




Scaling New Heights



**International Workshop on
Software Variability
Management (SVM)**

*ICSE'03
International Conference on Software Engineering
Portland, Oregon
May 3-11, 2003*

Table Of Contents

Software Configuration Management Problems and Solutions to Software Variability Management Lars Bendix, Lund Institute of Technology	5
Using UML Notation Extensions to Model Variability in Product-line Architectures Liliana Dobrica, University Politehnica of Bucharest; Eila Niemelä, VTT Electronics	8
Consolidating Variability Models Birgit Geppert, Frank Roessler, David M. Weiss, Avaya Labs Research	14
Managing Variability with Configuration Techniques Andreas Hein, John MacGregor, Robert Bosch Corporation	19
Supporting the Product Derivation Process with a Knowledge-based Approach Lothar Hotz, Thorsten Krebs, Universität Hamburg	24
Variability Analysis for Communications Software Chung- Horng Lung, Carleton University	30
Evolving Quality Attribute Variability Amar Ramdane-Cherif, Samir Benarif, Nicole Levy, PRISM, Université de Versailles St.- Quentin; F. Losavio, Universidad Central de Venezuela	33
A Practical Approach To Full-Life Cycle Variability Management Klaus Schmid, Isabel John, Fraunhofer Institute for Experimental Software Engineering (IESE)	41
Capturing Timeline Variability with Transparent Configuration Environments Eelco Dolstra, Utrecht University; Gert Florijn, SERC; Merijn de Jonge, CWI; Eelco Visser, Utrecht University	47
Component Interactions as Variability Management Mechanisms in Product Line Architectures M. S. Rajasree, D. JanakiRam, Indian Institute of Technology, Madras	53
Feature Modeling Notations for System Families Silva Robak, The University of Zielona Gora	58
Variability in Multiple-View Models of Software Product Lines Hassan Gomaa, George Mason University; Michael Eonsuk Shin, Texas Tech University . .	63
Managing Knowledge about Variability Mechanism in Software Product Families Kannan Mohan, Balasubramaniam Ramesh, Georgia State University	69
Towards a component-based, model-driven process supporting variability of real-time software Vieri Del Bianco, Luigi Lavazza, CEFRIEL - Politecnico di Milano	74

Organizers

Jan Bosch, University of Groningen

Peter Knauber, Mannheim University of Applied Sciences

Program Committee

Gert Florijn, SERC

Danny Greefhorst, IBM Global Services

Henk Obbink, Philips Research

Klaus Pohl, University of Essen

Paul Sorenson, University of Alberta

Kai Koskimies, Tampere University of Technology

Software Configuration Management Problems and Solutions to Software Variability Management

Lars Bendix

Department of Computer Science

Lund Institute of Technology

Sweden

bendix@cs.lth.se

Abstract

These days more and more software is produced as product families. Products that have a lot in common, but all the same vary slightly in one or more aspects. Developing and maintaining these product families is a complex task. Software configuration management (SCM) can, in general, support the development and evolution of one single software product and to some degree also supports the concept of variants. It would be interesting to explore to what degree SCM already has solutions to some of the problems of product families and what are the problems where SCM has to invent new techniques to support software variability management.

1. Introduction

Software product families are becoming more and more common. There are many benefits from making and selling basically the same product in many slightly different variations. You can amortise your investment on a greater number of sold products. You can satisfy your customers' desire for specialised products at a reasonable cost. Your product will be able to run on many platforms and support many languages in a globalised market. You will be able to reuse parts from existing products allowing a faster time-to-market time.

However, there is also a down side to software product families. They are very complex to develop and even more complex to maintain and further evolve. And because of the high development costs and popularity of such products they tend have long lives. In order to balance the costs of the production with the benefits from increased sales of software product families, we need to be able to manage the complexity of software variability.

In my opinion, Software Configuration Management (SCM) has a major role to play when it comes to handling

the complexities of software variability. SCM already has a fundamental role in supporting the development and evolution of single product software. If we were able to distinguish exactly how product families differ from single product software, we might discover that SCM can provide – or develop – techniques to support product families too.

In the following, I will first briefly outline the most important concepts and principles of SCM relevant to product families, after which I describe how I look at software product families. Then I will detail where I see previous relations between SCM and SVM – and finally state my position on what could be some of the discussion points at the workshop regarding the relationship between SCM and Software Variability Management (SVM).

2. Software Configuration Management

For the past two decades my main interest has been SCM. It is a discipline that spans a wide spectrum of functionality, ranging from computer supported co-operative work, that supports the developers in their tasks, to product data management, that enables the company to document the exact composition of their products.

SCM is the control of the evolution of complex software systems. Traditionally it is said to cover the life-cycle phases from coding to retirement. However, you can apply SCM principles and techniques to everything from your requirements specifications through design documents to documentation.

One of the core concepts of SCM is the management of a repository of components. Everything that the developers produce goes into this repository where it remains forever, such that it can be retrieved again at any time. When changes are made to something in the repository, they are not made directly to the component, but rather to a copy and the modified copy is then added

to the repository creating a new version of that component. This means that we can have many versions of the same component.

Another core concept is that of a product model. This product model describes the structure of the product by relating the components in a dependency graph. Furthermore, the product model also contains information about which actions to perform to build the product from its components. This means that once this product model has been described it is possible to automatically build the product.

When we put these two concepts together we get a slightly more complicated picture. The fact that the repository might contain more than one version of a given component means that there is a choice between which version to use for each component when we build the product. This gives rise to the concept of a generic product model that gives only the structure. This generic model is then bound to particular versions of components using the repository and a selection profile describing which version to use for which components.

These are problems where SCM has solutions that are well understood and mature. A naive and simplistic view at SVM would be to look at variants the same way as versions and use these already established solutions. However, variants do differ from versions so such an approach has strong limitations.

3. Software Product Families

Unfortunately I am not yet an expert on the subject of product families. However, in this section I will describe what I perceive as some of the important aspects of software product families.

From what I know, variation points have emerged as a way to describe the parts where a component can or is allowed to vary. There may be tens of thousands of variation points in a product line – and that might sound frightening. However, in my opinion it is not the number of variation points that should cause most concern with regard to complexity. It is the number of dimensions in which a product family is allowed to vary. Usually that number is in the tens or less, but still this low number generates far more complexity.

The complexity of managing the amount of variability, both in number of places and number of dimensions, can become extremely high. Especially as the product family evolves in time introducing also versions.

4. SCM relations to SVM

Within the SCM community there was an early interest in variants and the handling of variability. Variations that could be confined within a single file and involved only

minor pieces of code was studied in [WS88], and a mechanism very similar to conditional compilation was proposed. A later study by [Reichenberger89] distinguished versions and variants as two separate dimensions in which a component could differ. In his approach he put emphasis on a model for the whole product.

So there are already solutions for managing software variability provided by SCM. But are they sufficient for the variability found in today's product families?

5. Position statement

From my point of view it would be interesting to look at SCM from a solutions perspective. Can we “discipline” or stream-line the use of variability in product families such that it can be managed by present SCM techniques – and how?

Equally as interesting would it be to look at software variability management from a problems perspective. To listen to an analysis of what the problems are to discover the limitations of SCM support. This could lead to the development of new SCM techniques to support (part of) these problems.

It is “common wisdom” that evolution and variability should be “kept apart” – but how do you implement that?

More specifically such a discussion could investigate the following five themes:

Top-down vs. bottom-up approach: Is the variation points a top-down approach, whereas current SCM techniques are bottom-up? Are these for large and small variations and which can be supported by SCM and how?

Anticipated vs. unanticipated variability: The latter cannot be designed into the architecture. They could also be considered as sort of proactive and reactive approaches. Can we make a mix – and how should that be done? Would it be possible – and desirable – to use techniques like refactoring to bring unanticipated variability under control?

Life-cycle phases: Is there any difference in the support you need depending on when in the life-cycle you bring in variability? Early (analysis/design), middle (implementation), late (release – or production – and maintenance). Which phases are variation points, SCM- and PDM techniques most suited for?

Evolution vs. variability: These two dimensions should be kept apart. But why – and how?

Traceability: How do we trace and relate a variation point from the analysis to the design to the code to the tests to the documentation?

6. References

[Reichenberger89]: Reichenberger, Christoph:
Orthogonal Version Management, in
proceedings of the 2nd International Workshop on
Software Configuration Management, Princeton,
New Jersey, October 24-27, 1989.

[WS88]: Winkler, Jürgen F. H. & Stoffel, Clemens:
Program-Variations-in-the-Small, in
proceedings of the 1st International Workshop on
Software Version and Configuration Control,
Grassau, Germany, January 27-29, 1988.

Using UML Notation Extensions to Model Variability in Product-line Architectures

Liliana Dobrica
University Politehnica of Bucharest
Spl. Independentei, 313, Sect. 6, 77206
Bucharest, Romania,
ldobrica@dig.ro

Eila Niemelä
VTT Electronics
P. O. Box 1100 FIN-90571
Oulu, Finland
Eila.Niemela@vtt.fi

Abstract

The purpose of this paper is to define the extensions of the UML standard specification for the explicit representation of variations and their locations in software product line architectures based on a design method already established. The method will benefit a more familiar and widely used notation, facilitating a broader understanding of the architecture and enabling more extensive tool support for manipulating it.

The description of the modeling constructs that manage variability represents a part of a profile of the extended or applied UML concepts intended primarily for use in modeling product-line architectures. These new constructs have to be used in combination with all the other UML modeling concepts and diagrams to provide a comprehensive modeling tool set.

1. Introduction

Product line software development requires a systematic approach from such multiple perspectives as business, organizational, architecture and process. In the product line context, the architecture is used to build a variety of different products. For several years the focus of our research has been product line architecture (PLA) design and analysis. One of our goals was to define a quality-driven architecture design and analysis (QADA) method [1]. An important issue in our research was to explicitly represent variation and indicate locations for which change is allowed. In this way, the diagrammatic description of the PLA defined by using the QADA method helps in instantiating PLA for a particular product or in its evolution for future use. From the PLA documented diagrammatically, it is easy to detect what kind of modifications, omissions and extensions are permitted, expected or required.

The QADA method was described by defining and using a framework that consisted of the following ingredients:

- an underlying model, referring to the kinds of constructs represented, manipulated and analyzed by the model;
- a language, which is a concrete means of describing the constructs, considering possible diagrammatic notations;
- defined steps, and the ordering of these steps;
- guidance for applying the method; and
- tools that help in carrying out the method.

In order to achieve an optimal method for a certain development effort, these ingredients can be defined or selected more properly. Some of these ingredients may already be available (e.g. from the literature, from tool vendors, etc.), whereas others may have to be specially developed, configured or extended.

The work in this paper puts in practice this idea of method improvement with the purpose of defining the UML extensions for the management of variability in space in the software architectures of product lines. The extensions are described through the viewpoints defined by the QADA method. The method will benefit a more familiar and widely used notation, therefore facilitating a broader understanding of the architecture and enabling more extensive tool support for manipulating it.

One of our goals is to describe modeling constructs that manage variability and represent a part of a profile of the extended or applied UML concepts intended primarily for use in modeling the product line architectures. These new constructs have to be used in combination with the other UML modeling concepts and diagrams to provide a comprehensive modeling tool set.

The beginning of this paper is a brief description of the viewpoints of the QADA method, with the focus on modeling elements and relationships with UML extension mechanisms and notation. The next section examines some of the structural and behavior constructs that model variability, trying to interpret them based on UML concepts. UML extension mechanisms are used if a refinement of the UML metamodel is necessary. The final

result of our research is the definition of a UML profile for designing software architectures based on the QADA method. We think that standardization of the UML profile defined in our study will be of benefit to the software architecture developer community, especially for software product lines where a systematic approach is mostly required.

2. Modeling constructs and notation

The modeling constructs used by the QADA method for representing software architectures for product line development are partitioned into four groups: constructs for modeling a structural view, constructs for modeling a behavioral view, constructs for modeling a deployment view and constructs for modeling a development view. Variation in space is an integral part of the first three views, contrary to the development view that represents the categorization and management of domains, technologies and work allocation. There are also two levels of abstraction to be considered in PLA descriptions: the conceptual level and the concrete level.

Entities of each view are defined in detail and described in [1] and [2]. Figure 1 presents the entities of three major conceptual views that embody variation in space. Variation in time is managed through the conceptual and concrete development views, but that is outside the scope of this paper.

On a concrete level there are other architectural elements (i.e. capsules, ports, state diagrams, deployment diagrams, etc.) and relationships between them in each of the views.

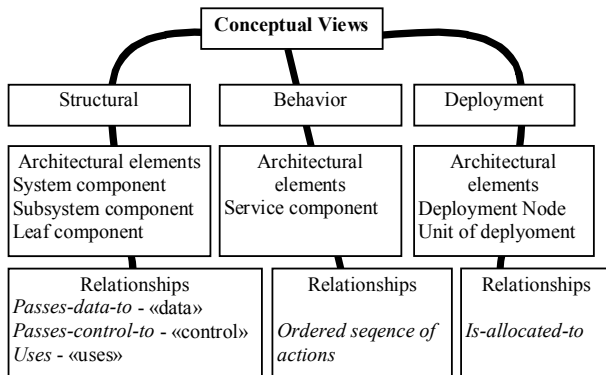


Figure 1. Entities of the Conceptual Views.

To address UML extensions in accordance with the QADA method we decided to define and apply a framework, accompanied by a set of activities and techniques, for identifying differences between the UML standard [3] and the QADA viewpoint description.

The framework is based on the following activities:

- *Mapping*: Identifies what information is overlapping between the existing QADA language and UML;

- *Differentiation*: Identify differences between the UML standard model elements and those defined by QADA;
- *Transformation*: By using UML extension mechanisms or other techniques we try to integrate the UML standard with the new required QADA elements.

UML supports the refinement of its specifications through three built-in extension mechanisms [3]:

- *Constraints* that place semantic restrictions on particular design elements; UML uses the Object Constraint Language (OCL) to define constraints.
- *Tagged values* that allow new attributes to be added to particular elements of the model.
- *Stereotypes* that allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements.

Tabular forms for specifying the new extension elements have been organized (Figure 2). For stereotypes, the tables identify stereotype name, the base class of the stereotype that matches a class or subclass in the UML metamodel, the direct parent of the stereotype being defined (NA if none exists), an informal description with possible explanatory comments and constraints associated with the stereotype. Finally, the notation of the stereotype is specified.

<p>Tabular form of a Stereotype definition</p> <ul style="list-style-type: none"> • Stereotype: Leaf • Base Class: Subsystem • Parent: Architectural element • Description: ... • Constraints: None or self.isMandatory=true • Tags: None • Notation: A UML package stereotyped as «leaf» 	<p>Tabular form of a Constraint definition</p> <ul style="list-style-type: none"> • Constraint: isMandatory • Stereotype: Leaf • Type: UML::Datatypes:Boolean • Description: Indicates that the Leaf is Mandatory
	<p>Tabular form of a Tag definition</p> <ul style="list-style-type: none"> • Tag: isDynamic • Stereotype: Capsule • Type: UML::Datatypes:Boolean • Description: Identifies if the associated capsule class may be created and destroyed dynamically.

Figure 2. Examples of stereotypes, constraints and tag definitions.

For example, based on QADA, the conceptual structural view is used to record conceptual structural components, conceptual structural relationships between components and the responsibilities these elements have in the system. Specifically in QADA, the constructs for modeling this view are summarized in Figure 1.

Typically, UML provides class diagrams for capturing the logical structure of systems. Class diagrams capture universal relationships among classes – those relationships that exist in all contexts.

Components of this conceptual structural view are mapped onto the *Subsystem* UML concept. We identified

a hierarchical description of components that introduces differences between them and requires transformations using new stereotypes. The stereotypes add additional conceptual-specific semantics onto the various aspects that are associated with the UML-based classes. We proceeded with mapping elements and identifying the new required stereotypes.

A graphical equivalent of the stereotype declarations previously described for tabular form is presented in Figure 3. The class diagram has been realized in Rose RT [4]. This shows the relationships among UML metaclasses and the new stereotypes they represent in architectural components. Generalization and predefined «stereotype» dependency are included in this diagram.

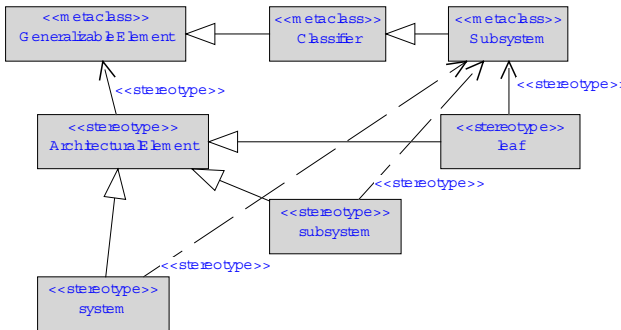


Figure 3. Graphical equivalent of the stereotype declarations described in tabular form.

We also indicate whether or not a commercial tool case supports the UML standard and extended elements.

3. Modeling variability using UML extensions

An important aspect of product line architectures is variation among products. A variability mechanism is a wide range of generalization and specialization techniques. Jacobson [7] defined the following variability mechanisms: inheritance, uses, extensions, parametrization, configuration and generation. Nevertheless, variation is difficult to model in architectural descriptions. UML models provide the means to use specific variation mechanisms [6] [7] to describe hierarchical systems (ways to decompose systems into smaller subsystems). However, the UML standard does not support a description of variation, as QADA requires.

3.1 Conceptual structural view

We consider variation in the conceptual structural view to be divided into internal variation (within Leaf components) and structural variation (between Leaf/Subsystem components). To enable variation, we separate components and configurations from each other. Flexible

representations are needed to instantiate components and bind them into configurations during product derivation.

3.1.1. Structural variation. The structural conceptual view has to offer the possibility of preventing automatic selection of all Leaf or Subsystem components included in a System during product derivation. Variability can be included in this view by using specific stereotypes for the architectural elements (Figure 4).

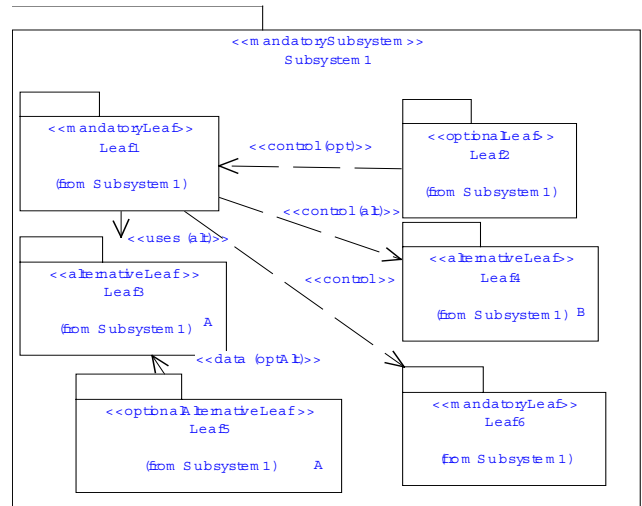


Figure 4. Variation points included in the conceptual structural view.

