependency Manager
- Consistency Manager
- Reconfigurator

The Reconfiguration Analyzer takes a *reconfiguration request*, analyzes and classifies the requested change. Our Dependency Manager monitors the run-time dependencies among components, determines a minimal set of change-affected components and sends a *change request* for each

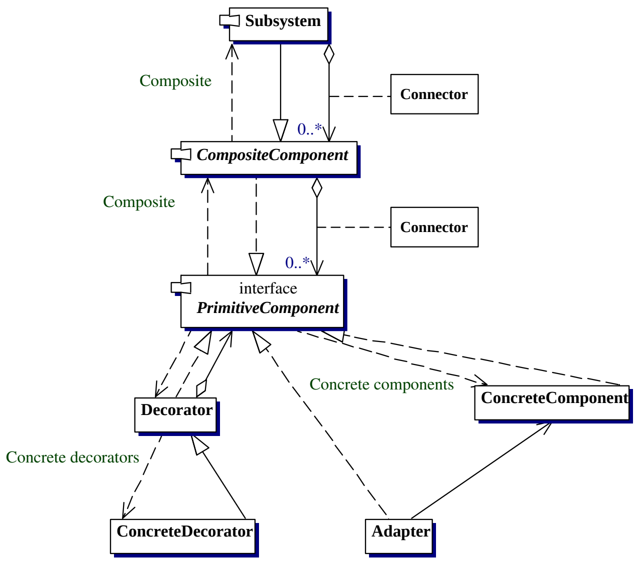


Figure 1. System Configuration

involved component to the reconfigurator. The Consistency Manager controls the system. We divide its activities into:

- Pre-Reconfiguration: checking the static consistency of the intended system configuration and moving a consistent system into a *ready-to-configure-state*, or refusing the reconfiguration request on failure.
- Post-Reconfiguration: checking the (run-time) consistency of a changed system and, on success, confirming a reconfiguration, or sending a *rollback request* to the reconfigurator.

The Reconfigurator realizes the reconfiguration as a dependent change transaction [4]. It starts a transaction, transfers all affected components into a *blocked* state, isolates an affected subsystem, applies the changes, and sends a *consistency-check-request* to the consistency manager. On success it commits the transaction, on failure it initiates a rollback and transfers the changed or unchanged system into a running state.

Figure 2 displays all states a component can take at system runtime. Just after it has been deployed we assume that it is *free*. We distinguish between the states *busy*, which means *is in use* and *active*, which means *is executed*. Therefore, a component can't directly move into a state *active & busy*. Only free components can be transferred into a *blocked* state and be changed afterwards. This means, our reconfiguration takes place while the system is running, we are not trying to achieve an ad-hoc component change.

We assume that a (sub)system can take only four states at runtime: *running, ready to configure, reconfiguring and restoring* (Figure 3). For each state a corresponding part of the reconfiguration manager initiates and controls possible changes from one state into another.

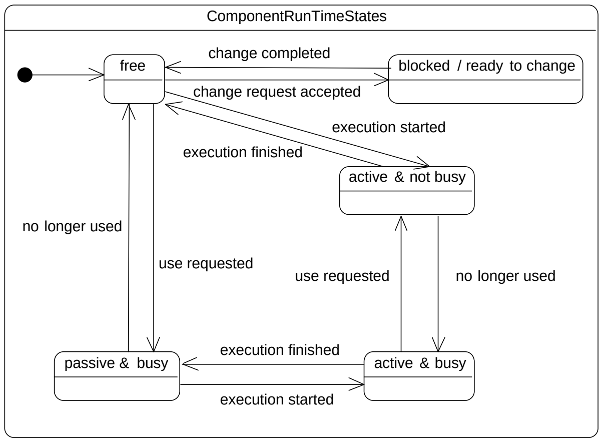


Figure 2. Component Run Time States

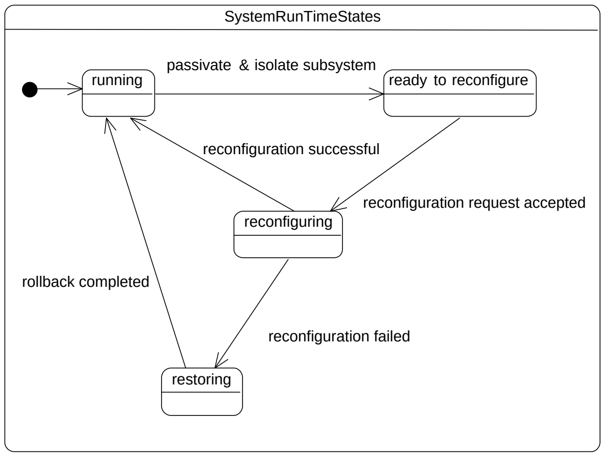


Figure 3. System Run Time States

3 Summary

We present an approach to enabling reconfiguration of component-based systems at runtime. This approach combines the disciplines software architecture, configuration management and component deployment.
As an implementation platform we are using J2EE-Technology [10]. We are intending to extend its Specification of the deployment process with a subprocess of *reconfiguration* [9]. Currently, we are investigating the possibilities to control or manipulate the deployment process at different application servers and develop a methodology for determining and formally specifying dependencies among already deployed components.

References

[1] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report 857-98, Department of Computer Science, University of Colorado, Apr. 1998.

[2] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Object-Oriented Technology. Addison-Wesley, Massachusetts, 1995.

[4] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[5] M. Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. PhD thesis, Uppsala University, Sweden, Dec. 2001.

[6] M. Larsson and I. Crnkovic. Configuration management for component-based systems. In *Proceedins of the Tenth International Workshop on Software Configuration Management*, Toronto, Canada, May 2001.

[7] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[8] R. H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. PhD thesis, Universität (T. H.) Karlsruhe, 2001.

[9] M. J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Reconfiguration in the Enterprise JavaBean component model. In J. Bishop, editor, *Proceedings of IFIP/ACM Working Conference on Component Deployment*, pages 67–81, Berlin, Germany, June 2002. Springer-Verlag Berlin Heidelberg.

[10] Sun Microsystems. *Java 2 Platform, Enterprise Edition Specification, Version 1.3*, 2002.

[11] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[12] A. van der Hoek, R. S. Hall, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Software deployment: Extending configuration management support into the field. *CrossTalk The Journal of Defense Software Engineering*, 11(2):9–13, Feb. 1998.

[13] A. van der Hoek, D. Heimbigner, and A. L. Wolf. Software architecture, configuration management, and configurable distributed systems: A ménage a trois. Technical Report 948-98, University of Colorado, Department of Computer Science, Software Engineering Research Laboratory, Colorado, 1998.