Thus we consider that a Leaf or a Subsystem could be further stereotyped in:

- «mandatoryLeaf» or «mandatorySubsystem»
- «alternativeLeaf» or «alternativeSubsystem»
- «optionalAlternativeLeaf» or «optionalAlternativeSubsystem»
- «optionalLeaf» or «optionalSubsystem».

In case of «alternative» or «optionalAlternative» variability of a Leaf or Subsystem, the inclusion of a letter “A” or “B”, etc., at the bottom of the UML package symbol indicates the product that requires that specific architectural element.

Variation points included in the conceptual structural view are shown in Figure 4. Subsystem1 is a «mandatorySubsystem» that consists of «mandatoryLeaf» components (Leaf1 and Leaf6), «optionalLeaf» component (Leaf2), «alternativeLeaf» (Leaf3 of product A and Leaf4 of product B) «optionalAlternativeLeaf» (Leaf5 of product A). In this way, variation points identify locations at which the variation will occur.

Some of the constraints that govern variability modeling cannot be expressed by the UML metamodel. They concern the following:

If a «mandatorySubsystem» only consists of «optionalLeaf» components, at least one of them must be selected during the derivation process; otherwise, a «Subsystem» that only consists of «optionalLeaf» components must be an «optionalSubsystem».

Two «alternativeLeaf» or «alternativeSubsystem» components of different products are exclusive, meaning that only one can be selected for a product. The product is specified at the bottom of the notation.

There should be no relationships between *alternative* or *optionalAlternative* components; they belong to different products.

The relationships are appropriately stereotyped to the associated components (Table 1).

Table 1. Stereotypes of relationships for variability.

<i>Stereotype</i>	<i>Description</i>
«control» «data» «uses»	Represents a <i>Control/ Data/ Uses</i> association between two <i>mandatory</i> subsystems (UML).
«control (opt)» «data (opt)» «uses (opt)»	Represents a <i>Control/ Data/ Uses</i> association between two subsystems (UML). At least one of them is an <i>optional</i> stereotype.
«control (optAlt)» «data (optAlt)» «uses (optAlt)»	Represents a <i>Control/ Data/ Uses</i> association between two subsystems (UML). At least one of them is an <i>optionalAlternative</i> stereotype.

3.1.2. Internal variation. We define internal variation only for Leaf components. A Leaf component is on the lowest hierarchical level and may perform functional requirements variable for different products.

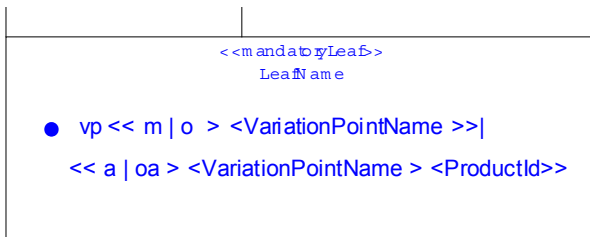


Figure 5. Internal variation of a mandatoryLeaf component.

The internal variation of Leaf components is designated by a • symbol (Figure 5). Although the symbol is not included in the UML standard, Jacobson [7] and later Webber [6] introduced the • symbol for variation points. The UML tag syntax

`vp <<m|o>><VariationName>> |`
`<<a|oa>><VariationName>><ProductId>>`

shows the reuser the parts of an internal variation so that the reuser can build a product. Mandatory (m) or optional (o) functionality (*VariationName*) of a Leaf component is specified in the tag syntax. In the case of alternative (a) or optionalAlternative (oa) the product identifier (*ProductId*) is also specified.

3.2 Conceptual behavior view

The conceptual behavior view may be mapped directly onto a hierarchy of UML collaboration diagrams. The elements of this view are roles/instances of the Subsystem stereotypes defined in the conceptual structural view.

Variable parts of a collaboration or interaction diagram can be represented with dashed lines. Optional messages between ServiceComponents use dashed lines with solid arrowheads (Figure 6).

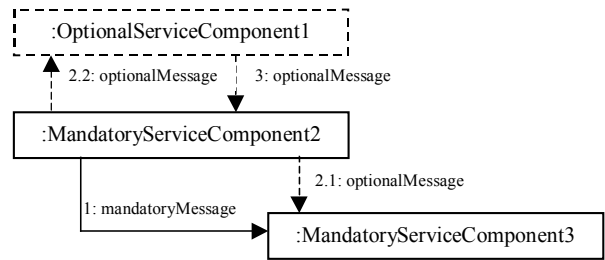


Figure 6. Optional interactions.

Collaboration diagrams describe each operation that is part of the specification requirements. Similar to the conceptual structural view, *alternative* and *optionalAlternative* ServiceComponents may be represented in this view. An identifier of the specific product that requires a particular interaction should be introduced and represented in the diagram. The notation used in collaboration diagrams for variability representation is shown in Figure 7.

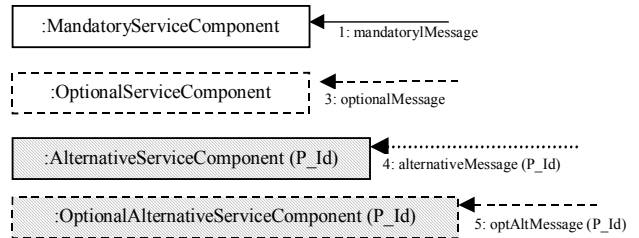


Figure 7. Variability in the conceptual behavior view.

3.3 Conceptual deployment view

In UML a deployment diagram shows the structure of the nodes on which the components are deployed. The concepts related to a deployment diagram are Node and Component. DeploymentNode in QADA is a UML Node that represents a processing platform for various services. The notation used for DeploymentNode is a Node stereotyped as «DeploymentNode». UML notation for Node (a 3-dimensional view of a cube) is appropriate for this architectural element.

A DeploymentUnit is composed of one or more conceptual leaf components. Clustering is done according to a mutual requirement relationship between leaves. It cannot be split or deployed on more than one node. The stereotype, «deploymentUnit» is a specialization of the ArchitecturalElement stereotype and applies only to Subsystem, which is a subclass of Classifier in the metamodel. The other three stereotypes «mandatory», «optional» and «alternative» are specializations of the DeploymentUnit stereotype and also apply to Subsystem.

Figure 8 describes a class diagram that defines alternative deploymentUnits. DeploymentUnitA is alternative to DeploymentUnitB; if there are at least two elements - a LeafA in DeploymentUnitA, and a LeafB in DeploymentUnitB - those exclude each other. Exclude is a new stereotype of UML association introduced in this diagram.

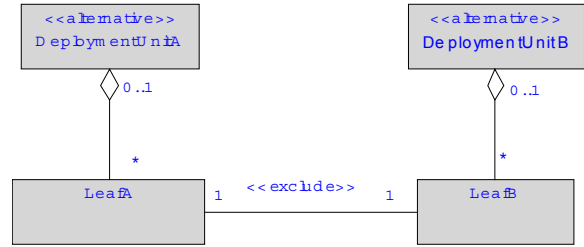


Figure 8. Alternative deploymentUnits.

3.4 Concrete structural view

The notation in this view includes a means to represent the decomposition of Capsule components. This feature allows step-by-step understanding of more and more details of the product line architecture. Decomposition is also used to show possible variations. A Capsule cannot only be decomposed into componentCapsules but can also be decomposed so that new functionality is added.

Figure 9 illustrates how the stereotypes of this view can have relationships with each other. Between abstract components we have decomposition relationships, and concrete components for particular products are obtained by specialization. The notation of Capsules specifies particular product (A) or subset of products (B, C) at the bottom of the symbol.

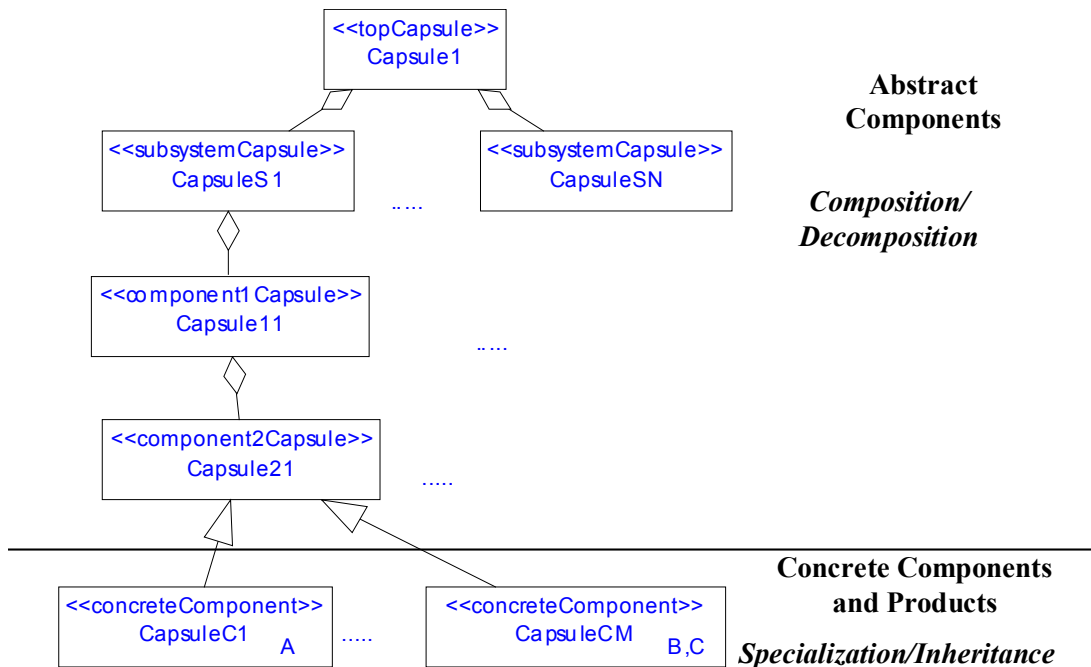


Figure 9. Structural variation in the concrete view.

Looking top-down, the AbstractFeatures encapsulated in the «TopCapsule» are decomposed into «subsystemCapsule» abstract components: CapsuleS1,..., CapsuleSN. Decomposition continues on «component1Capsule» and «component2Capsule» if necessary. In each component, abstract features of the corresponding sub-domains are collected, which are subsets of the parent domain abstract features. For each product that is a member of the family, each of the abstract components is specialized in a «concreteComponent». For example, Capsule21 is specialized in CapsuleC1,..., CapsuleCM, and so on.

The diagram includes specification of products or product sets, providing information on the reusability of each component. CapsuleC1 is modeled for product A and CapsuleCM is modeled for a subset of products {B, C}.

3.5 Concrete behavior view

The concrete behavior view of QADA is mostly modeled using two main important diagrams: a state diagram and a message sequence diagram. This view describes how the system reacts in response to external stimuli.

State machines are used with the concrete structural view's entities: capsules, ports and protocols. Standard UML state charts are applied for modeling the state machines of capsules. A particular usage of this, in combination with inheritance, facilitates reuse in modeling the concrete behavior view.

Variability is included in notation and state decomposition. As for notation, parts that are not needed in all products are represented with dashed lines (optional states) or a different fill pattern and Product_Id (alternative states). State decomposition is the other source of variants. The decomposition of a state may be shown by a small symbol in the top left corner of a state.

3.6 Concrete deployment view

The concrete deployment view of the QADA method is mapped directly on the deployment diagram of UML. UML deployment diagrams are much less well explained in the standard than other elements of UML. However, nodes - the UML elements which represent processing elements - are Classifiers in UML, which means that they can have instances, play roles in collaborations, realize interfaces, etc. They can also contain instances of components.

4. Conclusions

This paper describes how UML standard concepts can be extended to address the challenges of variability management in space of software product line architectures. In particular, a new UML profile has been defined to be integrated in a systematic approach, a

quality-driven architecture design and quality analysis method. Standard UML extensibility mechanisms can be used to express diagrammatic notations of each view of the architecture modeled using the method. The detailed description of each required extension presented in this paper would allow a possible standardization of this profile. Integrated use of a standard profile and a design method as described here would allow extensive and systematic use, maintenance and evolution of the software product line architectures.

By using UML notation extensions, our method models the variability, and hence explicitly describes, where in the PLA views software evolution can occur. A variation point specification is needed in a PLA view to communicate to reusers where and how to realize a PL-member-unique variant.

In the area of tool support a feasibility analysis of the implementation of the new UML extensions was also performed. A concrete CASE tool for software design was examined during our study to investigate whether or not it supports the new UML refinement. With smaller adaptations the required extensions can be made available in a CASE tool. In the experiment we evaluated the Rational Rose RT tool, which strongly supports capsules-based design [5]. With regard to how the tool can be configured or what other new components it needs, our evaluation showed that the conceptual views are mostly affected by the missing required extension components, but the concrete structural and behavior views need no new components supported by the tool.

References

- [1] M. Matinlassi, E. Niemelä, L. Dobrica, Quality-driven architecture design and quality analysis method – A revolutionary initiation approach to product line architecture, VTT Publications 456, VTT, Espoo, Finland, 2002.
- [2] A. Purhonen, E. Niemelä, M. Matinlassi, Viewpoints of DSP software and service architectures. To be appeared in the Journal of Systems and Software in 2003, 11 p.
- [3] OMG Unified Modeling Language Specification, version 1.4, September 2001.
- [4] Rational Rose RealTime CASE tools <http://www.rational.com/products/rosert.index.jsp>.
- [5] B. Selic, J. Rumbaugh, Using UML for Modeling Complex Real-Time Systems, White Paper, 1998.
- [6] D. Webber, The variation point model for software product lines, Ph.D Dissertation, 2001.
- [7] I. Jacobson, M. Griss, P. Jonsson, Software Reuse-Architecture, Process and Organization for Business Success. ACM Press, New York, NY, 1997.

Consolidating Variability Models

Birgit Geppert
Avaya Labs Research
Basking Ridge, NJ, USA
bgeppert@avaya.com

Frank Roessler
Avaya Labs Research
Westminster, CO, USA
roessler@avaya.com

David M. Weiss
Avaya Labs Research
Basking Ridge, NJ, USA
weiss@avaya.com

Abstract

Variabilities must be managed during many stages of product-line development including requirements elicitation and architecture design. We are reporting on our experience with mapping variabilities between requirements and architecture models that we have gathered during three different product line projects within Avaya. One of them being small-sized and exploratory in nature, prototyping new ways of IP telephony feature development, a mid-sized project currently entering implementation phase, and a third project that is of substantial size with major organizational changes involved. In all three projects, we were following a common approach of developing two separate variability models: one for capturing commonalities & variabilities during requirements elicitation (commonality analysis) and another one for realizing commonalities & variabilities during architecture design (module guide). In all three projects requirements and architecture models decomposed the system in different ways, which reflects different user and developer views, but results in many traceability issues. Further analysis, however, has shown that for the aforementioned projects, there is no need to express user and developer views in structurally different decompositions of the same system. What is actually needed are different refinements of the very same decomposition and additional statements that cut across the modules of that decomposition.

1. Context

Product-line engineering in Avaya follows a variant of the FAST software product-line engineering process [17] with project-specific extensions, if needed (such as strategic product-line engineering [5], domain assessments [9], or collaboration-based design [6]). A common aspect of product-line projects is capturing family

requirements in the form of a commonality-analysis document and using a module guide as a key component for architecture description [14][15]. Both of these model variability across the product family.

A commonality-analysis document specifies system requirements that all family members share (commonalities) and also specifies how family member requirements differ (variabilities) [17]. In terms of a set of attributes that can be used to describe the family domain, a commonality is an attribute for which all family members yield the same value, while a variability is an attribute for which different family members yield different values. We quantify each variability by specifying its value space (parameters of variation) and also decide upon its binding time, i.e., the time when the actual value must be determined (e.g., specification time, compile time).

A module guide is a key structure for describing software architectures and specifies how a system is decomposed into an information-hiding hierarchy of the sort suggested in [15]. Each module describes its secret and responsibilities. System properties that are likely to change independently should be hidden in different modules, while we expect the interfaces between modules to be stable. While a commonality analysis specifies what variabilities must be supported by the architecture, a module guide specifies how to decompose the system in order to best support these expected changes.

A module guide also represents work assignments because each module can be designed and maintained independently. Depending on the size of the product-line project¹ one or more development teams have responsibility for developing the modules. Each team must know about the commonalities and variabilities that are relevant for their work assignment(s). Tracing commonalities and especially variabilities to the

¹ Our largest-scale project currently has more than 200 leaf modules in the module guide.

corresponding architecture modules is therefore crucial to the success of a product-line engineering project.

The people who perform the commonality analysis are not necessarily the same as those who are responsible for designing the architecture. This makes traceability even more important. For our large-scale project a team of about 5 system engineers works on the commonality analysis and a team of about 10 lead developers and architects works on the module guide. It is not unusual for the two to proceed concurrently. It is therefore necessary to maintain traceability continuously between commonalities/variabilities and architecture modules while the two models develop.

In later phases we must continue to maintain traceability also. If one variability model changes (for instance, a change in performance requirements may result in a restructuring of the software architecture) then traceability is either lost or we must go back and reanalyze how requirements relate to architecture components. It is a well-known problem that when the models evolve independently, we get non-trivial relations between requirements & architecture and traceability gets more and

more difficult [1] [4] [11]. It is therefore advisable to look for ways to keep traceability as simple as possible.

2. Differences between Variability Models

Commonality analysis and module guide are variability models that serve different purposes. As already mentioned, this may result in traceability issues, i.e., the two models become difficult to compare. There are two possible ways to deal with that problem. First, we can allow the models to evolve independently and try to bridge the semantic gap by establishing increasingly complex and possibly imprecise relations. Second, we can limit the semantic gap and hence preserve traceability in the first place, i.e., create the simplest possible mapping between the two.

From our experience, traceability degrades because of structural mismatch. We believe that a sufficiently precise variability model (such as a commonality analysis or module guide) needs to introduce a module structure in order to manage complexity. If there is no clear one-to-one correspondence between modules of a commonality

General model for specifications

For our purposes, we only need a few basic notions. In [1] [10] [12] [13], for example, similar notions are used as a basis for much more elaborate models of requirement and problem specifications.

A *system* is part of the real world that is considered a unit. In a system, several *phenomena* are collected and correlated. Phenomena are, e.g., states (e.g., the water temperature read by a buoy at sea) and events (e.g., a message is sent via a communication channel). *Terms*, i.e., words in natural language are used to designate phenomena of the real world. We use terms to formulate *statements* about phenomena. Statements specify how phenomena are related to each other. An example for a statement could be: *If an incoming call is not accepted by a call-center agent within 10 seconds, the agent's work state will change to "absent"*. This statement describes a relation between the event *incoming call*, the state of a *call*, and the state of an *agent*. In addition, other terms are used. There is the value *10 seconds* and a temporal relation *within <time duration>*. Apparently, phenomena can be quite different things. There is, however, one type of phenomenon that we want to emphasize: phenomena whose value can change over time. These phenomena could be represented by timed functions or timed predicates. We will simply use *variables* for their representation. We call the set of terms and statements that describe a system, the system *specification*.