Supporting Evolution in Software using Frame Technology and Aspect Orientation

Neil Loughran, Awais Rashid
Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{loughran | awais} @comp.lancs.ac.uk

Abstract

This paper discusses how the problems involved in supporting evolution in software can be resolved by using aspect oriented programming and frame technology. Throughout the lifetime of a software system, new requirements may arise that will require the existing system to be altered or evolved in someway. Evolution is something which is almost impossible to predict at the design stage. Although it is common to anticipate future evolutions and therefore prepare and design our code to accommodate this, there will eventually come a time when a certain feature or scenario appears where this may not be practical.

1. Introduction

Throughout the lifetime of a software system or architecture, new requirements may arise that will require the existing system to be altered or evolved in someway. Therefore an effective mechanism for evolution is an important factor in the creation of software systems. It is estimated that up to 80% of lifetime expenditure on a system will be spent on maintenance and evolution. However, achieving effective evolution across the board with current technologies is difficult because of the complexities involved.

Evolution is something which is almost impossible to predict at the design stage. Although it is common to anticipate future evolutions and therefore prepare and design our code to accommodate this, there will eventually come a time when a certain feature is required or a scenario appears where this may not be practical.

2. Background

2.1 Categories of evolution

Software evolution and maintenance can be divided into the categories shown in Table 1, which are derived from [6].

Table 1. Traditional categorisation of evolution

Category	Description / Example
Corrective	Fixing of bugs
Adaptive	Addition of new features Changing of functionality Support for new platforms
Perfective	Improving system functionality Improving performance
Preventative	Preventing problems before they occur

It should be noted here that any evolution made to a system could fall into one *or more* of the categories shown. For instance perfective evolution where, for example, the performance of a particular component needs to be improved, may also require other components of the system to be evolved thus requiring adaptive and possibly preventative evolution. Evolution of a particular component or feature may require other assets at different stages of the software lifecycle to also be evolved such as testing and documentation. This brings forward cases where evolution effectively crosscuts system structure *and* architecture. From this we can add two sub categories to the aforementioned, namely crosscutting and non-crosscutting evolution.

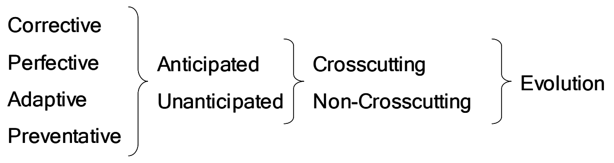


Figure 1. Evolution types

Another important notion is that of anticipated and unanticipated evolution. While anticipated evolutions can be obviously accommodated, unanticipated evolutions are of great concern if the system or architecture is to avoid erosion. Figure 1 illustrates the possible evolutions types.

Aspect orientation is designed to be used with conventional separation of concerns mechanisms, such as object-orientation, and should not be seen as a replacement for these techniques. It should be noted that the notion of aspect orientation now goes far beyond just programming level and is now being used at different levels of the software lifecycle such as the software design [7] [8] and requirements stages [9][10].

2.2 Crosscutting and separation of concerns

One of the principle requirements in software composition is to achieve a good level of separation between the different concerns in the system. By separation of concerns we mean the encapsulation of particular functional or non functional properties of the system which crosscut the system structure. This allows each concern to be viewed in it own space making system comprehensibility and manageability easier to understand thus facilitating reuse and evolution.

2.3 Software erosion

Erosion occurs when software, which has been continually evolved, eventually becomes difficult to understand, maintain and therefore evolve and reuse. When evolving a system we want to lessen the negative effects of the evolution in order to minimise the possibility for erosion. Erosion can occur anywhere from erosion of a particular component to the much larger problem of erosion in software designs and architectures. [1] cites cases where projects have had to be started from scratch as the source had become eroded beyond repair.

3. Approaches

3.1 Frame Based Technologies

Frame technology [2] is a concept that has its roots in the 1970s and was conceived by Paul G. Bassett as a means to providing *adaptive reuse*. By adaptive reuse we mean the process of creating generalised components that can be easily adapted or modified to different reuse contexts. From a simple perspective frame technology is a language independent textual pre-processor that creates software modules by using code templates and a specification from the developer. Variations of the technology inspired by Bassetts work such as XVCL [3] and FPL [4] use the XML language in order to implement the framing syntax. Frame technology works by organising frames into a hierarchy as shown in Figure 2, which depicts a partial view of a simple web browser.

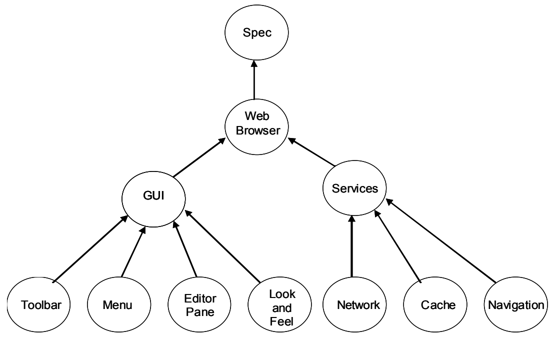


Figure 2. Example of frame technology hierarchy

Frames allow points of interest in the code, such as variation points, code repetition, configuration routines, optionality etc…, to be explicitly marked in place with metadata tags or moved to a *child* frame. By allowing these points of interest to be marked or modularised the developer can quickly create highly customisable systems. The basic granularity for a frame is the separation of a particular concern, class, method or related attributes with the hierarchy of frames serving to isolate content into separate layers, allowing the localisation of the effects of change and easing evolution. Usually the lower order frames are the most reusable as they contain less context sensitive information such as IO routines, library functions etc...

3.2 Aspect Oriented Programming

Aspect oriented programming (AOP) [5] technologies are now gaining popularity as a means for supporting the separation of concerns for features and constructs that would otherwise cause unmanageable code tangled across multiple classes in traditional object-oriented systems (Figure 3).

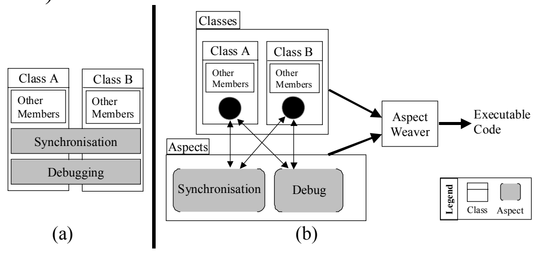


Figure 3. (a) Crosscutting concerns in OO languages (b) Separation of crosscutting concerns with AOP

Examples of the type of concerns that can cause this fragmentation of context are logging, profiling and tracing. Having all of the code for each particular concern modularised has the benefit of making system code easier to evolve, maintain and be reused hence increasing productivity, flexibility and reducing costs thus making them conducive for use within the software product line context.

There are numerous aspect oriented programming approaches available for use with the most well known being *AspectJ* [11], *Hyper/J* [12], and *composition filters* [13]. There also AOP approaches to *run time* evolution of programs such as *Java Aspect Components* [16] (JAC) and *JMangler* [17]. Run time evolution promises the facility for programs to be modified while they are executing. This facility will be of great importance to systems where stopping the system and evolving the code thus rendering the system from functioning is an undesirable characteristic from economic and safety perspectives. Examples of these systems could be 24/7 banking facilities, online commerce and air traffic control systems.

3.3 Other approaches

There are other approaches which seek to solve the problems associated with software product line issues notably Gen Voca [14] and work from the SEI [15]. However for the purpose of this paper we will only concentrate on frame based and aspect oriented approaches.

4. Supporting evolution

4.1 Evolution with frames

In section 2 we mentioned the notion of crosscutting and non crosscutting evolution. Non crosscutting evolutions are generally easy to solve with frame technology as their implementations are localised, the main problem being where the evolution might be spread out over many child frames spawned from the parent frame. In this sense the framing process can suffer from fragmentation of context.

Crosscutting evolution however, is not very effective with framing alone as there is no separation of concern mechanism beyond class and frame boundaries. For this reason aspect oriented technologies can play an important role in improving the evolution of systems which impart crosscutting behaviour.

4.2 Evolution with aspect orientation

Aspect orientation has been created with separation of crosscutting concerns in mind and thus would seem to be an ideal candidate for supporting the crosscutting evolutions that is difficult to achieve by framing.

However, while it is possible to use aspect oriented technologies alone to perform some form of evolution, it is constrained by the lack of configurability, generalisation and optionality that framing allows.

4.3 Hybrid approach

We have previously made a case where neither framing nor aspect orientation can support various evolutionary scenarios effectively in isolation. With this in mind it makes sense to combine the two technologies to improve on current techniques. Table 2 shows a comparison of the two techniques with their associated merits and demerits.

Table 2. Comparing frames and aspect orientation

Capability	**Framing**	**AOP**
Configuration Mechanism	Very comprehensive configuration possible	Not supported natively, dependent on IDE
Separation of Concern	Only non crosscutting concerns supported	Addresses problems of crosscutting concerns
Templates	Allows code to be generalised to aid reuse in different contexts	Not supported
Code Generation	Allows autogeneration of code and refactoring.	Not supported
Language Independence	Supports any textual document and therefore any language	Constrained to implementation language although this will change as AOP gains wider acceptance
Use on Legacy Systems	Not supported	Supports evolution of legacy systems at source and byte code level
Dynamic Runtime Evolution	Not supported	Possible in JAC and JMangler. Future versions of AspectJ will have support.

By combining the two approaches we gain increased flexibility which will allow aspects to handle the crosscutting concerns and framing to impart configuration, optionality and generalisation of those aspects where required. Figure 4 demonstrates how a generalised aspect can be used to perform a crosscutting evolution on a system or architecture

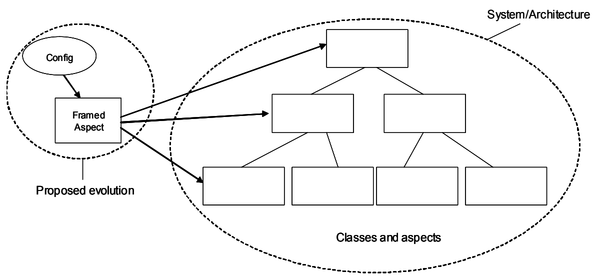


Figure 4. Evolution with framed aspects

It should be noted here that the framed aspect could work on the architecture even if the architecture itself was framed or not, thus allowing frames in some sense to work on legacy systems. Using these approaches brings forward exciting possibilities for the following:-

- Generalised reusable components which solve crosscutting problems.
- Refactorisation of aspectual code
- Configurable dynamic run time aspects
- Configurable legacy aspects
- Configurable development aspects (tracing, profiling etc)

These could be used to perform various kinds of tasks and evolutions that previously would have been difficult to realise in a particular technology alone.

5. Conclusion

We have seen that neither frame technology nor aspect oriented technologies alone can solve all the problems that evolution brings. There is clearly a need for configurable aspects for crosscutting evolution. By combining aspect orientation with a variant configuration mechanism such as frame technology we get the best of what both have to offer in terms of flexibility and evolvability. Generalisation of aspects allows them to be used in different situations thus making them ideal candidates for use within software product lines. By utilising aspect orientation and allowing crosscutting concerns to be localised we improve our understanding of system comprehensibility and thus lessen the risks of architectural erosion.

6. Acknowledgements

The authors would like to thank Dr Stan Jarzabek and Dr Zhang Weishan of the National University of Singapore with regards to queries on framing technologies.

References

[1] van Gurp J. and Bosch J., "Design Erosion: Problems & Causes", *Journal of Systems & Software*, volume 61, issue 2, 2002.

[2] Bassett, P. 1997. Framing software reuse - lessons from real world, Yourdon Press, Prentice Hall.

[3] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Symposium on Software Reusability, SSR'01,* Toronto, Canada, May 2001, pp. 164-172.

[4] Sauer, F. "Metadata driven multi-artifact code generation using Frame Oriented Programming", OOPSLA 2002.

[5] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., "Aspect Oriented Programming," *Proc. of the European Conference on Object-Oriented Programming (ECOOP),* 1997.

[6] Lientz, B., Swanson, E., and Tompkins, G., "Characteristics of Application Software Maintenance," CACM 21, No. 6 June 1978

[7] Clarke, S., Walker, R. J., "Composition Patterns: An Approach to Designing Reusable Aspects" *proceedings of the 23rd International Conference on Software Engineering* (ICSE), Toronto, Canada, May 2001.

[8] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "N Degrees of Separation:Multi-Dimensional Separation of Concerns". *Proceedings of the International Conference on Software Engineering* (ICSE'99), May, 1999.

[9] Rashid, A., Sawyer, P. et al., "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", *IEEE Joint International Requirements Engineering Conference*, IEEE Computer Society Press, 2002.

[10] Grundy, J., "Aspect-Oriented Requirements Engineering for Component-based Software Systems". *4th IEEE International Sympsium on RE*, IEEE Computer Society Press, 1999.

[11] Xerox PARC, USA, AspectJ Home Page, http://aspectj.org/

[12] IBM Research, Hyperspaces, http://www.research.ibm.com/hyperspace/

[13] Aksit, M., Bergmans, L. & Vural, S., "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", *ECOOP '92*, LNCS 615, pp 372-395, Springer-Verlag, 1992.