Many notations have been proposed in the literature such as function tables [12] or temporal logic [16], [2], that allow expressing statements formally.

For managing complexity, we structure a specification into several *modules*. A module in this context is a collection of terms and corresponding statements. A module declares those variables among the involved phenomena that are visible to the environment. All other variables are hidden from the environment. Variables are a means of interaction among modules. For instance, a unidirectional data flow between modules can be established, if a variable that is controlled by one module is visible to the other. *Aggregation* (submodule-of relation) is an additional relation among modules that we allow in our model for specifications.

A commonality analysis is a system specification with parameterized statements. Statements with a value space greater than one are variabilities.

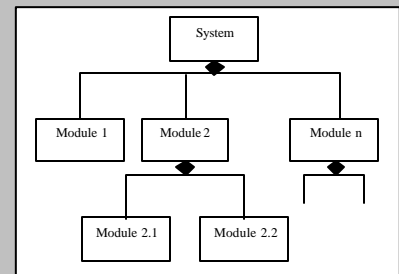


Figure 1: General model for specifications

analysis and module guide², traceability issues become inevitable. Figure 1 illustrates a simple semantic model for a commonality analysis and a decomposition into modules.

If we demand structural equivalence between commonality analysis and module guide, we avoid most traceability issues and still allow for sufficient flexibility to express differences in user and developer view. Our product-line projects suggest (Section 3) that differences in user and developer views do not present themselves as structural differences, but as different refinements of the very same structure and cross-cutting statements relative to that structure. Commonality analysis and module guide describe the same structure, but with emphasis on different aspects:

Commonality analysis:

- Every module of the commonality analysis has a counterpart in the module guide, but not vice versa.
- Commonality/variability statements can refer to single leaf modules.
- Commonality/variability statements can refer to single intermediate modules or the root module.
- Commonality/variability statements can refer to several (arbitrary) modules and describe how these modules cooperate.
- Commonality/variability statements can refer to several (arbitrary) modules and describe common characteristics of those modules (such as common look-and-feel).

Module guide:

- Module structure is complete in the sense that
 - there is no module from the commonality analysis structure that can't be mapped to a module of the module guide, and
 - it defines a useful subsystem.
- Every statement refers to single modules only.
- A statement describes a variability hidden by a module (secret), or it describes a commonality expressed by the module interface.

Apparently, even structural equivalence between commonality analysis and module guide leaves some complicated mapping issues, because, e.g., cross-cutting statements must be broken down into module-specific statement to become real work assignments. However, it seems that such mapping issues are inevitable and by far less complex than structural mismatch. Take for instance a statement about the common look & feel of the different user interfaces. This abstract statement cuts across

² We allow modules from the module guide not to have counterparts in the commonality analysis, but not vice versa.

several interface modules and must be refined into a precise specification for each of them. Another example would be a statement that describes how work items arriving at a media channel must be accepted and directed to the corresponding media processing module. Input medium and media processing vary and can be either for voice or text-based email. The statement describes a collaboration that cuts across media channel and media processing modules and must be refined into module-specific statements for the involved modules, i.e., modules for voice media processing & voice channel and modules for text processing & email channel.

3. Empirical Evidence

We have studied three different product-line projects within Avaya. One of them is small-sized and exploratory in nature, prototyping new ways of IP telephony feature development, a mid-sized project currently entering implementation phase, and a third project that is of substantial size with major organizational changes involved. For all three projects commonality analysis and module guide were developed independently and compared later on. In all three cases it turned out that structural differences were not essential for expressing different views and that a consolidation was not too difficult. We consider this important evidence that structural mismatch between variability models can be avoided without sacrificing expressibility.

- In cases where we could find one-to-one correspondence between modules from the commonality analysis and modules from the module guide a simple renaming is sufficient for consolidating the two models.
- In other cases we found subordinate levels in the commonality analysis that provided a finer decomposition than the module guide. For instance, we had a commonality analysis module dealing with agent activities at the user interface that had a one-to-one correspondence in the module guide. In the commonality analysis this module was further decomposed into submodules dealing with the different types of activities (e.g., accept work), while the module guide did not offer this refinement. The reason for this was that the two models developed in parallel and the refinement in the module guide is planned for later phases. Therefore no further action was necessary.
- In yet another case we had correspondence in lower-level modules that were aggregated differently to higher-level modules by the different variability models. For instance, in the commonality analysis we grouped domain data model, user UI, knowledgebase management, and system data model all into a high-

level module “data”. In the module guide, however, domain data model and user UI both belong to the domain behavior module, while knowledgebase management and system data model are aggregated into the platform services module. This implied a restructuring (i.e., new aggregation of the lower-level modules) of one or both decompositions. This restructuring was usually not difficult to perform because the structural difference was only minor.

- Another case that implied a minor restructuring was when higher-level modules corresponded to each other, but their submodules were not consistent. We had, for instance, a higher-level module dealing with external interfaces in both the commonality analysis and the module guide. The decomposition of this module, however, differed for both models. The commonality analysis considered additional aspects such as communication channels or resource assignments that were assigned to different higher-level modules by the module guide. On the other hand the commonality analysis also assigned some commonalities and variabilities to other high-level modules that the module guide would assign to the external interface module. A new aggregation of commonalities and variabilities solves this problem.
- And finally we had collaboration statements for which a commonality or variability specifies a requirement that does not relate to one single module, but rather describes a collaboration among several modules. One example is that all family members shall provide functionality to upgrade data schemas when data structures are changed or added. This is a collaboration that concerns modules dealing with installation, data management, as well as system data models. As mentioned earlier these collaboration statements must be captured by a separate document

To summarize we found that a consolidation was possible and only minor rework is necessary. Besides collaboration statements and not considering refinements all commonalities and variabilities could be mapped onto the module guide.

4. Conclusion

Based on theoretical considerations and practical experience, we are planning to adapt our development practice and always pursue structural consolidation while developing the variability models. Figure 2 illustrates this process. Whenever a structural change happens in one of the models it is necessary to consolidate it with the other model. The goal is to have corresponding structures in the end with the exception of cross-cutting statements that show up only in the commonality analysis and must be kept separately.

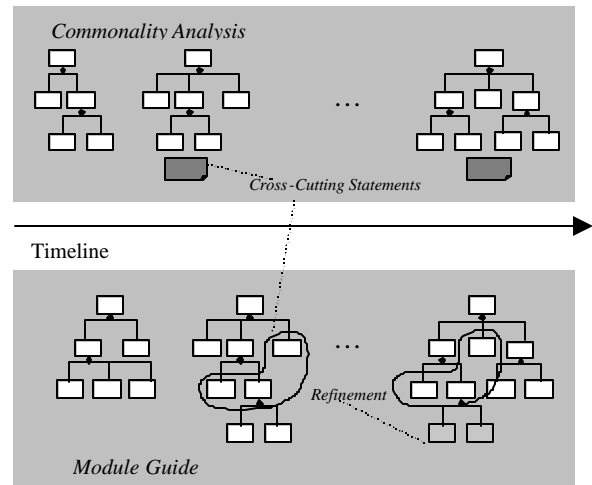


Figure 2: Developing variability models

5. References

- [1] Abadi, M., Lamport, L., *Conjoining Specifications*, ACM Transactions on Programming Languages and Systems, 17(3):507-534, 1995
- [2] Alur, R., Henzinger, Th.A., *Logics and Models of Real Time: A Survey*, In: J.W. de Bakker, C. Huizing, W.P. de Roever, G. Rozenberg (Eds.), REX-Workshop, Real-Time: Theory in Practice, LNCS 600, 1991
- [3] Bosch, J., Molin, P., *Software Architecture Design: Evaluation and Transformation*, IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, Tennessee, March 1999
- [4] Clarke, S., Harrison, W., Ossher, H., Tarr, P., *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code*, OOPSLA 1999
- [5] Geppert, B., Roessler, *Combining Product-Line Engineering with Options Thinking*, Avaya Software Symposium 2001, Denver, CO (also published in Proc. Of the 1st Int. Workshop on Product Line Engineering – The Early Steps, Erfurt, Germany, Sept. 2001)
- [6] Geppert, B., Roessler, F., *Collaboration-based Design – Exemplified by the Internet Session Initiation Protocol (SIP)*, Working IEEE/IFIP Conference on Software Architecture, The Netherlands, 2001
- [7] Geppert, B., Roessler, F., Weiss, D.M., *Specifying Requirements for Product Families: The Commonality*

Analysis, Avaya Labs Research Report ALR-2003-xxx, 2003

- [8] Geppert, B., Roessler, D.M., *A Complementary Modularization for Communication Protocols*, Research Report ALR-2002-022, 2002
- [9] Geppert, B., Weiss, D.M., *Goal-Oriented Assessment of Product-Line Domains*, Avaya Labs Research Report ALR-2003-005, 2003
- [10] Gunter, C.A., Gunter, E.L., Jackson, M., Zave P., *A Reference Model for Requirements and Specifications*, IEEE Software, May/June 2000
- [11] Gruenbacher, P., Egyed, A., Medvidovic, N., *Reconciling Software Requirements and Architectures: The CBSP Approach*, 5th Int. Symposium on Requirements Engineering, 2001
- [12] Janicki, R., Parnas, D.L., Zucker, J., *Tabular Representations in Relational Documents*, in: Daniel M. Hoffman, David M. Weiss (eds.), *Software Fundamentals – Collected Papers by David L. Parnas*, Addison Wesley, 2001
- [13] Kronenburg, M., Peper, C., *Application of the FOREST Approach to the Light Control Case Study* in: *Journal of Universal Computer Science*, Special Issue on Requirements Engineering 6(7), pp. 679-703, Springer, 2000
- [14] Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules*, *Communication of the ACM*, 15(12), 1972
- [15] Parnas, D., Clements, P., Weiss, D.; *The Modular Structure of Complex Systems*, in *Software Fundamentals*, D. Hoffman and D. Weiss, Eds., Addison Wesley, 2001
- [16] Pnueli, A., *The Temporal Logic of Programs*, In 19th Annual Symposium on Foundations of Computer Science, 1977
- [17] Weiss, D., Lai, C.T.R.; *Software Product-Line Engineering - A Family-Based Software Development Process*, Addison Wesley, 1999

Managing Variability with Configuration Techniques

Andreas Hein, John MacGregor
Robert Bosch Corporation, FV/SLD
Corporate Research and Development, Software-Technology
P.O. Box 94 03 50
D-60461 Frankfurt, Germany
{Andreas.Hein|John.MacGregor}@de.bosch.com

Abstract

Many manufacturers offer a broad range of software-intensive products. In a product generation, thousands of product variants may be developed for different price segments or to account for different technologies. Variability must therefore be carefully managed to best meet each customer's individual needs. Structure-oriented configuration provides a suitable general approach to variability management, but it must be integrated into the existing reuse process to be useable. This paper motivates the importance of variability management in product development. It discusses the chances, tradeoffs, and open issues of managing software variability with structure-oriented configuration techniques, and presents our position with respect to other approaches.

1. Introduction

Software-based products are rarely only-children. Over time, sibling products evolve through demand for variants of the same product or for separate but similar products. Examined in their entirety, these products represent a family of related products – a software product line.

In recent years, software product lines have become a topic of intensive research and widespread industrial interest as they offer seductive benefits. The product line approach provides a framework for systematic development of product line assets and their reuse in product development.

According to Clements and Northrop [1], a software product line is “A set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

This paper focuses on the latter half of the product line definition: how product line products are developed using the assets (product development) and explores how the variability within and among those assets can be managed.

The central task in product development is to select a set of variable components which fulfills a particular set of customer requirements while also satisfying certain business and technical considerations (product derivation). Note that product line product development involves far more than reusing components however: “A core asset may be an architecture, a software component, a process model, a plan, a document, or any other useful result of building a system” [1].

Structure-oriented configuration is a generalized product derivation approach that is well suited to product line product development. Moreover, its language is amenable to existing tools.

The ConIPF project [2] is currently investigating how to combine the product line approach with structure-oriented configuration to produce a product development methodology that is practicable in industrial application. This methodology should address both direct product derivation as well as hybrid forms of product derivation and product development.

2. Industrial Software Product Lines

The product line approach is appropriate for various organizations. It seems especially suitable for the parts of industry that exhibit the following characteristics:

Systems, with software – The products are integrated packets of hardware and software. In the case of embedded systems, the algorithms for controlling the physics of the system must sometimes also be developed along with the hardware and software.

Hardware resource constrained – In a volume manufacturing context the hardware platform unit cost is significantly greater than the amortized development costs. The least expensive feasible hardware is therefore employed and the software must be adapted to suit.

Large number of variants – Companies can develop hundreds or even thousands of variants in a product generation, each adapted to meet special customer or legal requirements, to conform to national preferences or to international standards. Each variant is based on common assets which minimizes the effort to develop it.

A stable and well-understood technological basis – A product line requires an architecture and reusable components, which cannot be adequately defined without an intimate knowledge of the subject domain.

The fundamental hypothesis behind the product line approach is that the investment in analysis, construction and maintenance of the assets will be justified by the benefits accrued from reuse. In situations where customer projects are highly complex, the variability introduced through product line considerations must be managed efficiently. The major challenges are:

Magnitude – Requirements specifications can number in the thousands of pages. A solution may consist of hundreds of modules to be integrated and parameterized.

Traceability – It is hard to make suitable selections from the vast number of components and parameters. Safety and reliability requirements, e.g., are not trivial to trace and verify.

Interdependencies – Complex interdependencies may exist between different options, and consistency of the overall system must be guaranteed.

Change – It is difficult to estimate the consequences of a specific change in requirements. In systems development, hardware and also algorithms may be developed in parallel with the software. Software requirements may therefore change repeatedly during development.

Product lines of the scale and complexity outlined above are demanding to use, especially when a holistic approach to variability management and comprehensive tool support are missing.

3. Structure-Oriented Configuration Techniques

In order to get a grip on the product derivation problem, we first examine the individual variability management or configuration steps. A simplified process would contain the following tasks:

- Select appropriate generic components
- Set the attribute values for these components

Technical configuration methods can be used for these tasks as they employ formal languages that guarantee the solutions' correctness and completeness with respect to the models. To date, the representation language's adequacy and maintainability have been the primary problems in technical configuration. The structure-oriented approach ameliorates these concerns as follows:

Adequacy – The formal description language provides a means to structure variability hierarchically and therefore reduce complexity. Non-hierarchical constraints can be added independently to restrict variability.

Clarity and maintainability – The different types of knowledge required for describing variability are explicitly distinguished in the models: knowledge about the variable entities is separated from solution procedures and concrete tasks. Also, the representation of the knowledge is separated from its presentation (visualization).

Applicability – Structure-oriented configuration provides a general method that can be applied to different domains and development phase. Tools (so-called inference engines) are available.

Configuration techniques have been proven useful for building complex products in various technical domains [3]. They are not limited to routine cases where all variants are anticipated and the variability models are assumed to be static. Dynamic adaptation and enhancement of the models has been integrated to support non-routine configuration [4].

4. Incorporating Configuration in Product Development

A product development methodology for product lines must address the following:

- Which system attributes correspond to the customer requirements
- Which assets correspond to those attributes
- How can these assets be used to develop the product as quickly and efficiently as possible while ensuring the product's quality

While assembling product components is clearly a configuration task, the product line approach has other dimensions. Firstly, it involves the entire software development process, not just component configuration. Moreover, it also involves deciding how to reuse components when appropriate and, when not, how to develop a product through reusing other suitable product line assets. This requires identifying software artifacts that may be relevant to considerations at the code, design and architectural levels – that is, a traceability to non-component assets.

4.1 End-to-End Traceability

The product line approach attempts to provide an end-to-end traceability – from initial requirements elicitation through product release. Frequently, there is no direct connection from specific requirements to specific components, or the components do not exist in the appropriate form. This brings other issues to the fore, such as:

Feature modeling – Individual customer requirements must be aggregated to ascertain the desired product characteristics. In conventional development they can be aggregated according to many schemas. With product lines, features are used to express the user-visible variability between products.

Parameter management and product calibration – In the industrial setting, there can be hundreds, even thousands, of products, each composed of thousands of components, each having numerous interrelated parameters. At the moment, these are managed on a component-by-component basis.

From a product line perspective, the guarantee of end-to-end traceability itself is a variability management issue. Product lines can legitimately take different forms. They may differ in the level of parameterization support depending on the inherent complexity of the inter-component dependencies. But support for feature modeling is more an explicit modeling decision.

Basically, feature modeling is best used to support the transformation of customer requirements to system attributes: from problem to solution. This transformation is typically done manually by experts using undocumented knowledge. Not only can this process be error-prone, inconsistent and sub-optimal, it leaves organizations vulnerable to the loss of these experts and to unrecognized knowledge gaps. Whether or not this is acceptable or at least tolerable depends on a number of factors:

Size of the domain – For smaller domains, the complexity may remain manageable.

Maturity of the domain – In new or fast-changing domains, the state of the explicit knowledge may lag the state of the development environment, or the effort to maintain the knowledge model may be too great.

Maturity of the organization – Knowledge about the transformation of requirements to system attributes must be learned and becomes more precise over time. New organizations may therefore not possess this knowledge even when they are in mature domains.

Features can therefore be used to obtain an initial system configuration. Should the investment in feature modeling not be justified under consideration of the above factors, reference configurations of the system based on similar cases could be used as a starting point.

Product line methodologies have always recognized the existence, and separation, of variability at the problem and solution (or requirements and realization) levels [5],[6],[7]. Regardless of whether feature modeling is included, the knowledge about the variability must be made explicit. Note that this knowledge exists in the organizations anyway. The danger is that only a few have it or that it is used sub-optimally.

4.2 Traceability to Non-Component Assets

Structure-oriented representations can be used to describe the required system attributes formally. The resulting model can be traversed to identify suitable assets, but these assets need not be components. This allows the freedom to define intermediate, more abstract or incomplete representations of the components as well as to integrate development documents.

With a structure-oriented representation of the variability, it is natural to proceed top-down and to refine the configuration stepwise. Parts of the configuration process can be automated – when decisions follow from other decisions. Other parts must be handled interactively. A special case is that the models and the product line, respectively, are insufficient to obtain a satisfactory solution. This can be resolved either by customer-specific adaptations or enhancements to the individual system, or by evolution of the product line. Initial ideas of how to integrate and support development in the product configuration process are described by Hotz et al. [8].

4.3 Investment Considerations

Using structure-oriented modeling has the advantage of a mature approach and existing tools compared to other product line approaches. Available configuration engines provide a domain-independent core technology that must, however, be extended by a domain- and process-specific modeling and configuration interface.