[14] Batory, D., Chen, G., Robertson, E. and Wang, T. "Design Wizards and Visual Programming Environments for GenVoca Generators," *IEEE Trans. on Software Engineering*, Vol. 26, No.5, May 2000, pp. 441-452

[15] Carnegie Mellon, Software Engineering Institute, homepage http://www.sei.cmu.edu

[16] Pawlak, R., Martelli, L. and Seinturier, L. The JAC project home page. http://jac.aopsys.com

[17] Kniesel, G., Costanza, P. and Austermann, M. JMangler home page, http://javalab.cs.uni-bonn.de/research/jmangler/

Managing Software Change for Variability

Christopher Thomson
Department of Computer Science
Sheffield University
Regent Court
211 Portobello Street
Sheffield S1 4DP
UNITED KINGDOM
Email: c.thomson@dcs.shef.ac.uk

***Abstract*— Software change is considered as motivation for managing software variability.**

I. Introduction

Understanding how software can be changed is essential if we want a measure of its variability and if we want to adapt it easily. Software is in part requirements, specification, code and test sets. Whenever we change any part we must ensure that it remains valid in the context of the other parts, if the system is to remain valid. Therefore we must define system validity in terms of a semantical structure. Different operators act on this semantic definition, we call these the change types.

It is hoped that there are a finite number of change types that could be built into a taxonomy of change. These could be used to describe how hard a type of change is to implement (impact) and how likely a change is to occur (risk), these together could translate to a variability factor. The taxonomy may also identify where some types of change are invalid (on areas of the system where a class of change would cause a semantic schism), and where we may be able to automatically change another part of the software to reflect a change.

Of course many of the possible change types may be unmanageable, this suggests that we may want to impose some design for change restrictions. Whilst these would not impact on the power of any software they would allow it to be changed more easily, by imposing a structure which was capable of change. Such restrictions would be designed to enhance the variability of the software at all levels of design, implementation and testing.

Supporting Variability Management at Nokia

Tanya Widen
Software Architecture Group
Nokia Research Center
tanya.widen@nokia.com

Abstract

At Nokia, software product lines are being invested in more and more to keep the business units competitive in today's market. This short position paper captures our current goal of evaluating, using, and changing or extending where necessary current best practices in variability management in order to provide integrated, full life cycle, sufficiently detailed support to our business units to enable them to successfully manage and control the variability in their domains.

1. Introduction

Nokia is a large, global company in the mobile phone and network markets, working to stay competitive in today's cutthroat business world. Nokia, as many other organizations nowadays, is moving more and more towards institutionalized software product lines to realize the benefits of systematic large scare reuse. In both mobile phones and networks, Nokia's two main lines of business, there are ample opportunities and needs to transition development to fully supported software product lines. Some domains even cross mobile phones and networks, such as DSP. Within these there is also potential benefit for investing in a joint reuse infrastructure to enhance development and maintenance efficiency.

As complete products are developed with embedded software, as opposed to only software, many different aspects of product variabilities affect the software for the products developed. These include the requirements/features and qualities variabilities, as embedded systems have many, sometimes contradicting, quality goals that can vary by products. But also include the possible variations in the HW/SW interface, the various HW platforms the software must run on, as well as system configurations. All of these aspects need to be modeled and managed in a consistent way throughout the product life cycle.

Variabilities can be introduced in any of the phases or steps along the infrastructure development path, as well as, during product instantiation. Therefore, integrated, full life cycle support is required in order to consistently variability management.

In addition to the technology aspects of variability management, technology transition issues must be addressed. To facilitate transition, detailed, practical guidelines and examples, whenever possible even in the domains of Nokia, are to be developed in order to lower the barriers to transition and acceptance. Additionally, we tailor the methods whenever necessary to our specific situation to limit the amount of information developers need to learn.

2. Variability Management

We are currently looking into many aspects of variability management as a complete, integrated solution is required that is tailored to meet the needs of our business units, yet still flexible to adapt to the varying situation specifics of each development group.

Much has been published in the field recently, as software product lines research and practice is becoming more prevalent. We are looking at these results and evaluating them for our situation. From these we will adopt what we can and integrate them with each other and our existing process, technologies, and tools. Of course, changing and extending them as necessary to reach our goal of integrated and detailed coverage.

In particular we are focusing on studying variability mechanisms in order to provide detailed practical guidance in:

- Variability mechanism selection
- Setting up support to manage the variabilities based on the mechanisms used
- Guidance on implementation and testing
- Proving support for instantiation
- Guidance on evolution aspects

For example, in the area of variability selection, we are looking at the published lists of variability mechanisms and the currently available comparison frameworks that support variability mechanism selection [2,3,4]. These tend to focus only on technology issues in selecting a mechanism, such as binding time or variability type. We would like to extend these frameworks to include other aspects we believe play an important role in variability mechanism selection. This will include business issues, such as costs of setting up and maintaining the support systems; organizational issues, such as culture or organizational structure and their impact on selecting and setting up an appropriate mechanism; as well as, skill level or experience of both developers of the reuse infrastructure and those who are intended to build systems from the infrastructure.

Related to this, we are looking at the growing field of work on variability patterns for product lines in both functional and quality aspects as these will also support developers in learning and applying variability mechanisms in their domains [1]. One key issue here is the additionally complexity of supporting tradeoff analysis to determine an appropriate product line architecture, or architectures, as instances can have varying quality requirements that would benefit from different decompositions and connections.

3. References

[1] L. Bass, M. Klein, and F. Bachmann, "Quality Attribute Design Primitives and the Attribute Design Method", *Proceedings of PFE- 4, LNCS 2290*, Springer-Verlag, Berlin, 2002, pp. 169-186.

[2] C. Gacek, and M. Anastasopoules, "Implementing Product Line Variabilities", *Proceedings of the 2001 Symposium on Software Reusability*, ACM Press, NY, NY, USA, 2001, pp. 109-117.

[3] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process, and Organisation for Business Success*, Addison-Wesley-Longman, May 1997.

[4] M. Svahnberg, J. van Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques" submitted 2002.

Leaving the Variability Management to the End-User; A Comparison Between Different Tailoring Approaches

Jeanette Eriksson, Olle Lindeberg, Yvonne Dittrich
Blekinge Institute of Technology
Department of Software Engineering and Computer Science
P.O. Box 520, S-37225 Ronneby, Sweden
Phone: +46 457 385000   Fax: +46 457 27125
jeanette.eriksson@bth.se, olle.lindeberg@bth.se, yvonne.dittrich@bth.se

Extended Abstract

A tailorable system is considered designable after the system has come in use. This means that some design decisions are postponed until the system is up and running. It is the end-user that will adjust the program to fit altered requirements. In other words tailoring entails that the variability management of the system is left to the end-user.

In the article "There's No Place Like Home: Continuing Design in Use" the authors [1] identify three ways of doing tailoring. The three possible ways are:

- o To choose between different expected behaviors.
- o To construct new behaviors out of existing components.
- o To alter the artifact.

We compare our own approach [4] with two other approaches within the area of tailoring and end-user development. Those are Anders Mørch's work with application units [5][6] and the work done within a project concerning tailorability in CSCW-systems [7][8]. All three approaches are of the latter kind of tailoring. The artifact is changed when tailoring the system.

A general problem is that when you add tailoring capabilities to a system this often makes the system more complicated: not only do you have to construct the tailoring interface but the basic program may also become more complicated. To explore how to avoid this we constructed a prototype using ideas based on the metaobject protocol (MOP) approach [2]. The metaobject protocol approach originates from the CLOS programming language in which it is possible to change program behavior by interacting with the runtime system through a metaobject protocol [3]. The metaobject protocol is based on the idea that one can and must open up programming languages so that the developer is able to adjust the language implementation to fit his or her needs. This idea has subsequently been generalized to systems other than compilers and programming language. In the article "Towards a New Model of Abstraction in the Engineering of Software" [2] it is argued that the metaobject protocol concept can be used as a general principle for abstraction in computer science. The idea is that any system that is constructed as a service to be used of client application (as for example an operation system or a database server) should have two interfaces; a base-level interface and a meta-level interface [2]. The base-level interface gives access to the functionality of the underlying system and through the meta-level interface it is possible to alter special aspects of the underlying implementation of the system so that it suits the needs of the client application. The meta-level interface is called the metaobject protocol (MOP).

We have adopted a different approach towards the metaobject protocol. The idea of the metaobject protocol approach has inspired us to transfer the concept to end-user tailorable software. In most systems the end-user has no access to the implementation of the program; in our approach the end-user is given the opportunity to alter or tailor the software should the need arise. Our aim is to give the user the opportunity to add components to the program in a controlled way which does not require any programming. To do this we use a dual-interface: a traditional base-level program and a meta-level program that provides tailoring for the base-level program. [4]

The distinction between a computational base level and a tailoring meta level is a useful one in a tailorable system. In the same way as in a metaobject protocol, the base-level implements what the system normally does. At the meta level you can change what the base level does. The two levels are also often separated in the user interface with a separate tailoring interface. The same separation may exist in the internal design. Perhaps the obvious way to do this is to let the base-level program be controlled by meta-data which stores the choices the user has made when tailoring. If the tailoring possibilities affect a large part of the program, the base-level program may become littered with tests for the value of the meta-data. If the tailoring is complicated the result may be that the base-level program looks more like an interpreter of the meta-data than a straightforward program. The alternative way to implement a tailorable system is closely linked to the metaobject protocol approach. With this approach the base-level program is a normal program which performs the normal computation only. When the system is tailored by the meta-level this is implemented by changing the base-level program. In the meta-data approach the meta-level can inspect the meta-data to see how the program is configured; it is the meta-data that will be changed during tailoring. In the alternative approach the base-level does not need any meta-data. The radical solution is to take away the meta-data from the meta-level too. This means that it is the base-level program itself that is the meta description of the current configuration. This is the method we have chosen in the prototype.

Anders Mørch at the Oslo University works with issues concerning tailoring using components called application units. An application unit is software components associated with GUI widgets but the application unit is extended with event handlers that take care of tailoring events. The structure of everyday artifacts acts as a model for application units. The units include three aspects: user interface, rational, and program code. Every aspect has three characteristics. They all have available point of use, ability to connect to other aspects by well-defined interfaces and they can be looked at as separating concerns. [6]

The application units have been used in a tool for creating and editing geometric shapes, called BasicDraw. BasicDraw has different tailoring tools embedded. The extension editor makes it possible to tailor the application by changing program code at runtime. The software components are encapsulated as a glass box. It exposes program code but the code cannot be modified. [5] The new code is built on top of the existing code because by safety reasons none of the old code in Basic Draw may be removed. The new code is compiled and linked to the existing before the application can be re-executed. The application units are to a large extent independent and can be tailored separated from other aspects but some application unit aspects are also dependent of others therefore changing one aspect may require an update of other aspects or interfaces. [6]

A research team at Bonn University takes another line of action. The group is working with tailoring in CSCW-systems. They have constructed a search tool that makes it possible for different users to tailor the presentation of search results, the handling of search results and the search space. The search tool is a part of the POLITeam-system, which provides electronic support for the work of the German government in Bonn and Berlin. The opinions of the functionality of the search tool were diverse among the users, which resulted in a tailorable system. [7]

The search tool is implemented using the JavaBeans component model. The search tool employs four different types of atomic components: search engine, result list, result switch and control button. The search engine for example is a complex component that embraces components for the search specification and for database connections. The components have a graphical representation that the user can combine in different ways. The graphical representation visualizes available ports. Gray ports visualize input and white ports output. The shape of the ports indicates the type of input or output. The graphical representations are used in a compositional technique that allows the user to customize the components and link the different ports together. Simplified, the search tool has the following functionality: The control button triggers the search engine and the search results are transported to a switch, which is parameterized to channel all documents that corresponds to certain criterion, for example found on the users own desktop, to one specific result list. Other documents that correspond to another criterion, for example found elsewhere, are displayed in another result list. [7]

Conclusively we can say that while BasicDraw supplies tailoring by adding code to the application units, the component itself, the search tool in the POLITeam-system implement tailoring in a different way, by composition of customized components. Our prototype is implementing tailoring by making use of a combination of the two implementing manners.

In the article we provide a comparison between the three approaches concerning variability, techniques and usability in the context of variability management.

References

[1] Henderson, Austin & Kyng, Morten. 1991: “There’s No Place Like Home: Continuing Design in Use”, in *Design at Work*, GreenBaum, J & Kyng, M., eds., Lawrence Erlbaum, Hillsdale, NJ.Kiczales, et.al. 1991: “The Art of the MetaObject Protocol”, *MIT Press*, England.

[2] Kiczales, Gregor 1992: “Towards a New Model of Abstraction in the Engineering of Software”, in *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tama City, Tokyo, November 1992.

[3] Kiczales, et.al. 1991: “The Art of the MetaObject Protocol”, *MIT Press*, England.

[4] Lindeberg, Olle & Eriksson Jeanette & Dittrich, Yvonne 2002: “Using Metaobject Protocol to Implement Tailoring;

Possibilities and Problems*”*, in *The 6th World Conference on Integrated Design & Process Technology (IDPT ‘*02), Pasadena, USA, 2002.

[5] MØrch, Anders I. 2003: ” Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units”, in *N. Patel: Adaptive Evolutionary Information Systems*. Idea group Inc. 2003.

[6] MØrch, Anders I. & Mehandjiev, Nikolay D. 2000: ”Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units”, in *Computer Supported Work 9:*75-100, Kluwer Academic Publishers.