The hybrid configuration methodology described above is especially appropriate for large and complex variability management problems that cannot be handled manually (cf. chapter 2). A reasonably stable application domain and a basically mature domain model strengthens the case for investment.

Compared with single systems development, the product line approach requires a considerable additional modeling effort in order to ensure traceability. Aligning it with the structure-oriented configuration approach best assures that this will be recouped.

5. Related Work

Clements and Northrop [1] describe software product line practices and patterns. The practice areas it introduces focus on building product lines rather than on using them. Our methodology is meant to complete the product line approach in this respect. Particularly, it integrates configuration techniques into product line product development.

Bosch et al. [7] present a collection of variability issues that the product line community encounters in practice and theory. These issues are input into the methodology development.

Günter and Kühn [4] give an overview of various configuration approaches (e.g., rule-based systems and case-based technologies). The technical problems encountered in configuration of complex products have been primarily related to the representation language's adequacy and maintainability. The structure-oriented approach addresses these concerns.

Kühn [9] introduces state-based behavior descriptions into configuration models. This allows the configuration of system behavior; an aspect which might be especially important to software-based systems. The importance depends on the level at which a system is to be configured and on the configuration goal, respectively. The organizations that we have studied have all modularized the variability without crossing the state-machine boundaries.

Felfernig et al. [10] present a way to use UML as a configuration language. Essentially, they extend UML with respect to variability representation and define the semantics of the UML constructs using a formal description language that is amenable to an inference engine. This represents an option for integrating software engineering and configuration knowledge engineering.

Czarnecki and Eisenecker [6] and Batory et al. [11] apply domain-specific languages (DSLs) and generators to produce variants from a generic implementation platform. The DSL specifies the customized applications and the generator converts the specifications into source code. This approach only handles predefined cases, however. Our methodology also addresses adaptation and evolution.

Other approaches to variability management focus on specific phases (e.g., Bachmann and Bass [12]) or build on existing software engineering techniques (e.g., Clauß [13]). In contrast, we are aiming at a general, well-founded concept for managing variability from requirements to code.

6. Conclusions

Structure-oriented configuration offers a formal method for solving configuration problems which lie at the heart of product line product derivation: the selection and parameterization of components. It guarantees consistent and complete component configurations. Commercial structure-oriented configuration tools exist already.

Product lines, however, tackle a bigger problem: they aim at a comprehensive approach to systematic reuse, where variability management plays a significant role. We therefore examined how structure-based configuration and allied knowledge management techniques can be applied to support end-to-end traceability and traceability to non-component assets. Moreover, we have emphasized their role in reducing the complexity of parameter management.

We have outlined techniques to tailor the methodology to immature organizations or small domains, thus minimizing the investment needed to instantiate a viable product line.

The ConIPF project is defining a comprehensive methodology that uses configuration techniques to improve product line product development. The main deliverable of the project will be a book aimed at practitioners which will outline the issues and challenges in introducing the methodology and our experiences in applying it in two industrial-scale experiments.

7. Acknowledgements

This paper represents the work of the ConIPF project team. We thank our colleagues from the Robert Bosch GmbH business units for their contributions to ConIPF, and our ConIPF project partners Thales Nederland B.V., University of Groningen and University of Hamburg for the technical discussions. ConIPF is partly funded by the European Commission.

8. References

- [1] P. Clements, L. Northrop: *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.
- [2] *Configuration of Industrial Product Families (ConIPF)*, Information Society Technologies (IST), Contract No. IST-2001-34438, <http://www.conipf.org>
- [3] A. Günter (ed.): *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON*. infix-Verlag, St. Augustin, 1995.

- [4] A. Günter, C. Kühn: Knowledge-Based Configuration – Survey and Future Directions. In: F. Puppe (ed.): Knowledge-Based Systems, Lecture Notes in Artificial Intelligence, Vol. 1570, Springer-Verlag, 1999.
- [5] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson: Feature-Oriented Domain Analysis (FODA). Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
- [6] K. Czarnecki, U. W. Eisenecker: Generative Programming. Methods, Tools, and Applications. Addison-Wesley, 2000.
- [7] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl: Variability Issues in Software Product Lines. In: Software Product-Family Engineering, Lecture Notes in Computer Science, Vol. 2290, pp. 13-21, Springer-Verlag, 2002.
- [8] L. Hotz, A. Günter, T. Krebs: A Knowledge-Based Product Derivation Process and Ideas how to Integrate Product Development. To Appear In: Proceedings of the Workshop on Software Variability Management, Groningen, The Netherlands, February 13-14, 2003.
- [9] C. Kühn: Modeling Structure and Behavior for Knowledge-Based Software Configuration. Workshop on “New Results in Planning, Scheduling and Design” (PuK 2000), Berlin, Germany, August 21-22, 2000.
- [10] A. Felfernig, G. Friedrich, D. Jannach: UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems. International Journal of Software Engineering and Knowledge Engineering, Vol. 10, pp. 449-469, 2000.
- [11] D. Batory, G. Chen, E. Robertson, T. Wang: Design Wizards and Visual Programming Environments for GenVoca Generators. IEEE Transactions on Software Engineering, Vol. 26, No. 5, May 2000.
- [12] F. Bachmann, L. Bass: Managing Variability in Software Architectures. ACM SIGSOFT Software Engineering Notes, Vol. 26, Issue 3, pp. 126-132, May 2001.
- [13] M. Clauß: Generic Modeling using UML extensions for variability. Proceedings of the Workshop on Domain Specific Visual Languages, Jyväskylä University Printing House, Jyväskylä, Finland, 2001.
- [14] A. Hein, J. MacGregor, S. Thiel: Configuring Software Product Line Features. European Conference on Object-Oriented Programming (ECOOP 2001), Workshop on Feature Interaction in Composed Systems, Budapest, Hungary, June 18, 2001.
- [15] M. Schlick, A. Hein: Knowledge Engineering in Software Product Lines. European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems for Model-Based Engineering, Berlin, Germany, August 22, 2000.
- [16] S. Thiel, A. Hein: Modeling and Using Product Line Variability in Automotive Systems; IEEE Software, Special Issue on Initiating Software Product Lines, pp. 66-72, July/August 2002.
- [17] S. Thiel, A. Hein: Systematic Integration of Variability into Product Line Architecture Design. In: G. J. Chastek (ed.): Software Product Lines, Lecture Notes in Computer Science, Vol. 2379, pp. 130-153, Springer-Verlag, 2002.

Supporting the Product Derivation Process with a Knowledge-based Approach

Lothar Hotz
HITEC c/o Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: hotz@informatik.uni-hamburg.de

Thorsten Krebs
LKI, Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: krebs@informatik.uni-hamburg.de

Abstract

In this paper, a product derivation process is described, which is based on specifying customer requirements, features and artifacts in a knowledge base. In such a knowledge base a model about all kinds of variability of a software-intensive systems is represented by using a logic-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, this evolution task is integrated in the proposed knowledge-based derivation process.

1. Introduction

The product line approach makes a distinction between domain engineering, where a common platform for an arbitrary number of products is designed and realized, and application engineering, where a customer product is derived based on the common platform (*product derivation process*) [3, 5]. In this paper, a product derivation process which includes both the selection and assembling of configurable assets (like requirements, features, artifacts) out of a platform and their adaptation, modification, and new development for customer specific requirements is presented.

The main 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 [2, 14]. 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*. Thus, we speak of a *knowledge-based prod-*

uct 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 partially automated configuration of technical systems, where "configuring" can be selecting, parameterizing, constraining, decomposing, specializing and integrating components of diverse configurable assets (e.g. features, hardware, software, documents etc.). With *partially automated* we mean a process where user interactions are made for specifying a configuration goal and logical impacts are made automatically by the system.

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 necessarily describe classes for an object-oriented implementation.

Summarizing, a product derivation process which is supported by a knowledge base, includes the following basic tasks:

1. Make the software, i.e. design and implement the artifacts. This is done in domain engineering, but also when changes according to specific requirements have to be realized.
2. Model all assets related to the software, i.e. customer requirements and features, as well as the software itself according to their variability facilities. This means generate the knowledge base.
3. For a specific product, the product derivation process is performed to realize the product configuration.

A major difference of configuring software and configuring hardware is that the creation of the software (or minimum parts of it in the evolution case) is closely related to the configuration process itself (i.e. point one and three have to be interchanged). This is normally not the case for hardware, where the creation of technical entities like evaluators, PC's or aircraft cabins are strictly separated from their descriptive configuration, and later changes are hardly possible.

In the following, we first describe some distinct levels which we have to deal with when describing configurable assets (Section 2). In Section 3, we present the language entities as well as their interplay in the product derivation process. Evolution aspects are discussed in Section 4. A short survey on some related work is given in Section 5.

2. Levels of abstraction

We identify three tasks 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 goal description (see [8, 10] for an example of such a language and a suitable tool). Entities of this language are further described in Section 3.

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.

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 between these aspects is expressed. The upper model describes common parts of domain specific models. Upper models are used to facilitate domain specific modeling. They reflect the phases of the product derivation process as well as their aspects. Furthermore, relations between those aspects are specified, e.g. *require* relations between customer requirements and features. This means, relations between parts of the upper-model are specified.

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 [7, 12, 13, 19] can be used for specifying an upper model, which in turn can be used for automated 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.

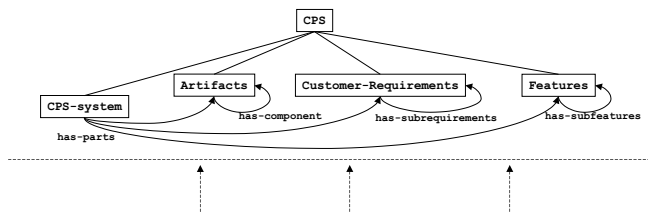


Figure 1. Example of an upper model

3. Domain specific level – What is modeled for a specific domain?

On this level, a domain specific model is created 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.

3. Entities of the knowledge based model

Basic entities of the model and the process are listed in the following:

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 restrictions 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 is used for resolving a conflict (e.g. by introducing explicit backtracking points).
3. A **goal specification** describes a priori known facts, a specific product has to fulfill.

Strategies are performed in *phases* which focus 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”¹, “take a value of the concept model” or “invoke a

¹By the means of this value determination method the *partially* automated process is realized, i.e. user interactions come in here.

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. The resulting configuration, or *final configuration* describes the product with all configurable assets (features that are included, artifacts that have to be used etc.). Sometimes this is called *product model* (do not confound that with the previously mentioned *configuration model*, which describes not one but *all* products in a generic way).

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 (constraints represent relations between concepts and their properties) 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.
 - *Calibration:* the current partial configuration might only give ranges for some parameters, which can be further specified by calibrating the real system. This calibration process can be started as a global mechanism. Its results can be stored in the partial configuration for further considering their impacts on other parameters in the model.
- **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 (e.g. if the compilation of the source files fails). A conflict means that the goal 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 [8]). Those conflict resolution methods all try to change the goal description or subsequent decisions made by the user, because they are not consistent with the current model. 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 goal descriptions and user needs. This gives a starting-point for evolution, i.e. to modify or newly develop artifacts and include them in the model to fulfill the needs (see Section 4).

Summarizing as a general skeleton 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.

4. Including evolution aspects in the process

Above, a well-known configuration process is described (see [6, 9]). 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 admissible configurations. Innovation related to this model is given if the configuration process computes a configuration which does not belong to the predefined set. For supplying a product derivation process where evolution of artifacts is a basic task, we expect to apply methods known in innovative configuration to be used.²

²A survey on innovative configuration is given in [8, 15].

4.1. Points of evolution

Following situations which come up in the process described in Section 3 indicate the necessity for evolution:

1. Anticipated evolution can partially be realized with 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 goal 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 system 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.

4.2. Evolve the configuration model

All dependencies of new concepts (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 basis of this model. For instance, the related constraints of a depending aggregate or a part-of decomposition hierarchy can be presented to the developer for consideration during the evolution of the model.

4.3. 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 already defined concepts with their parameters and relations.

- Present the specialization relations 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 details 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 supported.

4.4. 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 model and already carried out inferences stored in the partial configuration. What kind of resolution techniques are useful, still has to be developed. One trivial approach is to start the total process again with the new model and the old tasks, 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 the user input is stored in the partial configuration, only the impacts of the inference machine (e.g. constraint propagation) have to be computed again, based on the new model. Another approach is to start some kind of reconfiguration or repair technique, which changes the partial configuration according to the new model.

4.5. 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.

4.6. The kb-pd-process with the evolution task included

Summarizing, the kb-pd-process where evolution is included looks like the following:

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 evolution is started by the user.
 - (Optionally) execute the global mechanisms.
 - If a conflict occurs, evaluate conflict resolution knowledge or start evolution for changing the model.

5. Related Work

There are some approaches which try to automate software processes [17, 18]. 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 large domains [9, 11, 20], 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 product derivation process with distinct knowledge types is implemented in the tools EngCon [1] and KONWERK [8, 10]. A requirement which is e.g. not followed in [4], 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 4.3). But for maintainability and adequacy reasons it is of specific importance to separate them.

In [16] 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 a mark-up language, where the main problem is to create a document type definition that describes the documents to be used for representing software. One could see the language described in Section 3 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 more than a syntactic way (e.g. similarity-based methods or data-mining techniques) to get a real benefit of those kinds of representations.

6. Conclusion

Modeling knowledge about features, customer requirements, and artifacts and a tool-based usage of such a model yields to a partially automated product derivation process. *Partially* means that goal descriptions and user interactions are still possible, but logical impacts are drawn by the inference engine. 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.

7. Acknowledgments

This research has been supported by the European Community under the grant IST-2001-34438, ConIPF - Configuration in Industrial Product Families.

References

- [1] V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz. EngCon - Engineering & Configuration. In *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, July 19 1999.
- [2] T. Asikainen, T. Soininen, and Männistö. Towards Managing Variability using Software Product Family Architecture Models and Product Configurators. In *Proc. of Software Variability Management Workshop*, pages 84–93, Groningen, The Netherlands, February 13-14 2003.
- [3] J. Bosch. *Design & Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Addison-Wesley, May 2000.
- [4] M. Clauss. Generic Modeling using UML Extensions for Variability. In *DSVL 2001*. Jyväskylä University Printing House, Jyväskylä, Finland, 2001.
- [5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [6] 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*, pages 866–874, June 6-9 1989.
- [7] A. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Re-engineering a Legacy Product Line. In *Proc. of 2nd Software Product Line Conference (SPLC-2)*, Lecture Notes in Computer Science, pages 235–256, San Diego, CA, USA, August 19-23 2002. Springer Verlag.
- [8] A. Günter. *Wissensbasiertes Konfigurieren*. Infix, St. Augustin, 1995.
- [9] A. Günter and R. Cunis. Flexible Control in Expert Systems for Construction Tasks. *Journal Applied Intelligence*, 2(4):369–385, 1992.
- [10] A. Günter and L. Hotz. KONWERK - A Domain Independent Configuration Tool. *Configuration Papers from the AAAI Workshop*, pages 10–19, July 19 1999.
- [11] A. Günter and C. Kühn. Knowledge-based Configuration - Survey and Future Directions. In F. Puppe, editor, *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, Springer Lecture Notes in Artificial Intelligence 1570, Würzburg, March 3-5 1999.
- [12] 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.
- [13] 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.
- [14] L. Hotz and A. Günter. Using Knowledge-based Configuration for Configuring Software? In *Proc. of the Configuration Workshop on 15th European Conference on Artificial Intelligence (ECAI-2002)*, pages 63–65, Lyon, France, July 21-26 2002.
- [15] L. Hotz and T. Vietze. Innovatives Konfigurieren in technischen Domänen. In *Proceedings: S. Biundo und W. Tank (Hrsg.): PuK-95 - Beiträge zum 9. Workshop Planen und Konfigurieren*, Kaiserslautern, Germany, February 28 - March 1 1995. DFKI Saarbrücken.
- [16] R. Kneuper. Supporting Software Processes Using Knowledge Management. In *Handbook of Software Engineering and Knowledge Engineering*, volume 2, Singapore, 2002. World Scientific.
- [17] L. Osterweil. Software Processes are Software too. In *Proceedings of the 9th International Conference on Software Engineering (ICSE9)*, 1987.
- [18] H. D. Rombach and M. Verlage. Directions in Software Process Research. In *Advances in Computers*, volume 41, 1995.
- [19] 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.
- [20] E. Soloway and al. Assessing the Maintainability of XCON-in-RIME: Coping with the Problem of very large Rule-bases. In *Proc. of AAAI-87*, pages 824–829, Seattle, Washington, USA, July 13-17 1987.

Variability Analysis for Communications Software

Chung-Horng Lung
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
chlung@sce.carleton.ca

Introduction

Most software systems contain areas where behavior can be configured or tailored based on user objectives. These areas are referred to as variation points. Management of variability becomes more and more important, because it is closely related to software reuse, object-oriented design frameworks, domain analysis, and software product lines. Software variability is the ability of a software system that can be changed, tailored, or configured for specific use in a particular environment. Variability management is recognized as a critical concept in software engineering. Successful management of variability can shorten development time and lead to more flexible and better customizable software products.

Generally, the main reason for software variability management is to support reuse in a product families. Variability management could range from more formal approach based on mathematical models [Lung94], systematic methods like domain analysis, to simple programming support, e.g., inheritance in object-oriented programming languages or the `#ifdef` compiler directive. This paper, however, studies variation points and software variability from the performance point of view. Specifically, this paper deals with analyzing and building a framework for communications software for routing applications with an aim to support detailed software performance evaluations.

There are many possible alternatives for concurrent and networked software. Schmidt et. al, [Schmidt00] captured and documented a set of design patterns for this area. The book discussed alternatives in details. However, it is often still difficult to make concrete evaluation or objective tradeoff analysis based on patterns from the performance point of view due to the details we need in performance evaluation.

This paper studies various variation points for communications area. The study will be used to build a generative framework. The framework will be studied together with software performance engineering techniques, layered queuing networks (LQNs) [Woodside95], to characterize performance aspects for various approaches. The approach will provide useful guidelines for the users to choose an appropriate model or design to meet their performance requirements.

Problem Description and Approach

In distributed applications, there exist many variations. For example, there are client-server model and peer-to-peer model. For each model, there are further variations depending on specific applications and requirements, typically scalability, performance, and portability. For instance, for a server design, we may adopt a straightforward Reactive design pattern. However, the approach often leads to scalability concern. This can be improved using either Half-Sync/Half-Async or Leader/Followers pattern. For a design pattern like Half-Sync/Half-Async, there still exist further variants, as discussed in [Schmidt00].

The design patterns document general guidelines and principles for building software systems. However, for some applications, we need deeper understanding and more detailed analysis. A simple example is demonstrated here. Figure 1 illustrates the structure of the Half-Sync/Half-Async pattern. It is easy to identify a simple variation point, which is number of worker threads in the thread pool. The number can be easily configurable. Yet, from the performance perspective, it is difficult to determine the number of threads that will provide the best result. The most commonly adopted approach in industry is measurement, because there are many implementation and platform specific details involved.