[7] Stiemerling, Oliver & Cremers, Armin B. 1998: ”Tailorable component architectures for CSCW-systems” in *Parallel and Distributed Processing, 1998. PDP '98.* Proceedings of the Sixth Euromicro Workshop pp: 302-308, IEEE Comput. Soc.

[8] Stiemerling, et.al, 1999: ”The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware”, in *EDOC’99 (Enterprise Distributed Object Computing*), Mannheim, Germany, Sept.27-30, IEEE Press.

A Knowledge-based Product Derivation Process and some Ideas how to Integrate Product Development

(Position paper)

Lothar Hotz and Andreas Günter
HITeC c/o Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: hotz@informatik.uni-hamburg.de

Thorsten Krebs
Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: krebs@informatik.uni-hamburg.de

Abstract— **In this position paper, a product derivation process is described, which is based on specifications of known customer requirements, features, artifacts in a knowledge base. In such a knowledge base a model about all kinds of variability of a combined software/hardware systems are represented by using a logical-based representation language. Having such a language, a machinery which interprets the model is defined and actively supports the product derivation process e.g. by handling dependencies between features, customer requirements, and artifacts. Because the adaptation and new development of artifacts is a basic task during the derivation process where a product for a specific customer is developed, the evolution task is integrated in the proposed knowledge-based derivation process.**

I. Introduction

The product line approach makes the distinction between a domain engineering part, where a common platform for an arbitrary number of products is designed and realized, and an application specific engineering part, where a customer product is derived (*product derivation process*) [1], [3]. In this position paper, a product derivation process which includes both the selection and assembling of artifacts out of a platform and their adaptation, modification, and new development for customer specific requirements is presented.[1]

The main underlying assumption is based on the existence of a descriptive model for representing already developed artifacts and their relations to features and customer requirements as well as the underlying architectural structure with its variations. All kinds of variability are represented (described) in such a model. Thus, variability is made explicit while the realization of the variability in the source code is still separate. This model is called *configuration model*. It is specified in a *knowledge base*. Thus, we speak of a *knowledge-based* product derivation process (*kb-pd-process*). Furthermore, it is assumed, that such a model is necessary to manage the increasing amount of variability in software-based products. Such a configuration model can be used for automatically configuring technical systems, where ”configuring” means selecting, parameterizing, constraining, decomposing, specializing, and integrating components of diverse types (e.g. features, hardware, software, documents, etc.).

A configuration model describes all kinds of variability in a software system. Thus, it describes all potentially derivable products. But this is done on a descriptive level: when using a configuration model with an inference engine, only a description of a product is derived, not the product itself. But it is intended to use the description for collecting the necessary source code modules and realizing (implementing, loading, compiling etc.) the product in a straight forward manner. Furthermore, a configuration model is *not* a model to be used for *implementing* a software module, e.g. it does not describe classes for an object-oriented implementation.

In the following, we first describe some distinct levels of abstraction which we have to deal with when describing system entities (Section II). In Section III, we present the language entities as well as their interplay in the product derivation process. Evolution aspects are included in Section IV. A short discussion of some related work is given in Section V.

[1] We only consider engineering aspects of the process, we exclude economical aspects. As roles we simply see a team of software developers, which have to do both: developing a commonly used platform for all products and customer specific products.

II. Levels of abstraction

We can identify three kinds of work to be done on distinct levels of abstraction for exploring a knowledge-based product derivation process:

1) **Language for specifying the knowledge base – What is used for modeling?**
   This level describes what can be used for modeling the general aspects of the process and the domain specific part. This is done by specifying a language, that can be used to describe the necessary knowledge. Furthermore, a machinery (inference engine) for interpreting this description is specified and realized in a tool. Basic ingredients of the language are concepts, relations between concepts, procedural knowledge and a specific task description (see [7], [9] for an example of such a language and a suitable tool). Entities of this language are further described in Section III.
2) **Aspects of the process – What are the general ingredients of a product derivation process?**

On this level, general aspects that have to be modeled for engineering and developing products are specified. This level determines, which entities for the kb-pd-process have to be described. This is intended to be a description for a number of kb-pd-processes in distinct business units or companies, ideally for development of combined hardware/software systems in general. The description of a *specific* domain is done on the next level. Specification is done on a textual basis as well as on a model basis by using the language.
Following aspects of the kb-pd-process are currently taken into account:

- **Customer requirements:** A description of known and anticipated requirements expressed in terms which can be understood by the customer.
- **Features:** A description of the facilities of the system and its artifacts.
- **Artifacts:** A description of the hardware, software components and textual documentations to be used in products.
- **Phases of the process:** A description of general phases of the process, e.g. "determine customer requirements", "select appropriate features", "select and adapt necessary artifacts".
- **Reference configurations:** A description of typical combinations of artifacts (cases), which can be enhanced or modified for a specific product.

For each aspect an *upper model* with e.g. decompositions (e.g. sub-features) and relations of aspects is expressed. The upper model describes common parts of domain specific models. Upper models are used to facilitate the domain specific modeling. An example of an upper model is given in Figure 1. Two different views on features (i.e. customer-view (`cv-feature`) and technical-view (`tv-feature`)) are shown. Both specialize to a concept which has sub-features and one which doesn't (`cv-no-subs`, `cv-with-subs`). The dotted arrows indicate places where the domain specific models come in. Lines indicate specialization relations and arrows decomposition relations. This example shows how conceptual work done in [5], [10], [11], [16] can be used for specifying an upper model, which in turn can be used for automatic product derivation.
Each aspect of the process is modeled by using the language. Thus, it is described how e.g. customer requirements and their relations can be represented by using concepts and concept relations. In this paper, we do not further elaborate on this topic.

3) **Domain specific level – What is modeled for a specific domain?**
On this level a domain specific model is specified by using the language and the upper model. By interpreting the model with a machinery (given by a tool), this model is used for performing the process. For developing software modules (i.e. on a file, source code, developer model level) development tools and software management tools are integrated. In this paper, we do not further elaborate on this topic.

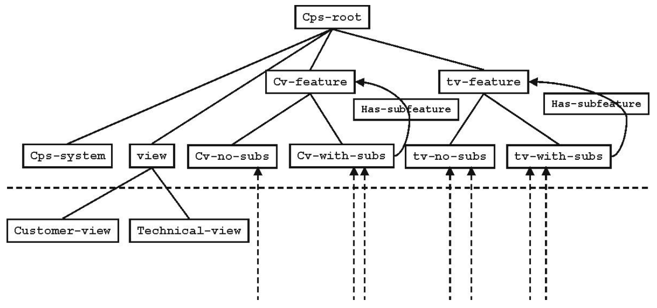


Fig. 1. Example of an upper model

III. Entities of the knowledge based model

Basic entities of the model and the process are listed as follows:

1) A **concept model** for describing concepts by using names, parameters and relations between parameters and concepts. Main relations are decomposition relations, specialization relations, and n-ary relations between parameters of arbitrary concepts expressed by constraints. Such concept models can be used to describe properties and entities of products like features, customer requirements, hardware components, and software modules.
2) **Procedural knowledge** mainly consists of a description of strategies. A strategy focuses on a specific part of the concept model. E.g. a strategy focuses on features, another one on customer requirements and a next one on software components, or on the system as a whole. Furthermore conflict resolution knowledge which is used for resolving a conflict (e.g. by introducing explicit backtracking points) is described.
3) A **task specification** which describes a priori known facts, a specific product has to fulfill.