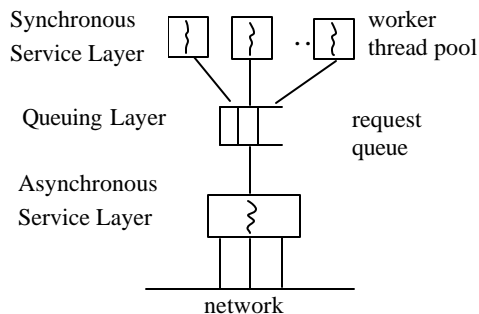


Figure 1. Structure of the Half-Sync/Half-Async Pattern

Another variation point that is more difficult to deal with is the number of request queues. Multiple queues provide more flexibility to support QoS (Quality of Service), but we need a scheduling policy to retrieve data from those queues. Moreover, we need to consider if it is better to have a dedicated thread for each request queue than a thread pool.

An even more difficult tradeoff analysis is to determine an appropriate design pattern or structure. The Leader/Followers pattern can also be used as an alternative for concurrent and networked software. There are advantages and disadvantages for each approach. Schmidt et al, [Schmidt00] discussed those issues. However, there are many questions need to be answered in order to derive an objective tradeoff analysis. On the other hand, it is almost impossible in practice to develop several alternative designs and perform thorough evaluations for each of the alternative due to resource constraints and competitions.

The main idea of this paper is to actually develop some typical alternative designs and conduct thorough performance analysis and characterization for each design. Hands-on experience is critical in building a useful framework. The process will help identify concrete variation points and the results will be useful in predicting performance and building a generative framework to support future system development.

The focus of this project is on network router software. One of the main functions of a router is to route and forward data packets. However, many

features or requirements are related to data routing and forwarding. For example, there may be different levels of QoS requirements. Each level may need a separate queue associated with a queuing mechanism. Each level of traffic may also need to be policed differently based on pre-defined policy. Even for the same level of QoS, there exist different approaches.

We did not build a system from scratch; instead, we obtained a router software system from industry. The original design of the software was similar to the Reactive pattern as shown in Figure 2. The software process contains a main thread. When a router receives a packet from the network, the packet is stored in a kernel buffer. The main thread will then read packets from the buffer and process them and put them in a destination queue. There is a dedicated thread for each destination queue to forward the packet to an adjacent router. The `select()` function is used to demultiplex a set of socket handles.

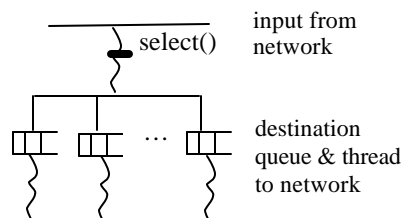


Figure 2. The Structure of a Router Software Process

Figure 3 illustrates our initially modified design based on the Half-Sync/Half-Async pattern. The software process now contains several threads. Multiple threads cannot use the `select()` function concurrently to demultiplex a set of socket handles because the operating system will erroneously notify more than one thread calling the `select()` function when I/O events are pending on the same set of socket handles [Steven98]. Therefore, there is only one thread for this layer to properly read data from the network. The asynchronous layer reads data packets from the network and stores them into an appropriate queue, depending on the data type. There are several worker threads in the synchronous layer. The number of worker thread is configurable. Currently, the number of input queue is static, because there are two types of data packet. The number of input queues, however, can be changed. Moreover, a scheduling algorithm is

needed among multiple queues. The scheduling policy is another point of variation.

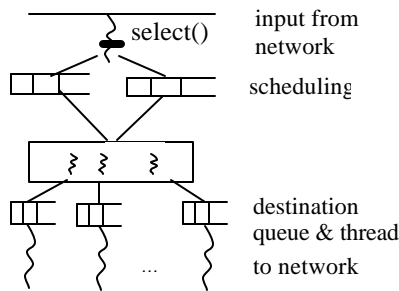


Figure 3. An Alternative Design based on the Half-Sync/Half-Async Pattern

Work in Progress

Currently, we are in the process of building another alternative design based on the Leader/Followers pattern as diagrammed in Figure 4.

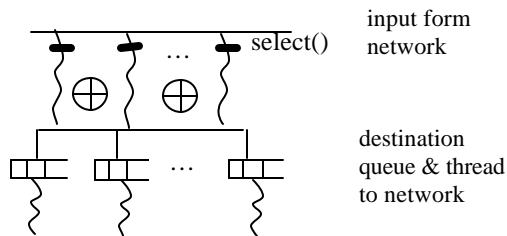


Figure 4. An Alternative Design Based on the Leader/Followers Pattern

In this design, multiple threads coordinate themselves. Only one thread at a time – the leader – waits for an event to occur. Other threads – the followers – can queue up waiting for their turn to become the leader. After the leader detects an event, it promotes one follower to be the leader. It then becomes a processing thread [Schmidt00].

The main reason that we choose to convert the original router system to the Leader/Followers pattern is that the model adopts a different design principle that is closely related to performance. By doing it, we will identify more variation points, which will provide valuable lessons in building the framework. Moreover, this design will help us better understand related performance issues.

We are also considering other alternatives. Figure 5 illustrate some examples.

We are also investigating issues associated with notation and evolution. Evolution is more complex and may be problematic for a generic system that is not well represented and designed.

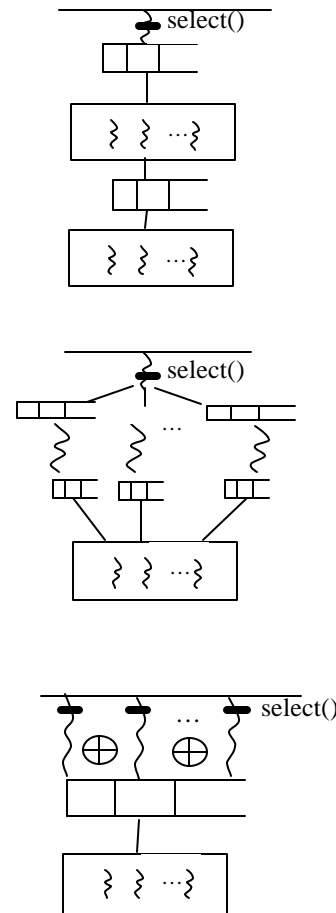


Figure 5. Other Alternatives: an Example

References:

- [Lung95] C.-H. Lung, J. Cochran, G. Mackulak, and J. Urban, "Computer Simulation Software Reuse by the Generic/Specific Domain Modeling Approach," *Int'l J. of Software Eng. and Knowledge Eng.*, vol. 4, no. 1, pp. 81-102, 1994.
- [Schmidt00] D. Schmidt, M. Stal, H. Rohnet, and F. Buschmann, *Pattern-Oriented Software Architecture*, vol. 2, John Wiley & Sons, 2000.
- [Stevens98] W.R.Stevens, *Unix Network Programming, Volume I: Networking APIs: Sockets and XTI, 2nd Edition*, Prentice Hall, 1998.
- [Woodside95] C.M. Woodside, "A Three-View Model for Performance Engineering of Concurrent Software", *IEEE Trans. On Software Eng.*, vol. 21, no. 9, pp. 754-767, Sept. 1995.

Evolving Quality Attribute Variability

S. Benarif, A. Ramdane-Cherif and N. Levy
Lab. PRISM
Université de Versailles St.-Quentin
45 Av. des Etats-Unis, 78035 Versailles Cedex
France
{sab,rca,nlevy}@prism.uvsq.fr

F. Losavio
Centro ISYS
Universidad Central de Venezuela
Ap. 47567, Los Chaguaramos, 1041-A, Caracas,
Venezuela
flosavio@isys.ciens.ucv.ve

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.

Keywords

Platform-Based-Agents, Fault-Tolerance, Monitoring, Variability.

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 [13][3].

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 [9][11][15]. 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 [1] 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 [6]. Another axe of research is the study of fault descriptions [4] and the role of event description in architecting dependable system [5]. 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 [14][16]. 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 [7].

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. 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 [2]. 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 [10]. 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” [12], and “the new revolution software” [8]. 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 cooperative view point of agent can be based on four dimensions 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 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.

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 (see Figure 1).

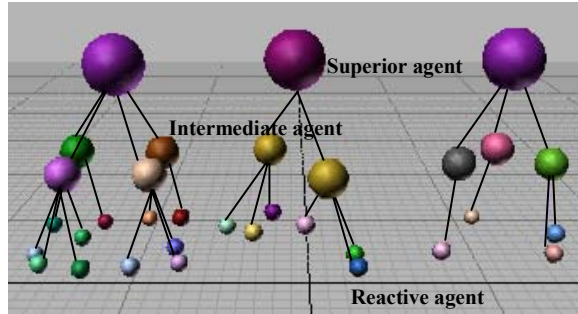


Figure 1. Topology of the layers of the platform.

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. 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.

2.2. The 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. The 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. The error can propagate through connector of software architecture by using the different interactions between the components. 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.

3.3. Detection of Faults With the Platform

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 (see Figure 2).

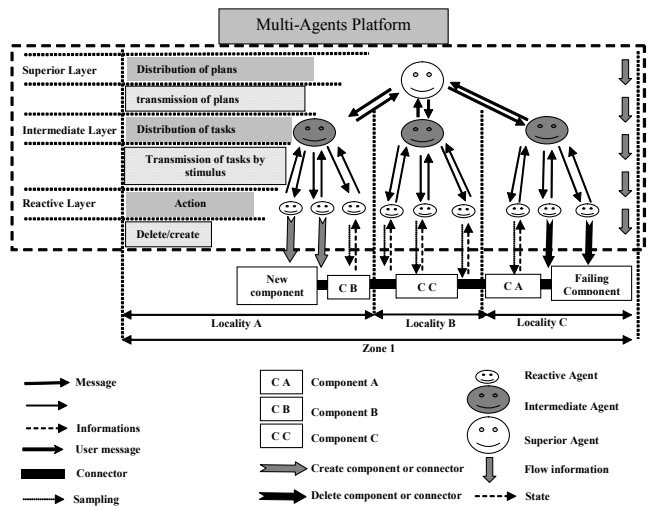


Figure 2. Multi-agents platform treatment process.

3.4. 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.

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, all of this steps descript of the first strategy of treatment process, one speaks about reconfiguration of the connection.

The second treatment process is creation of new component, it operate when 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.

4. IMPLEMENTATION OF THE PLATFORM ON PIPE-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 (see Figure 3) 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.

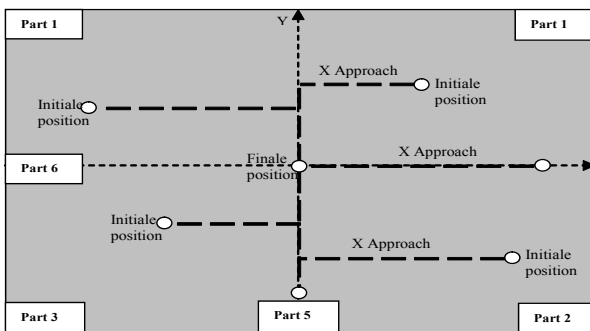


Figure 3. Strategy of navigation of the mobile robot.

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. The first component (see Figure 4), "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.

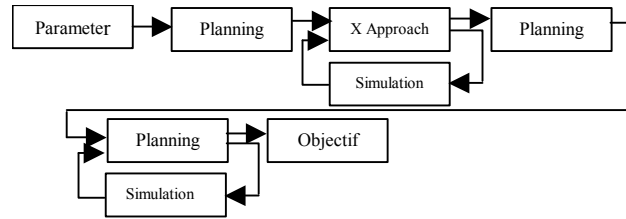


Figure 4. Pipe-and-Filter architecture for the navigation of mobile robot.

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 (see Figure 5). We will install a sensor on the robot which will help the mobile robot to detect the obstacles, when it tries to reach the finale 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.

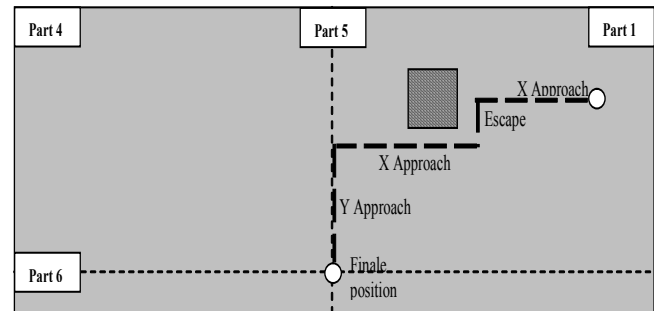


Figure 5. The navigation of the mobile robot in an environment with obstacle.

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" (see Figure 6). 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.

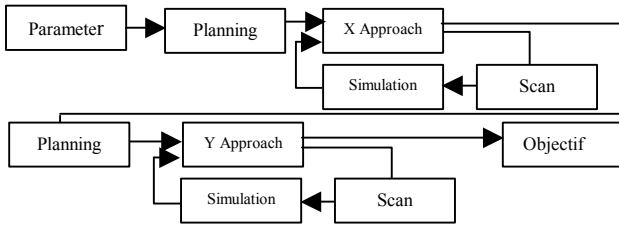


Figure 6. Pipe-and-Filter architecture for the navigation of mobile robot.

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 5), 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:

- The mobile robot will use the starting configuration of the architecture, and starts its approach X.
- 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 (see Figure 7-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 (see Figure 7-b). The information on the reconfiguration goes up towards the agent of the superior layer so that it will have a precise sight of the architecture state.
- 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 (see Figure 7-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 (see Figure 7-d). The components "Scan" and "planning" will be connected by the reactive agent (see Figure 7-e). All of these modifications are transmitted to the superior agent.
- 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 (see Figure 7-f) so that new connector will be created to connect component "X Approach" to component "Planning" (see Figure 7-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.

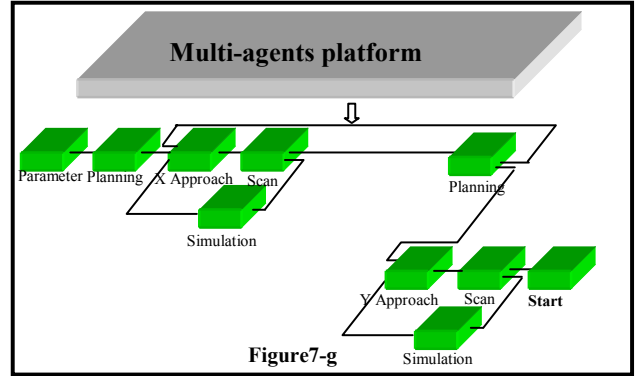
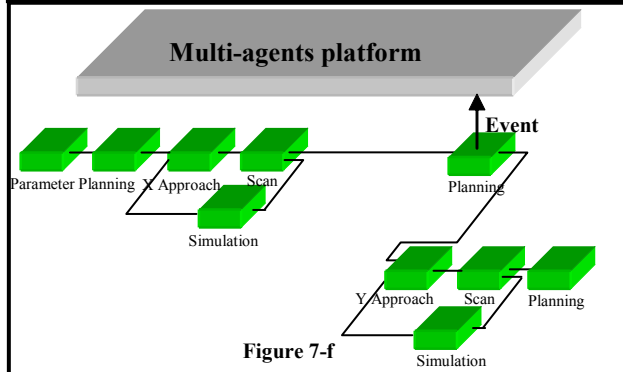
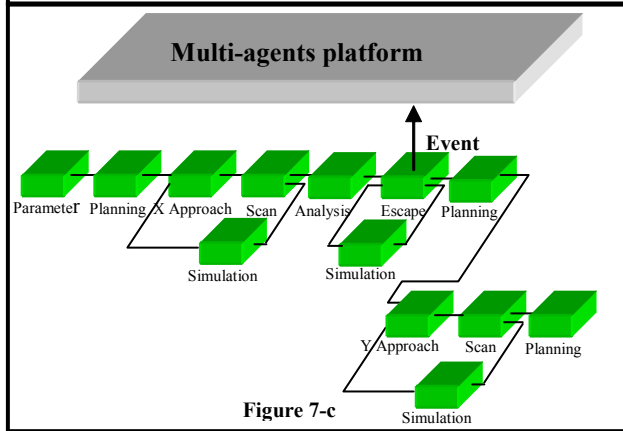
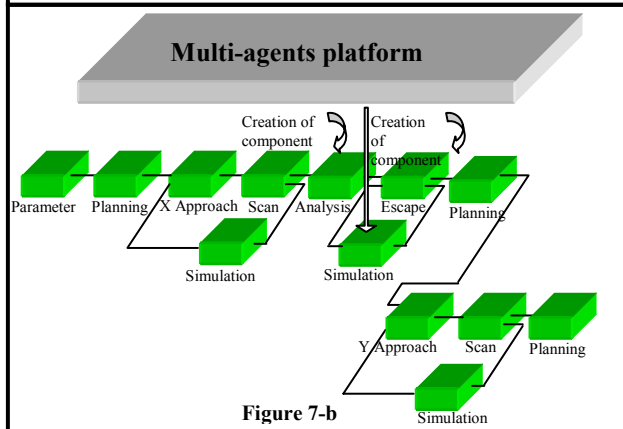
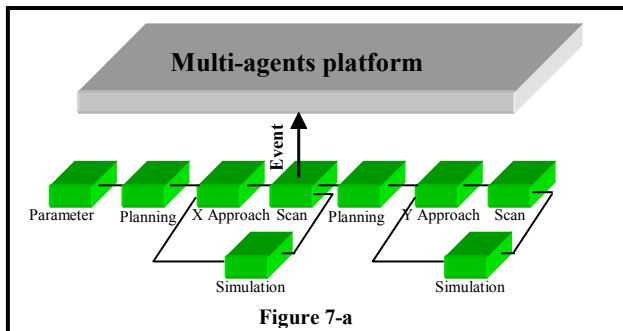


Figure 7. Scenarios of navigation of the mobile robot on an environment with obstacle.

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 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 8. Without the intervention of our platform the robot crash on the obstacle. In Figure 9, 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.

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.

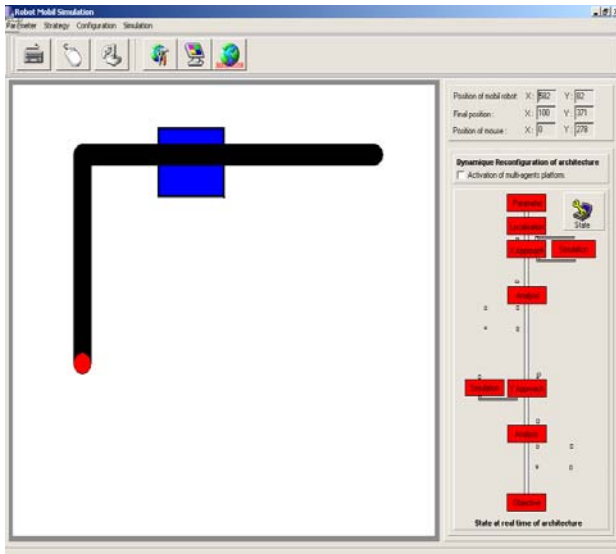


Figure 8. The crash of the robot on the obstacle without using our platform.