Strategies are performed in *phases*. In each phase one strategy is used, which focuses on a specific part of the model. After selecting this part, in a phase all necessary decisions (i.e. *configuration steps*) are determined by looking at the model. Each configuration step represents *one* decision, e.g. the setting of a parameter value, or processing a decomposition relation. Possible *configuration steps* are collected in an *agenda*, which can be sorted in a specific order, e.g. first decomposing the architecture in parts, then selecting appropriate components, and then parameterizing them. Decisions can be made by using distinct kinds of methods including automatic or manual ones. Each decision is computed by a *value determination method*, which yields to a specific value representing the decision. Examples for value determination methods are: "ask the user", "take a value of the concept model" or "invoke a given function". Thus, in a configuration step the decisions to be made are described and after applying some kind of value determination method the resulting value is stored in the *current partial configuration*. A partial configuration represents all decisions made so far and their implications, which are drawn by the mechanisms described in the following.

In a cyclic practice, after each configuration step more global (i.e. systemwide) mechanisms are (optionally) executed. Examples are:

- **Constraint propagation:** For computing inferences followed by a decision and for validating the made decisions, constraints defined in the knowledge model (i.e. constraints represent relations between parameters of concepts) are propagated, based on some kind of constraint propagation mechanism.
- **External mechanisms:** For performing an external method, which does not use the concept model but only the currently configured partial configuration external techniques can be applied. Examples are:
  - simulation techniques: a simulation model is derived from the partial configuration and a separated module (like matlab) is called for this task. Some specific kind of simulation in the area of software product derivation is "compiling the source files".
  - optimization techniques: the current partial configuration is used to compute optimal values for some parameters of the configuration.
- **Further logical inferences:** Methods, which perform logical inferences that are not performed using the decision process but use the concept model, can be invoked (e.g. taxonomic inferencing, description logic etc.).

The objective of global mechanisms is to compute values for not yet fixed decisions or to validate the already made decisions. Those mechanisms (if more than one is present) are processed in an arbitrary order but repeated until no new values are computed by those mechanisms, i.e. until a fixed point is reached. If this validation is not successful or the computed value for a parameter is the empty set, a *conflict* is detected. An example would be, if the compilation of the source files fails. A conflict means that the task description, the subsequent decisions made by the user, and their logical impacts are not consistent with the model. For resolving a conflict, diverse kinds of *conflict resolution methods* (e.g. backtracking) can be applied to make other user-based decisions (see [9]). On the other side, one could also try to change the model, because if a conflict is detected, with the given model it is not possible to fulfill the given task descriptions and user needs. This gives raise to evolution, i.e. to modify or newly develope artifacts and include them in the model, so that the needs can be fulfilled (see Section IV).

Summarizing the kb-pd-process performs the following (slightly simplified) cycle:

Until no more strategy is found:

1) Select a strategy
2) Compute the agenda according to the focus
3) Until the agenda is empty or a termination criteria of the strategy is satisfied:
   - Select an agenda entry
   - Perform a value determination method
   - (Optionally) execute the global mechanisms
   - If a conflict occurs, evaluate conflict resolution knowledge.

IV. Including evolution aspects in the process

Above a well-known configuration process is described (see [4], [6]). The changing of artifacts and further development of new components (i.e. *evolution*) can be included in this process as described in the following subsections. The aspect of evolution can be seen as a kind of *innovative configuration*. We see innovative configuration not as an absolute term but as a relative one – relative to a model. A model describes a set of configurations which can be configured by using it. Innovation related to this model is given, if the configuration process computes a configuration which does not belong to this set. For supplying a product derivation process where evolution of artifacts are a basic task, we expect to apply methods known in innovative configuration to be used. A survey on innovative configuration is given in [9], [12].

A. Points of evolution

Following situations which come up in the process described in Section III indicate the necessity for evolution:

1) Pro-active, foreseen evolution, more general models: Instead of narrowing the model, broader value ranges for parameters and relations can be modeled a priori. For example, the sub-models describing customer requirements or features can represent more facilities than the underlying artifacts can realize. If during the derivation process such a feature is selected by the task description or inferred by the machinery, it gives raise to evolution of an artifact.
2) Conflicts which cannot be resolved by backtracking, i.e. by using the current model, indicate places where evolution can take place. For example, if two artifacts are chosen which are incompatible, a resolution of such a conflict would be to develop a new compatible artifact and include it into the model.
3) Points set by the user: Instead of selecting a value at a given point, the evolution of the model can be started by the developer for integrating a new or modified artifact in the partial configuration. Another example is given when the user does not accept the automatically made decisions. Thus, an evolution process is explicitly started by the user to change the model for making another decision than the model indicates. Thus, evolution as a kind of value determination method is introduced.
4) A further point is given when evolution is seen as a further global mechanism. Thus, it is included after a decision is made. Some conditions are tested on the partial configuration when evolution should be started. One trivial condition is given when the user does not accept the automatically made inferences. Thus, transparency must be given to make such a decision. If the evolution changes existing descriptions, the partial configuration must be adapted and the other global mechanisms must be invoked to find a new fixed point.

B. Evolve the model

All dependencies of the new concept (features, artifacts, customer requirements) to existing ones must be specified. Having a model, the context where a new concept will be included, can be computed on the base of the model. For instance, the related constraints of an depending aggregate or a part-of decomposition hierarchy can be presented to the developer for considering during the evolution of the model.

C. Supporting the evolution of features, customer requirements and artifacts by a knowledge-base approach

By analyzing the knowledge base, following information used for development, can be presented to the developer. The underlying idea is to present those parts of the model, which can be used in special development situations, to the developer.

- Present the already defined concepts with their parameters and relations.
- Present the specialization relation of all, of some selected or of some depending concepts. In the last case subgraphs, which describe a specialization context of a given concept are computed, e.g. the path to the root concept with direct successors of each node.
- Present the decomposition relation of a given relation of all, of some selected or of some depending concepts. In the last case subgraphs which describe the decomposition context of a given concept are computed, e.g. all aggregates, which the concept are part-of and all transitive parts which the concept has.
- Given a concept, present all concepts which are in relation to it by analyzing the constraints, i.e. also a subgraph is computed. Because constraints relate parameters of concepts the subgraph presents not only concepts but also relations between parameters.
- Given a concept, present all strategies where a parameter or relation of the concept is configured.
- Given a new concept description (with parameters and relations), compute a place in the specialization hierarchy for putting the concept into.

Knowledge modeling can be seen as a specific kind of evolution. If no given model exists, knowledge modeling is an evolution of the always given upper model. The mentioned services can be used for bringing up the first model of the existing artifacts, features and customer requirements. Thus, by supporting the evolution task, the task of knowledge modeling is also be supported.

D. Conflict resolution with an evolved model

When the model is changed, e.g. because new artifacts are included, the changes must be consistent with the ordinary model and the already infered impacts stored in the partial configuration. What kind of resolution techniques are useful have still to be developed. One trivial approach is to start the total process again with the new model and the old taks, and make all decisions of the user automatically. Thus, test the new model with the user needs, if they are consistent. This can be done automatically, because all user inputs are stored as such in the partial configuration, only the impacts have to be computed again, based on the new model. Another approach is to start some kind of reconfigration or repair technique, which changes the partial configuration according to the new model.

E. Evolve the real components

Last but not least the new components have to be build. The new source code can be implemented by using existing tools for developing and changing software systems.

F. The kb-pd-process with the evolution task included

Summarizing the kb-pd-process where evolution is included looks like:

```
Until no more strategy is found:
  1) Select a strategy
  2) Compute the agenda according to the focus
  3) Until the agenda is empty or a termination criteria of the strategy is
     satisfied:
       • Select an agenda entry
       • Perform a value determination method or start evolution
       • (Optionally) execute the global mechanisms, included the evolu-
         tion task
       • If a conflict occurs, evaluate conflict resolution knowledge.
```

V. Related Work

There are some approaches which try to automate software processes [14], [15]. The main distinction to the approach proposed in this paper is the different kind of knowledge representation. Instead of using rule-based systems, which have deficiencies when used for big systems [6], [8], [17], a basic concern of the language we propose is to separate distinct types of knowledge (like conceptual knowledge for describing components and their variability and procedural knowledge for describing the process of derivation). A requirement which is e.g. not followed in [2], where information about components is mixed with information about binding times in UML diagrams. One has to distinguish the knowledge representation and the presentation of the knowledge to the user. For presenting it might be useful to mix some knowledge types at certain situations (as described in IV-C). But for maintainability and adequacy reasons it is of specific importance to separate them.

In [13] a support for human developers, which is not based on automated software processes, is proposed. E.g. representations are mainly designed for human readability instead of machine interpretation. As a promising approach, structured plain text based on XML notations are considered. Thus, the combination of formal structured knowledge and unstructured knowledge should be achieved. On the one hand XML is only a mark-up language, where the main problem is to create a document type definition, which describes the documents to be used for representing software. One could see the language described in Section III as a specification for such a DTD. Thus, in our opinion for formally describing configuration knowledge in a structured way the necessary type definitions are already known. On the other hand, if unstructured knowledge should be incorporated one should also define tools which can handle them in a more than syntactic way (e.g. similarity-based methods or data-mining techniques), to get a real benefit of those kinds of representations.

VI. Conclusion

Making knowledge about features, customer requirements, and artifacts explicit in a model and a tool-based usage of such a model yields to an automatic product derivation process.[2] It was shown, how such a product derivation process can be defined. Furthermore, the evolution of artifacts is introduced in the process and can be supported by using the knowledge which is explicit in the model.

References

[1] J. Bosch, *Design & Use of Software Architectures: adopting and evolving a product line approach*, Addison-Wesley, 2000.

[2] M. Clauss, 'Generic modeling using uml extensions for variability', in *DSVL 2001*. Jyvaskylae University Printing House, Jyvaskylae, Finland, (2001).

[3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[4] R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode, 'Plakon - an approach to domain-independent construction', in *Proc. of Second Int. Conf. on Industrial and Engineering Applications of AI and Expert Systems IEA/AIE-89*, (1989).

[5] A. Ferber, J. Haag, and J. Savolainen, 'Feature interaction and dependencies: Modeling features for re-engineering a legascy product line', in *Proc. of 2nd Software Product Line Conference (SPLC-2)*, Lecture Notes in Computer Science, pp. 235–256, San Diego, CA, USA, (August 19-23 2002). Springer Verlag.

[6] A. Günter and R. Cunis, 'Flexible control in expert systems for construction tasks', *Journal Applied Intelligence*, **2(4)**, 369–385, (1992).

[7] A. Günter and L. Hotz, 'Konwerk - a domain independent configuration tool', *Configuration Papers from the AAAI Workshop*, (1999).

[8] A. Günter and C. Kühn, 'Knowledge-based configuration - survey and future directions', in *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, ed., F. Puppe, Springer Lecture Notes in Artificial Intelligence 1570, (1999).

[9] A. Günter (Hrsg.), *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995. in german.

[10] A. Hein, J. MacGregor, and S. Thiel, 'Configuring software product line features', in *Proc. of ECOOP 2001 - Workshop on Feature Interaction in Composed systems*, Budapest, Hungary, (June, 18 2001).

[11] A. Hein, M. Schlick, and R. Vinga-Martins, 'Applying feature models in industrial settings', in *Proc. of First Software Product Line Conference - Workshop on Generative Techniques in Product Lines*, Denver, USA, (August, 29th 2000).

[12] L. Hotz and T. Vietze, 'Innovatives Konfigurieren in technischen Domänen', in *S. Biundo (Hrsg.), 9. Workshop Planen und Konfigurieren*, Kaiserslautern, Germany, (1995). DFKI Saarbrücken. in german.

[13] R. Kneuper, 'Supporting software processes using knowledge management', in *Handbook of Software Engineering and Knowledge Engineering*, volume 2, Singapore, (2002). World Scientific.

[14] L. Osterweil, 'Software processes are software too', in *Proceedings of the 9th International Conference on Software Engineering (ICSE9)*, (1987).

[15] H. D. Rombach and M. Verlage, 'Directions in software process research', in *Advances in Computers*, volume 41, (1995).

[16] M. Schlick and A. Hein, 'Knowledge engineering in software product lines', in *Proc. of ECAI 2000 - Workshop on Knowledge-Based Systems for Model-Based Engineering*, Berlin, Germany, (August, 22nd 2000).

[17] E. Soloway and al., 'Assessing the maintainabiliy of xcon-in-rime: Coping with the problem of very large rule-bases', in *Proc. of AAAI-87*, pp. 824–829, (1987).

[2]"Automatic" does of cause not mean totally automatic, task descriptions and user interactions are still possible, but logical impacts can be drawn by the inference engine.