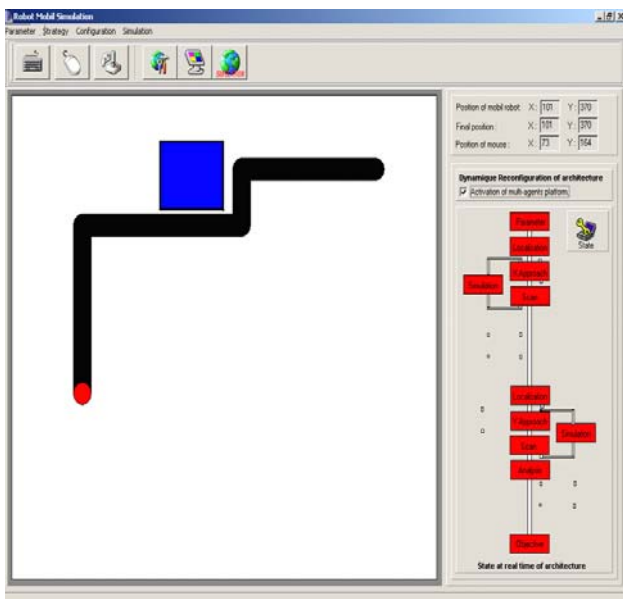


Figure 9. The mobile robot avoids dynamically the obstacle by using our platform.

8. REFERENCES

- [1] Asterio, P., de Guerra, C. et al. An Idealized Fault-Tolerant Architectural Component, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
- [2] Bass, L., Clements, P., and Kazman, R., “Software architecture in practice” SEI Series, Addison-Wesley. January 1998.
- [3] Bosch, J., Design and use of software architectures, Addison-Wesley, 2000.
- [4] De Lemos, R., and al. Tolerating Architecture Mismatches, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
- [5] Dias, M.S., and Richardson, D.J., 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.
- [6] Gokhale, S.S., and al. Integration of Architecture Specification, Testing and Dependability Analysis, In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002.
- [7] Liu, C., and Richardson, D. J., “Architecting dependable systems through redundancy and just-in-time testing. In proceeding of WADS: Workshop on Architecting Dependable Systems. Orlando, USA 25 May 2002
- [8] Ovum Report. Intelligent agents : the new revolution software, 1994.
- [9] Perry, D.E, Wolf, A.L., Foundations for the Study of Software Architecture, Software Engineering Notes, 17(4):40, Oct. 1992.
- [10] Ramdane-Cherif, A., Levy, N., and Losavio, F., 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.
- [11] Randell, B., and Xu, J., The evolution of the recovery block concept, In software fault tolerance, chapter 1. John Wiley sons ltd. 1995.
- [12] Sargent, P., Back to school for a brand new ABC. In: the guardian, 12 March 1992, p28.
- [13] Shaw, M., Garlan, D., Software Architecture. Perspectives on Emerging Discipline, Prentice-Hall, Inc. Upper Saddle River, New Jersey, 1996.
- [14] Shroeder, B., On-line monitoring, IEEE Computer, vol. 28, n. 6, June 1995. pp. 72-77.
- [15] Sloman, M., and Kramer, J., Distributed systems and computer networks. Prentice hall. 1987.
- [16] Snodgrass, R., “A Relation approach to monitoring complex systems”, ACM Trans. Computer Systems, vol. 6, n. 2, May 1988, pp. 156-196.

A Practical Approach To Full-Life Cycle Variability Management

Klaus Schmid and Isabel John

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

Abstract¹

In order to enable a smooth transition to product development for an organization that so far did only perform single system development, it is necessary to keep as much of the existing notations and approaches in place as possible.

In this position paper we propose a specific approach to the comprehensive management of variability that enables to leave as much of the existing notations and approaches in place as possible. This approach has so far been applied in several cases where PuLSE^{TM2} has been introduced into a software development organization.

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 [26] or UML-based techniques are proposed [11, 15, 24, 4, 17], specific design-based approaches have been proposed [5, 9], and of course, implementation mechanisms have been studied [19, 10, 25].

Already for some time proposals have been made for full-life-cycle management of variability using decision models [1, 18, 12]. However, these approaches are always related to a specific notation. The only exception we found so far is the Synthesis approach [13, 22].

In the context of industrial projects using the PuLSE-approach [2, 3] we needed an approach that enables us to homogenously manage variability across the different life-cycle stages, independent of the specific notation. In this paper, we will discuss this approach to variability management and the fundamental assumptions and concepts on which it relies.

2. Requirements on the Approach

We are focussing here at the situation of introducing a product line approach in an organization that so far performed only single system development. This is a quite common situation in the context of Fraunhofer IESE where we do technology transfer to different contexts. In order to facilitate the introduction of the variability concepts in such a situation we like to keep the existing notations and processes as far as possible. So for example, if an organization performs a text-based requirements process, we may often keep the text-based process and augment it with variability concepts. Later on, after the variability issues have been widely understood and accepted we then move to a more formal notation like the UML. In another case we developed an extension of a graphical notation, to which we added variability concepts by adding notational elements along with a decision model [20].

On the other hand we must support variability throughout the lifecycle. Thus, we must map the same variability in a consistent manner to the various artifacts like requirements or code, in order to widely apply it in industrial practice:

In total, we came up with the following list of requirements for our variability management approach:

- The approach needs to be notation-independent
- The approach must be applicable to all kind of life-cycle artifacts
- The approach must support traceability of variabilities both horizontally and vertically:
 - horizontally means that we must be able to trace a variability to the various places within a life-cycle artifact, where it has an impact
 - vertically means that we want to be able to trace a variability in one life-cycle stage to corresponding variabilities in artifacts of other life-cycle stages.
- The approach must support the instantiation of variability in order to support product derivation
- It must be possible to hierarchically structure the approach in order to keep the approach scalable.

We developed an approach that satisfies these requirements. This approach draws on earlier work like [8, 20].

1. This work was supported in part by Eureka $\Sigma!$ 2023 Programme, ITEA project ip00004, Caf e.
2. PuLSE is a registered trademark of Fraunhofer IESE.

3. Comprehensive Variability Management

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

1. A decision model as a basis for characterizing the effects of variability
2. An approach to describe interactions among different decisions
3. An approach to describe the relation between variation points and the specific decisions (or group of decisions) on which the resolution of the variability depends.
4. A common (maximal) set of variation types.
5. An accompanying mapping of the variability types to the specific notation to express the variation points.

Only the last point, the mapping, has to be adapted to the specific representation technique. The other parts as well as the semantic interrelation among them are independent of the specific representation approach. We will now briefly describe how these components are implemented in this approach.

3.1 The Decision Model

The decision model was initially devised in the context of the Synthesis approach for variability management [13]. In the meantime, this technique has been widely applied both in research and in industry [15, 16, 1, 14, 18, 20, 12].

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 contains.
- It does not explicitly relate to the variation points, but rather it defines decision variables which are then referenced at the specific variation points using the decision evaluation primitives.

The second characteristic makes this approach particularly notation-independent.

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 due to the fact that in practice we found cases where sets are required, but their cardinality is restricted. 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 I as a short-hand notation for $I-I$ and in this case we also write the value of the decision variable as a single value (without curly 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 can be treated as 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. [12]. 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 [23] as “binding sites”. In particular, this implies that a development decision for one system may be a runtime decision for another — a case we found quite frequently in practice.

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 [20]. In case, we want to use the decision model also as a basis for tracking implementation and evolution, it is useful to define an additional facet, describing the binding times already supported by the implementation. This may of course include “not yet supported” and in general the supported binding times should be later or equal than the current binding times.

1. Of course, this would usually be represented with a constant like `Min_Mem_Size := 16384`.

Using this description of a decision variable, we can define a decision model simply as a set of decision variable definitions. For practical reasons this will usually be represented by a table. However, especially for particularly large decision tables this may be impractical. To handle this case hierarchical decision models have been proposed [7].

There is another reason that may lead to splitting the decision model, which is that certain decision may be relevant only for certain binding times. In this case it is useful to define subsets of the decision model based on the binding times. In this case the decision model should be decomposed into parts that can be directly implemented, e.g., using makefiles, or preprocessor directives. However, as these notations do not support the full range of information that we require, it will always be necessary to keep additional information in the form of comments in these representations.

The decision model provides the basis for describing the concrete products, as a specific product can be defined by assigning values to the decisions in the decision model, where the constraints among the decisions determine the possible values. With these assigned decisions the variation points related to the decisions can be instantiated.

3.2 Decision Evaluation Primitives

As a basis for describing the relationship both among different decisions (in the decision model *relevancy* and *constraints*) as well as the relationship of a variation model to a concrete decision, we need to be able to describe more complex evaluations of the decisions. The basic constructs for describing these evaluations are called the decision evaluation primitives. These primitives support three tasks:

- The description of the *relevancy* dependencies
- The description of the *constraints* dependencies
- The description of the relation between variation points and the specific decisions

Thus, they support the components two and three of our approach. It would also be possible to use three different approaches to satisfy these needs, however, it is of course more practical to use an unified approach.

The following list provides some 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)

In addition logical relations (AND, OR, NOT) are used. Regarding the different contexts in which we use these evaluation primitives we do further differentiate:

- For describing the *relevancy* dependency we need to derive a boolean value, i.e., a *logical expression*. If for specific values for a product this value is evaluated as *true*, we need to determine a value for the correspond-

ing decision variable for this product.

- For describing the *constraints* we need to built *relational expressions*, involving the specific decision variable. Thus, the constraint $\text{MEM_PRESENCE}=\text{TRUE} \rightarrow \text{MEM_VALUE} > 100$ as part of the description of MEM_VALUE would restrict the possible values of MEM_VALUE if MEM_PRESENCE would be true.
- Finally, for describing the relation between variation points and decision values, we need both *logical expression* and *value expressions* (e.g., integer, enumeration), depending on the specific variability type.

Thus, depending on the specific context, slight variations of the underlying language might be used, we always use the same basic set of operators.

A key task of the decision evaluation primitives is to relate a decision to a variation point. We usually do not directly relate the impact of a decision variable to the variation points as 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.

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

3.3 Supported Variability Types

Many different variability types have been proposed in literature: optionalities, alternatives, set-optionalities (a set of options may be selected), etc. Based on our practical experience we deem the following types of variability to be the most relevant. They are neither minimal nor do they cover all proposed concepts, but they have been sufficient from a practical point of view:

- **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
- **set alternative:** only a single instance may be selected out of a range of possible alternatives
- **set selection:** several variabilities may be simultaneously 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.)

All these variability types are mapped to concrete representations in the context of a specific notation. Optionality and alternative use logical expressions to determine the specific instantiation that shall be made. Set alternative, set selection, and value take a value expression as basis. In addition set alternative and set selection take values as labels in order to describe the variabilities that should be part of the instantiation.

3.4 Representation-Specific Mapping of The Variation Points

The concepts we discussed so far are representation-inde-

pendent. However, we need to represent the variation points in the various life-cycle artifacts (domain model, code, etc.), which employ a specific specification technique. Therefore we need to map the different types of variabilities to the target notation.

As we discussed in a companion paper [21], the specific notation for the variation point may be graphical, textual based, or on any other basis. The different variability types should be mapped in a homogenous manner to the specification language. For each variability type a unique mapping must 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 the variability types to the target specification mechanism must be adapted for the different formalism. We will discuss below a textual mapping, which can be used as a basis for domain modeling and a mapping to an implementation in C. Both approaches are used in industrial transfer projects.

3.5 Discussion of the Approach

The approach outlined above is sufficient to describe all common forms of variabilities and dependencies among them that we encountered so far in industrial practice.

Dependencies like “requires” can also be modelled, and they are modelled on the level we believe to be the most adequate: they are made explicit on the level of the decision model in the form of constraints on the possible values of the variable.

4. Examples for 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.

4.1 Experiences with a Text Based Representation

Our variability management approach has been applied in practice 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

requirements 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 3.1 was introduced. This was realized using an Excel-table. A sanitized version of such a table excerpt is shown in Figure 1.
- We used the decision evaluation primitives shown in Section 3.3.
- 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 the mapping of 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 >>.
```

Here *expr1* is a logical expression. If it evaluates to true for a specific product (i.e., for the decision variable assignments for a specific product), then *text* is included in the instantiated product description.

Similarly, for set alternative variability, we use the term:

```
<<alt expr1 / value-1 / text1
      / value-2 / text2
      ..... >>.
```

Again, *expr1* is a value expression, while *value-1..-n* are values that are in the possible range of *expr1*.

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 1. Example of a decision model

<<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 2. An example using the textual notation

For set selection variability we used the same schema. However, we found that it is usually sufficient to use a decision variable as a basis. In order to express this case we introduced the keyword mult:

```
<<mult expr1 / value-1 / text1  
/ value-2 / text2  
.....>>
```

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

Using this approach we described the product line model. Figure 2 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.

4.2 Implementation Representation

Similarly we can describe the mapping to an implementation. As an example, we use the mapping to a compile-time binding, based on the C-language. The obvious approach for this is to use the C-Preprocessor. The preprocessor provides macro capabilities that can be used to select the code that is compiled. This language is very restricted. It allows to use *#define* to define a macro (which may contain parameters). It also allows to test for complex expressions using *#if* and whether a variable is defined (*#ifdef*). In addition an *#include* directive allows to include a large part of C code at the corresponding position.

Using our approach we can represent individual decision variables as precompiler variables. This technique allows to some degree to map also the decision model itself to preprocessor directives. The relevancy-section of a decision variable can in this case be mapped by defining the preprocessor variable only in case the corresponding decision variable is relevant. The constraints will usually only be mapped, if they lead to a unique value for a decision

```
#if (Time_Measurement = Software)  
start_software_clock();  
#else  
start_hardware_clock();  
#endif  
  
#if Memory_Size == 0  
get_no_memory();  
#elif Memory_Size == 10  
get_small_memory();  
#elif Memory_Size = 100  
get_medium_memory();  
#else /* Memory_Size = 1000 */  
get_large_memory();  
#endif
```

Figure 3. Example using the C preprocessor

variable, as in this case it can be automatically be defined.

The description of dependencies on the decision variable values for a specific product is then described using the *#if* directive. So we can map in a straightforward manner optionality and alternative. In order to map a set alternative we can use the *#elif* construct as shown in Figure 3. The set selection can not be directly translated. Rather, we actually need to break it down into the different specific cases. Finally, the value reference can be used in a straightforward manner, as this is automatically resolved by the preprocessor.

In addition, it is of course pretty common to use the *#ifdef* directive as a shorthand notation in the context of boolean decisions.

In Figure 3, we illustrate the translation of an optionality (Time_Measurement) and of a set alternative (Memory_Size). Both examples are based on the decision model given in Figure 1. As the example illustrates while the translation into the C preprocessor language is possible in order to implement the different variability types, it can be cumbersome and the expressiveness of the C preprocessor poses some restrictions. This can be improved using more powerful preprocessors. To some extent also constraints from the decision model can be implemented using the preprocessor, however, in our example, we do simply assume that permissible values are initially provided to the preprocessor.

5. Conclusion

In this paper we described an approach to variability modelling in a product line context. 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 can be applied systematically throughout the software lifecycle with a large range of artifact types.

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 further define the formal basis upon which this approach relies.

6. 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] M. Coriat, J. Jourdan, and F. Boissourdin. The SPLIT Method. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pp. 147-166, Kluwer Academic Publishers, 2000.
- [8] 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.
- [9] O. Flege. System family architecture description using the uml. Technical Report IESE Report No. 092.00/E, Fraunhofer IESE, 2000.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] M. Kasunic. Synthesis: A Reuse-Based Software Development Methodology, Process Guide, Version 1.0. Technical report, Software Productivity Consortium Services Corporation, Oct. 1992.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] M. Morisio, G. Travassos, and M. Stark. *Extending UML to Support Domain Analysis*. ASE'00, Grenoble, France, 11-15 September 2000.
- [18] D. Muthig. *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD Theses in Experimental Software Engineering; Fraunhofer IRB Verlag, 2002.
- [19] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proceedings of the Net.ObjectDays (NODE '02)*, Erfurt, Germany, Oct. 2002.
- [20] 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.
- [21] Klaus Schmid and Isabel John. Generic variability management and its application to product line modelling. In *Proceedings of the Variability Management Workshop in Groningen*, 2003.
- [22] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*, November 1993.
- [23] Software Technology for Adaptable, Reliable Systems (STARS). *Organization Domain Modeling (ODM) Guidebook, Version 2.0*, June 1996.
- [24] 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.
- [25] J. van Gorp, 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.
- [26] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

Capturing Timeline Variability with Transparent Configuration Environments

Eelco Dolstra
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Merijn de Jonge
Technische Universiteit Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
M.de.Jonge@tue.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

Abstract

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 deployment life-cycle. Some features can be bound only at specific moments in the life-cycle, while some can be bound at several distinct moments (timeline variability). This leads to inconsistent configuration interfaces; variability decisions are generally made through different interfaces depending on the moment in the life-cycle. In this paper we propose to formalize variability into a feature model that takes timeline issues into account and to derive from such feature models configuration interfaces that abstract over the life-cycle.

1. Introduction

Managing the variability in software systems is rapidly becoming an important factor in software development. Instead of developing and deploying a “fixed” one-of-kind system, it is now common to develop a family of systems whose members differ with respect to functionality or technical facilities offered [4]. As a simple example, consider a software development environment that is delivered in a light, professional, and enterprise version, each providing increasing amounts of functionality. As another source for variability, modern systems need to run on different computing platforms and provide a user interface in different natural languages and possibly interaction styles. Finally, systems typically offer extensive means for configuration and customization during installation, startup, and run-time. Again, this extends the space of actual systems of the family.

An important reason for explicitly introducing variability into a system is to obtain reuse of software. Building a separate system for each variant means that the overall development effort and time will increase, and that time to market will be seriously affected. In addition, having multiple systems with significant overlap among them seriously affects the programming and management effort needed in maintenance.

Variability is studied, among others, in the field of system families or software product lines [2]. A common approach found there is to explicitly identify features that are common to a family of products, or specific for some of its members, and to organize them in a feature model. A feature then represents a variation point in the system for which multiple choices or variants can be made available. For each variation point, a particular variant may have to be selected to actually use the system.

A feature model specifies the variability in a system on a conceptual level. It is typically used as a basis for the design and implementation of family-common and product-specific assets. However, since feature models are not first-class citizens in development environments, the link from variability on a conceptual level to the actual implementation typically has to be maintained manually and frequently is not available at all [5].

Timeline variability To create an implementation, the feature model itself is not enough, because the timing characteristics of variation points [7] play a prominent role in the design process. For each variation point, we have to ask ourselves when (in the development, deployment and usage timeline) it should be possible to extend or reduce the set of variants and when the variation point should be bound to a particular choice. Typical examples of such decision moments are compilation-time, distribution-time, build-time,

installation-time, start-time, and run-time. To identify such moments, it is necessary to consider the needs of various parties or stakeholders that might be involved in the decision process — e.g., designer, coder, product manager, system administrator, end-user.

From such an analysis it may follow that it should be possible to make a decision at *several* moments on the timeline. For instance, if we consider an operating system it may be desirable to statically link a driver into the kernel, but also to have the opportunity to load it at startup-time. This *time-line variability* is an extra dimension to variability that is often ignored. In current practice, the issues of the timeline, decision moments, stakeholders and timeline variability are not considered, nor modeled explicitly. As a result, variability is often not orthogonal to the timeline and the variability of a system appears to have been designed in an ad-hoc fashion. Some features can be configured at install-time, others at startup-time, and still others at run-time. Moreover, the *binding time* [7] of variability is usually fixed and cannot be altered, e.g., binding a run-time variation point at installation-time.

A typical scenario is a system that supports bundling of a selection of packages from a package repository. Configuration of the packages with file system layout information is done after package selection and distribution. However, it might be desirable to configure packages *before* bundling, for example, for mass deployment on machines with a standardized file system layout. Such alternative orders for making variability decisions typically require a completely different set-up.

Configuration mechanisms The realization of variability decisions can be achieved using a wide variety of *configuration mechanisms*. For example, conditional compilation allows us to choose a particular variant during compilation by identifying whether or not a piece of code should be compiled. Likewise, we can transform existing code to introduce particular behavior. Source tree composition on the other hand, allows us to include or exclude particular (source) components [9]. For more late-time variability we can use object-oriented techniques like inheritance and abstract-coupling combined with a factory object and a parameter file that defines the correct variant to use. Alternatively, we can use run-time discovery and binding of (distributed) objects in platforms such as Corba. In practice, there is a significant distance between variability on a conceptual level (features, variation points) and the configuration mechanisms actually used.

Configuration interface A software system with variability provides a *configuration interface*, through which variability decisions are made. Ideally this interface provides a view to the system that corresponds to variability at

the conceptual level (i.e., the feature model). However, the underlying mechanisms are usually reflected in the interface to such an extent that the high-level view is obscured by low-level mechanisms. Since variability decisions are realized through many different configuration mechanisms, which are closely tied to moments in the timeline, variability appears to be treated in an ad-hoc fashion. A particular mechanism is chosen arbitrarily, or for technical reasons, rather than for support of an appropriate configuration interface. As a result configuration is not transparent, but determined by the time of configuration.

Implementation of timeline variability Another issue of current practice and mechanisms is that changing the timing-characteristics of a variation point involves a lot of work, typically because the implementation of the variation point is scattered across many different artifacts (source code files, build files, et cetera). For example, allowing “pre-binding” of run-time variation points during installation (e.g. making a partially parameterized version of an X Window System server configuration) may involve a lot of work in different parts of the system. Finally, the consequences of a particular choice, e.g., in terms of performance, resource overhead, or maintainability, are often unclear or not considered. As a consequence, potential techniques to optimize the software at a particular binding time may not be fully used.

Contribution In this paper we demonstrate the problem of timeline variability and the configuration issues related to variability in general. The central idea is to provide a general formal model of variability that can cope with timeline aspects. Such a model can be annotated with *actions* to perform configuration transitions depending on the configuration state of the system (i.e., the “moment” on the timeline). From such a model we can generically derive configuration interfaces that close the current gap between variability at the conceptual and implementation levels.

Outline In section 2 we provide some concrete examples of timeline variability and their impact on the developers and users of a system. In section 3 we describe a general model of feature models. We describe in section 4 how configuration interfaces can be obtained generically from the feature models by annotating the models with actions. We discuss related work in section 5. Concluding remarks and directions for future work are given in section 6.

2. Motivating Examples

In this section we show some examples of variability in real systems. In particular we are interested in the impact of

variability on configuration of the system, and in the presence and implementation of *timeline variability*—the phenomenon that certain features may be selected at *several* different moments on the timeline.

The Linux kernel The Linux kernel provides the basis for several variants of the GNU/Linux operating system. 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 loaded 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 realized through the module system is for the most part straightforward. For example, operations on 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*. That is, they must be made known to the system, and this presents difficulties. Slightly simplified, every module exports an initialization function *f* which must be called during kernel initialization, in the case of statically linked modules, or at module load-time, in the case of dynamically loaded modules.

For dynamically loaded modules, obtaining the address of *f* is a matter of looking it up in the module's symbol table at load-time. For statically linked modules, the problem is harder, since the C language 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. This problem is solved by emitting these addresses in a specially designated *section* of the executable image, which can then be iterated over at run-time. The point here is not to show the details of the implementation of timeline variability in the Linux, but rather to show that it is non-obvious and quite different at each point on the timeline. In this case, we achieve timeline variability of module activation at build-time and run-time, through a combination of preprocessor, compiler, and linker magic.

Another issue is how variability appears to the user. A problem with systems that allow configurability at different moments on the timeline is that the configuration interface tends to be different at each conceptual moment. For exam-

ple in the case of the Linux kernel, modules are added at build-time through an interactive tool that allows variation points to be bound through a textual or graphical user interface based on a feature model of the kernel. On the other hand, at run-time the interface is more primitive: adding a module happens through commands such as `modprobe` which simply takes the name of a module to be loaded.

The Apache web server 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 at build-time, 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 (that is, 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. Note that this solution is again, in a sense, outside of the C language; we need to *generate* C code (a process external to the language proper) in order to deal with these open variation points. Registering a module at startup-time happens by loading the module and lookup up a fixed name in its symbol table.

Configuration is quite different at build-time and startup-time. At build-time, modules are selected by specifying the list of desired modules to an Autoconf configuration script (which constructs the build files). If modules are added later, at startup-time, they must be added to a configuration file (`httpd.conf`).

Issues The aforementioned examples demonstrate the two main issues in implementing timeline variability. First, we tend to have a *different configuration interface per configuration moment*, even when some features can be bound at several moments during the life-cycle (thus presenting an inconsistent interface to the user).

Second, *implementations techniques are ad hoc*. This is almost necessarily so, because the underlying languages do not offer the required support. Providing a variation point *either* at build-time *or* at run-time is not hard, but providing it at both tends to require some hackery. Consider, for example, a binary variation point that is bound at run-time, implemented in C. This might be implemented as follows:

```
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 to allow for this feature to be bound both at build-time and run-time, we would need, e.g.,:

```

#if FEATURE_BOUND_AT_BUILD_TIME
#if FEATURE
    f()
#else
    g()
#endif
#else
    if (feature) f() else g()
#endif

```

which is inelegant: not only do we need two different implementation mechanisms, but binding the feature will tend to happen through different configuration interfaces, and the corresponding implementation is difficult to understand.

3. Feature models and time

The main problem in timeline variability 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 order to generalize system configuration, we propose that a generic configuration interface is parameterized with a formalized feature model.

In approaches such as FODA [10] or FDL [6] feature models are described as graph-like structures, where the edges between features denote certain relationships (such as alternatives, exclusion, and so on). 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 [14] language was designed to drive the configuration process of the Linux kernel (and other systems) 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. That is, we should not place constraints on configurations but on transitions between configurations.

Formally, a feature model for a system with a statically fixed set¹ of variation points has the following elements:

¹It is possible for the set of variation points to be dynamic, e.g., loadable modules may add their own variability. For simplicity we do not take this possibility into account here.

- 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 an assignment of states to variation points, 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 is the transitive closure of $\{c_0\}$ under the T relation.

Note that static feature models such as FODA, FDL, and CML 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 an usability issue; the model is as described above.

It may be argued that implementation restrictions should not appear in the feature model. 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. An example would be a system administrator who restricts some end-user configurability. The specification language for the feature model should allow these constraints to be specified separately.

Example An example may be useful. We shall encode a very small subset of the variant space of the Apache web server using the formalism given above.

What is the set of variation points P and the associated sets of states for each variation point? An example of a simple variation point in this model is `debug` to enable or disable emission of debug information (with states `on` and `off`, respectively). This variation point can only be bound at build-time. More relevant to timeline variability is Apache's module support. For example, we have variation points such as `mod_cgi` (also with states `on` and `off`) to enable or disable support for CGI scripts, respectively. Recalling the discussing of modules in Apache in section 2, such features can always be bound at build-time, but they can only be changed at startup-time when support for dynamic loading of modules is enabled. This is also a variation point, of course, which we denote as `mod_dso` (for *dynamic shared objects*).

In order for our approach to work we need to encode in the valid-transition relation T that the state of `mod_cgi` can be changed up to build-time, but up to startup-time only if `mod_dso` is set to `on`. Hence, we need to be able to distinguish the point on the timeline that we are at. To do this we introduce a *pseudo variation point* time with states `initial`, `built`, and `running`, denoting the deployment points at which the source has been obtained, the system has been built, and the system has been started.

Note that there is nothing particularly special about time, except that it does not denote a real (i.e., conceptual) variation point; i.e., this is quite general: any aspect of the deployment state (such as an installation path) can be stored in the configuration.

Using time we can fill in T , which describes the set of valid transitions. Hence, we have to deal with *two* configurations: the configuration c_1 we are coming from, and the configuration c_2 that we are going to. For any c_1 and c_2 , $(c_1, c_2) \in T$ if and only if the following hold:

- (1) $c_1.time \leq c_2.time$
- (2) $c_1.time \geq built \rightarrow c_1.debug = c_2.debug$
- (3) $c_1.time \geq built \rightarrow c_1.mod_dso = c_2.mod_dso$
- (4) $c_1.time \geq built \wedge c_1.mod_dso = off$
 $\rightarrow c_1.mod_cgi = c_2.mod_cgi$
- (5) $c_1.time \geq running$
 $\rightarrow c_1.mod_cgi = c_2.mod_cgi$

(The ordering on time is `initial` \leq `built` \leq `running`). Condition (1) encodes that time is monotonically non-decreasing. Conditions (2) and (3) specify that the `debug` and `mod_dso` variation point can never change after the system has been built. On the other hand, condition (4) says that `mod_cgi` cannot change if, additionally, support for dynamic loading is disabled. Hence, `mod_cgi` can change after build-time if DSO support is enabled. Finally, condition (5) restricts this a bit: CGI support cannot be changed after the system has been started.

4. From models to configuration interfaces

The construction of a formal feature model as discussed in the previous section is valuable in itself because it enables analysis of both conceptual *and* implementation-defined variability in a system. The real strength of a formal model, however, is that it allows the automatic generation of configuration interfaces. The intent is that using the feature model we can drive a generic configuration tool called *TraCE* (for *Transparent Configuration Environment*). The idea is outlined in figure 1. At each point in time we maintain the configuration state corresponding to the state of the system. Starting with an initial system (e.g., the source code distribution of Apache), the user can make modification to the configuration through the TraCE user interface, which

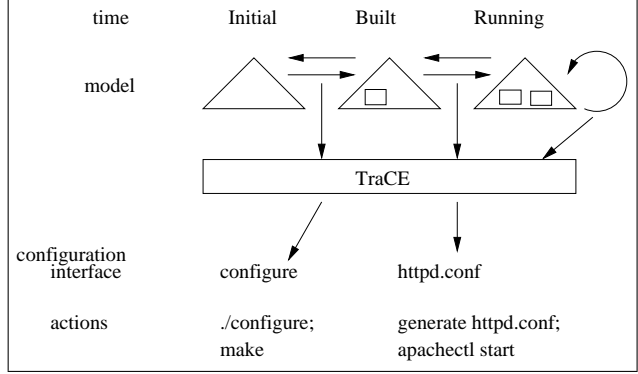


Figure 1. Sketch of the TraCE system operating on a feature model for Apache.

presents a visualization of the feature model. Such modifications are actualized upon the system by TraCE. For example, in the Apache example, the transition from initial to built stage is performed by configuring the source with the right parameters (depending on the selected variation points) and building it.

More precisely, given a *current* configuration $c \in C$, the user can modify c by changing the states of variation points, yielding a *target* configuration $c' \in C$. We associate with each valid transition $t \in T$ some imperative action that should be performed to *realize* the configuration transition. Hence, if $(c, c') \in T$, the configuration c' can be realized by executing the associated action. Note that actions are associated with transitions, and so configurations may be realized in different ways depending on the configuration we are coming from. This is necessary for supporting timeline variability, since the binding of variation points can proceed through different implementation points depending on the time, or on the state of other variation points.

A problem here is that while (c, c') may not be a valid transition, there may be a sequence of transitions $(c, c_1), (c_1, c_2), \dots, (c_n, c') \in T$ that realizes the desired transition. Finding a path in the transition space is computationally prohibitive. We can side-step this problem by requiring that the user always specifies transitions that are in T . This is not unreasonable if the developer of the feature model ensures that T is (more or less) transitive, that is, $(c_1, c_2) \in T \wedge (c_2, c_3) \in T \rightarrow (c_1, c_3) \in T$.

5. Related work

Variability is an emerging area of research. The first attempts to handle variability in a disciplined way are the feature-modeling formalisms originally developed in [10]. These models are directed at domain analysis, however, and are not directly used for implementation. Rather, such mod-

els suggest where in the system the implementor should construct variation points to deal with anticipated or unanticipated variants. In [16] another feature modeling is addressed, which uses feature logic to reason about collections of components and their properties. Basic support for timeline variability is addressed in [13]. They use partial evaluation techniques of components parameters to choose between compile-time and run-time variability. Variability mechanisms are described in [7]. They introduce the notion of variability binding time (i.e., the moment in time where a variability point is bound) but binding time is not explicitly modeled nor transparently handled. Feature binding cannot be rolled back in product instances to change parts of its functionality. Variation management in software product lines is discussed in [12]. They discuss variation during the life-time of a product line rather than during the deployment time of a product instance.

Several techniques have been developed to realize variability at compile-time, such as Frame Technology [8], Mixin layers [15], and aspect-oriented programming [11]. None of these explicitly model variability. GenVoca is another compile-time variability mechanism [1]. Feature modeling in combination with GenVoca is briefly addressed in [3] but timeline variability is not considered.

6. Conclusion

We have discussed some of the issues in timeline variability. We suggest that the problem of inconsistent configuration interfaces can be solved through formal feature models that encode timeline aspects and that these can be used to generically drive the configuration process.

We are currently implementing a prototype of TraCE. There are several important issues that must be addressed. First, we need a language (or interface) that allows the efficient formulation of feature models, as well as the association of actions to transitions. Second, there are user interface issues. For instance, how do we present the feature space to the user? In formalisms such as CML2 or FDL the presentation structure is more-or-less obvious (due to the use of an essentially tree-like model structure). In TraCE we need to automatically derive an appropriate presentation structure from the feature model.

The other main problem in timeline variability—implementation techniques—deserves study; e.g., language mechanisms and programming techniques that allow easier binding at several moments must be investigated.

References

[1] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-

specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, 2002.

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

[3] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In *ESEC/FSE '99*, volume 1687 of *LNCS*, pages 2–19. Springer-Verlag, 1999.

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

[5] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings: Second Software Product Line Conference*, number 2379 in *LNCS*, pages 217–234. Springer-Verlag, Aug. 2002.

[6] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[7] J. van Gorp, 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*, pages 45–54. IEEE Computer Society Press, 2001.

[8] S. Jarzabek and R. Seviora. Engineering components for ease of customization and evolution. *IEE Proceedings – Software Engineering*, 147(6):237–248, Dec. 2000. A special issue on Component-based Software Engineering.

[9] M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, Apr. 2002.

[10] 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.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, June 1997.

[12] C. Krueger. Variation management for software production lines. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *LNCS*, August 2002.

[13] R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.

[14] E. S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *9th International Python Conference*, March 2001.

[15] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.

[16] A. Zeller and G. Snelling. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, Oct. 1997.

Component Interactions as Variability Management Mechanisms in Product Line Architectures

M S Rajasree ,D Janaki Ram
Distributed & Object Systems Lab
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai, India
rajasree@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in

Abstract

Software product line is a collection of products that share a common set of features and allows for controlled variations. The common assets across all the products can be captured as a reference architecture referred to as the Product Line Architecture (PLA). PLA is a configurable architecture. It can accommodate variabilities in the hot-spots provided in the design. The interactions among the assets play the key role in designing the architectural instances for products with specific requirements. In this paper we discuss how component interactions and the resulting patterns of interactions can serve as powerful variability management mechanisms in PLA design.

Keywords: *Product Line Architecture, Patterns, Components, Connectors*

1. Introduction

Product Line Architecture (PLA) is a design for a family of applications. PLAs attempt to capitalize the domain expertise of companies and facilitate large-scale reuse in a systematic way. The intent of a PLA is to amortize the effort of development for similar products. So, it is very important to plan the design stages of PLAs. Also, the up-front investment in the development of PLA is high compared to single product development. Hence, enough care should be taken to see that the architecture is capable of addressing the development of a large number of products with less development effort. PLAs should also be capable of addressing easy evolution of assets.

When a PLA is built for the first time, intensive domain engineering is needed to capture the requirements, reusable assets and variations. After this, the reusable assets could be identified and subsequent generation of the architecture

can follow a bottom-up approach in composing these assets to give the product instances.

It is advisable to have a generic, configurable architectural model for a PLA, that is capable of generating individual architectural instances, provided with required specifications. This architectural model should be composed of abstractions in the form of patterns. The variabilities in the individual architectures and their interactions should also be captured in the form of patterns.

In this paper we discuss how the interactions between the reusable components can serve as powerful vehicle in handling variability in product lines. We believe that since the interactions can dictate the semantics of customizing the participating components, they can serve as more powerful variability handling mechanisms. Also, the patterns of these interactions can be captured as reusable artifacts in variability design.

The paper is organized as follows. Section 2 discusses importance of abstractions and interactions in PLA design. Section 3 explains connectors as variability management mechanisms in PLAs. Section 4 details interaction patterns in PLA design. Section 5 discusses some related work. Section 6 concludes the paper and gives some pointers to future work.

2. Importance of abstractions and their interactions in PLA design

Derivation of generic architecture is always based on successful abstractions and their interactions. These abstractions enable us to have a conceptual model of the entire architecture. Interactions can dictate the customization of these abstractions. A first-class explicit representation of the PLA and its assets enable us to place the entire software in a conceptual context [2]. Such a representation also enables the design and redesign of the reusable assets easy

since the unwieldy interfaces of the components are minimized in abstractions. Abstractions make the system more comprehensible and interactions between two abstractions can dictate the behavioral semantics between the two of them.

2.1. Abstractions in the form of components and connectors

It may not be possible to arrive at a uniform architecture, which contains the minute details of all the candidate architectures. So, an abstraction which does not show up the differences among the individual architectures is to be considered as the generic abstraction. This forms the reference architecture for the PLA. At a high level, these abstractions can be viewed as components which are the computational elements and connector which govern the interaction between two components [16].

Alternatives and options can be built into the generic architecture based on variability analysis. Options and alternatives refine this generic abstraction by means of the connector semantics that is used to connect the variable part with the fixed part of the design. Connectors can mediate between the two participating components in dictating the variability in the design. So we see them as powerful mechanisms in handling variability in product lines.

Initial step in arriving at the abstractions is to define the product context in which this abstraction operates. For example, if there is a product line for designing control program for various embedded systems, the hardware in which the embedded system operates forms the context abstraction. Next step is to find the abstractions themselves. These will facilitate the components as instances of the abstractions. These instances in turn will have provision for variabilities. These steps can be performed iteratively by assessing the suitability of the abstractions in realizing the generic architecture, allowing for controlled variations in it. Bounding the scope of the product line, that is, the range of products covered by the product line, is also an important activity in the initial phases.

On carefully examining the product lines, it can be seen that variance, optionality and conflicts are the three transformation aspects in their design. This basic idea should be guiding the identification of the components and their interactions. While analyzing variabilities, we should not be concentrating on abstractions which are very primitive, that is, those which have direct implementation mechanisms in the language. For these abstractions, variance is the key design force. So, future evolution of assets may not be easy. The best approach would be to start from the domain concepts and then come up with domain abstractions. This enables the management of abstractions easy in terms of the domain evolution.

For easier management and evolution, components can be grouped into various logical layers, each layer addressing one specific functionality. The core functionality of the generic architecture will be available, which different products can refine in order to have their designated functionality.

2.2. Features aspects and concerns

A feature is something especially noticeable, a prominent part of detail [18]. In the context of software, feature will be any part or aspect of a specification which the user perceives as having a self contained functional role [8]. For the user of a software it could mean one thing, for the developer, it could mean a different thing. For example, in order to have a problem domain feature implemented, the developer will be using more than one feature in the solution domain. Features will be manifested in programs as providing certain services or attaining some quality attributes. If a feature is implemented across several components in the system, it is called a cross-cutting feature. Aspect-Oriented Programming(AOP) [10] community calls such features as aspects.

The domain abstractions will be addressing the basic features in the product line architecture. It is very rare that the features stand in isolation. Once the abstractions are identified as architectural entities, global properties that are desirable for the system can emerge from the individual functionalities and the interactions that exist among them. It is equally important to ensure that undesirable interactions do not result from the features which are used to compose the system. We believe that the interconnected features have to be identified in the early architectural design stages itself and then subsequently be refined to programming level aspects. Use Case Maps(UCM) [3] seem to be a good choice at the architectural level for modeling cross cutting features [11].

Adequate documentation mechanism for defining mapping from feature space to solution space is very critical for PLAs. This will help in the traceability of requirements. Feature Solution (FS) Graph is used in [11] to connect quality requirements with solutions at an architectural level. Features may sometimes be non-functional requirements of software. Use of design patterns [7] as solutions for attaining certain quality attributes like flexibility, extensibility etc. is an example.

3. Connectors as variability management mechanisms

Connectors can be viewed as more powerful variability handling mechanisms. They have the capability to achieve various composition patterns for components that generate

a specific architectural instance. We view connectors to be more powerful than components in modeling variability because they have the potential to dictate changes in all participating components whereas a component can be customized only in its context. Achieving variability using connectors is not merely through the syntactic composition of the participating components, but also dictated by the connector semantics.

There are two levels of modeling variability using connectors. At first level, connector semantics can decide upon the variable assets in the form of components, that are chosen to compose the product architecture. At the next level, within a component it can address the second level of variability. It is to be noted that this is possible only for a white box component, whose code is available for an outside entity to modify.

On carefully examining the product lines it can be seen that variance, optionality and conflicts are the three transformation aspects in their design. Interactions between various components for each of these aspects can be achieved by properly designed connectors. Variance and optionality will mostly be addressed by the choice of specific components. Conflicts can be captured in the form of violation of contracts by the components that are participating in the composition. Say for example, there are two features which can not co-exist in a product, connector semantics should capture that as a conflict.

The instantiation of a product line architecture from existing assets by means of automatic composition of assets is gaining importance now-a-days. Given a particular requirement specification, a highly customized and optimized intermediate or end product can be manufactured on demand from elementary reusable implementation components by means of configuration knowledge using generative programming techniques [13]. The configuration patterns can be captured as reusable artifacts.

For arriving at fundamental design decisions and trade-off analysis at early stages in the design of architecture of product lines, architectural patterns [5, 9] can be used. These are collection of high-level design decisions which are already made and reused. They provide guidelines on how to connect components and connectors together. Architectural patterns basically provide the following. The functionality of components at runtime, topological layout of components as per their relations at runtime, a set of semantic guidelines, a set of connectors providing communication and co-ordination or co-operation among components.

4. Role of component interaction patterns in PLA design

Research community has addressed the use of patterns for attaining structural variability [1] based on mandatory,

alternative and optional feature properties as identified by Feature Oriented Domain Analysis (FODA) [14]. Only structural patterns have been addressed in this work. Based on whether a variable feature is optional, mandatory or alternative a pattern can be chosen to statically plugging in the variability.

The features that dictate the variations may interact with one another. So, it is unlikely that the variations are confined to a specific part of the design. The analysis for capturing the commonality and variability should concentrate on the common patterns of occurrence of variabilities. The interactions of intended variability in the architecture may not only be dictated statically. The assets may have interactions at the time of composition also. These interaction patterns, as dictated by the connector semantics forms a key role in dictating the composition of reusable assets. In PLA design, because of the probable inter-relationship between features and their realization mechanisms, interaction patterns play a vital role. So, it is not enough that we concentrate on the statically bound variabilities. We have to concentrate on the configuration patterns of components that are resulting from the composition. These patterns serve as reusable design solutions for the kind of variability they are addressing.

PLAs are intended to be long-lived designs. So, a conceptual abstraction of the entire PLA is an important requirement for its success. This abstraction can be captured as interaction patterns for realizing the candidate architectures. Use of interaction patterns is many-fold. In the automatic generation of product line architectures, these patterns dictate the desirable and undesirable forms of interactions while composing architectures from existing assets. Knowledge of standardized interaction patterns enable the learning of the architecture easier. While modifying reusable assets, the dependency between assets is very important. This dependency can not be captured fully by the static behavior of the system. Dependency between assets will be dictated during composition as well as run time. So variability will be decided based on all these dependencies.

4.1. Patterns in architectural and design level

Abstract solutions to recurring problems are available in the form of patterns at varying levels of granularity [5, 7, 9]. These solutions address the functional as well as non-functional requirements of the architecture. These patterns can be used to make fundamental design trade-offs in the architecture design. They also facilitate the evolution of PLA easier. Evolution takes place when new features are added and existing features are enhanced. Though this activity is costly, it is unavoidable in successful applications since requirements change. [6] suggest the use of design patterns and domain specific languages for achieving extensibility

in product lines.

Design patterns [7] like strategy, abstract factory, mediator and state support variants. In optionality transformation, in some products certain components can be omitted. Strategy and proxy patterns can be used for this. For resolving conflicts between PLA and individual products, adapter, proxy and mediator patterns can be used.

Patterns are capable of providing trade-off analysis in case alternate designs exist. A methodology for design of a framework for product lines based on a pattern oriented approach is available in [20]. This work uses a set of metrics suggested in [4] using an abstract model of pattern called a pattern graph.

When design patterns are used to model variabilities in product lines, we take single variable aspects into consideration. Most of the patterns address variability issues. The important point here is that the variabilities should not be treated in isolation. There has to be a mechanism bridging the gap between requirement modeling of variabilities and their design through implementation. Feature Solution Graph has been suggested as a means to indicate the relation between feature (problem) space and solution space.

It is not only adequate to represent the variabilities, their traceability is important. Undesirable interactions in the design using patterns are to be captured and such pattern combinations can not be used in modeling the solutions.

Previous research in software variability has identified variation points [12] as points where variability is to be plugged in. But we believe that the treatment of variability should be started right from the analysis phases precisely in the hot-spots identified during the analysis and design stages. Variable aspects can be designed either in the components or connectors. There are different ways of achieving variability. In some cases, it may be possible to attain variability statically bound to the program at compile time. Another alternative is to have dynamic behavior plugged in to the system by means of run time customization of components. Variability could also be achieved at composition time. An important reusable artifact in the last two cases will be the dynamic configuration or the pattern of interactions, that the system has during the behavioral change.

5. Related work

[17] initiated PLAs in early in 1969. Information-hiding principle by Parnas encodes a module's commonalities as its interface and variabilities as a module's secrets [19]. RSEB (*Reuse-driven Software Engineering Business*) [12] addresses development of application product families taking into account their organizational and technical issues. RSEB and FODA [14] are integrated in *FeatuRSEB* [15]. This reuse-oriented model serves as a catalog to link use cases, variation points, reusable components and configured

applications. In [6], a case study using GenVoca approach has been discussed to achieve extensibility of product lines. The method that we envisage differs from all these works, in that we give importance to the architectural aspects of the product line by concentrating on the interactions among the reusable assets. This approach thus helps in the automatic generation of architectural instances.

6. Conclusions and future work

A robust software architecture is critical for any product line. In this paper we have discussed the importance of interactions between components modeled in the form of a software connector in dictating the variability in product line architectures. A connector is a more powerful mechanism to achieve variability because it can be enriched with the semantics to customize both participating components in the composition. This can result in a configurable architecture model for a PLA. The configuration patterns thus resulted can also be used as reusable artifacts for modeling variability. Capturing the dependency between assets in the form of patterns will guide the evolution process. The dependencies may sometimes be simple usage dependencies or sometimes this may be complex behavioral patterns composed from independent features.

As part of our future work, we plan to address the issue of interaction between patterns in more detail for, we believe that addressing design reuse in the form of patterns and code reuse in the form of components will be a promising step in large scale software development.

References

- [1] B. Keepence, M. Mannion. Using Patterns to Model Variability in Product families. *IEEE Software*, 1999.
- [2] J. Bosch. Product-Line Architectures in Industry: A Case Study. In *Proceedings of the 21 st International Conference on Software Engineering, 1999, Los Angeles, California, United States*, pages 544–554. ACM, 1999.
- [3] R. J. A. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
- [4] D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S.V.G.K. Raju and A. Ananda Rao. An Approach for Pattern Oriented Software Development Based on a Design Handbook. *Annals of Software Engineering*, 10:329–358, 2000.
- [5] D. Schdmit, M. Stal, H. Rohnert, F. Buschmann. *Pattern Oriented Software Architecture: A System of Patterns - Vol II*. John Wiley and Sons, 1999.
- [6] Don Batory, Clay Johnson, Bob Macdonald, dale Von Heeder. Achieving Extensibility Through Product-Lines and Domain Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, April 2002.

- [7] E. Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*, Addison Wesley 1995.
- [8] ESPRIT Working Group 23531, <http://www.dcs.ed.ac.uk/home/stg/fireworks/workshop.html>. *FIREworks, Workshop on Language Constructs for Describing Features*. 2001.
- [9] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlab, M. Stal. *Pattern Oriented Software Architecture: A System of Patterns - Vol - I*. John Wiley and Sons, 1996.
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C V. Lopes, J-M. oingtier, J. Irvin. Aspect-Oriented Programming. In M. M. Askit, editor, *Proceedings of 11th European Conference on Object Oriented Programming (ECOOP '97)*, pages 220–242. Lecture Notes in Computer Science, LNCS 1241, Springer-Verlag, 1997.
- [11] Hans de Bruin, R.S. Cassleman. Scenario Based Generation and evaluation of Software Architectures. In J. Bosch, editor, *Proceedings of the Third Symposium on Generative and Component-Based Software Engineering (GCSE 2001)*, pages 128–139. Lecture Notes in Computer Science, LNCS 2186, Springer-Verlag, September 10-13, 2001.
- [12] I. Jacobson, M. L. Griss and P. Johnson. *Software Reuse Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.
- [13] K Czarnecki, U Eisenecker. *Generative Programming - Methods Tools and Applications*. Addison Wesley, 2000.
- [14] Kang K. S. Cohen Hess J. Nowak W. and Peterson S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report No. CMU/SEI-90-TR-21 Software Engineering Institute Carnegie Mellon University Pittsburgh, PA*, 1990.
- [15] M. Griss, J.Favaro, M. d. Alessandro. Integrating Feature Modeling With RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85. IEEE Computer Society, 1998.
- [16] M. Shaw, D. Garlan. *Software Architecture - Perspectives of an Emerging Discipline*. Prentice Hall, 1996.
- [17] D. McIlroy. Mass Produced Software Components. *Software Engineering: Report on a Conference by the Nato Science Committee*, pages 138–150, October 1968.
- [18] Merriam Webster Dictionary Online. <http://www.m-w.com>.
- [19] D. L. Parnas. On The Criteria to be Used in Decomposing Systems into Modules. *Communications of ACM*, 15(12):1053–1058, 1972.
- [20] Rajasree M S, D Janaki Ram, Jithendra Kumar Reddy. Systematic Approach for Design of Framework for Software Productlines. In *Proceedings of the PLEES'02, International Workshop on Product Engineering: The Early Steps Planning Modeling and Managing, OOPSLA 2002*. Fraunhofer IESE - October 28,2002.

Feature Modeling Notations for System Families

Silva Robak

University of Zielona Gora

Institute of Organization and Management

s.robak@ioz.uz.zgora.pl

Abstract

System families have to address the problems inherent in software artifacts due to the demand for variability, which is necessary to support the needs of different users or to enable functionality in diverse environments and constraints. The possible features of a software product-line may vary according to the needs of particular market segments and purposes. In this paper an overview of describing software variability with means of feature modeling is presented. The role of feature modeling within basic activities in the software development process is shown. Different leading feature modeling notations are compared and basic elements, feature types and relationships between the features are discussed. Further extensions as modeling features with UML and employing fuzzy logic in feature diagrams are also mentioned.

1. Motivation

An understanding of the concepts may be gained by listing their properties, which may be further described with features and dimensions. *Features* portray the qualitative properties of concepts, while dimensions present their quantitative properties. The value range of the *dimension* may be continuous (as integer numbers) or discrete (e.g., given as large, middle, small, etc.). The values of dimensions may be ordered or not (i.e., like the attributes).

A set of dimensions may be used to characterize a number of concepts, each of them represented by a group of values, with one value for each dimension. A similar report is given by listings of feature values for a specific member of the software family in the form of product and feature matrix [1]. The basic role of feature modeling in developing software families as product-lines is depicted in Figure 1.

There is no standard notation for feature diagrams available. The most widely accepted notation is the notation for purposes of the Generative Programming (GP) introduced by Czarnecki and Eisenecker [2].

Section 2 of this paper contains the meaning of the feature notion, feature model and comparison of some

leading notations applied in feature diagrams i.e.: FODA, FORM, FeatuRSEB, GP and Jan Bosch's notation. Section 3 summarizes the feature types and relationships for feature diagrams. The last section contains the conclusion of the work.

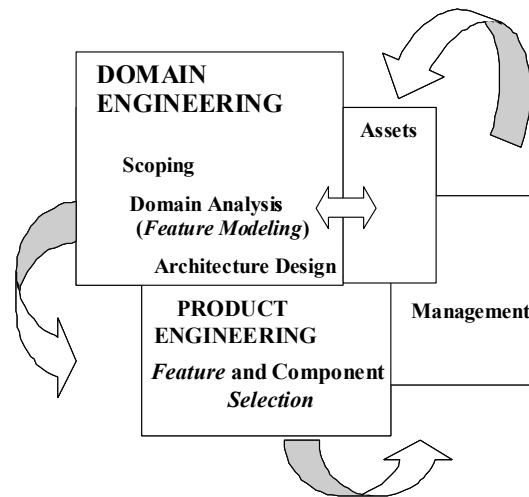


Fig.1 Role of features in basic activities in developing system families [12].

2. Feature modeling

2.1. Feature notion

It is important to remark that the feature notion in this paper (as also in domain modeling) has a somewhat different meaning than the *UML-feature* defined as a property, e.g., an operation or an attribute, that is encapsulated within another entity, such as an interface, a class, or a data type [10].

According to Feature-oriented Domain Analysis (FODA) [7] the feature is the property of a system which directly affects end-users: "Feature: A prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems." Application features have been largely classified according to [7] into: Capabilities (functional, operational, presentation features), Operating Environments, and Domain Technology Implementation Techniques. Functional features are basic services

provided by the applications from the end-user's perspective. Operational features show from the user's perspective their interactions with the applications. Presentation features are related to the way information is presented to the users. Operating Environments are the environments in which the applications are used and operated.

For purposes of Generative Programming (GP) Czarnecki and Eisenecker [2] define the feature notion as "property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances".

FODA definition has been extended in FORM-method [8] - the features are considered within the domain analysis to differentiate a specific application from other related ones and are the "application features" characterizing specific applications from the end-user's perspective and may be classified into the following categories [9]:

1. Capability features:
 - Service,
 - Operation,
 - Nonfunctional characteristics.
2. Domain technology features:
 - Domain method,
 - Standard,
 - Law.
3. Operating environment features:
 - Hardware,
 - Software.
4. Implementation technique features:
 - Design decision,
 - Communication,
 - ADT.

J. Bosch defines feature as a "logical unit of behaviour that is specified by a set of functional and quality requirements" [1].

2.2. FODA feature diagrams

The feature models first introduced in FODA to represent a configuration aspect of reusable software comprise:

- Feature diagram,
- Composition rules (relationships for optional and alternative features),
- Issues and decisions (logical basis of features),
- System feature catalogue.

FODA feature models are the means to describe mandatory, optional, and alternative properties of concepts within a domain. The most significant part of a feature model is the *feature diagram* that forms a tree (graphical AND/OR hierarchical features) and captures

the relationships among features. The root of the tree represents the concept being described and the remaining nodes denote features and their sub-features. A feature is mandatory unless an empty circle is attached to its node, indicating an optional feature. An arc spanning two or more edges of the feature nodes depicts a set of the alternative features (see Table 1). Alternative features in FODA are considered as specializations of a more general feature (i.e. category). The term *alternative feature* indicates that no more than one specialization can be made for a system. The parent node of a feature node is either the concept node or another feature or a sub-feature node, respectively. Direct features of a concept and *sub-features* (i.e., features having other features as their parents) are distinguished. Direct features of a software system may be mandatory, alternative, or optional with respect to all applications within the domain. A sub-feature may be mandatory, alternative, or optional with respect to only the applications, which also enclose its parent feature. If the parent of the feature is not included in the description of the system, its direct and indirect sub-features are unreachable. Reachable *mandatory* features must be always included in every system instance, while an *optional* feature may be included or not, and an *alternative* feature replaces another feature when included. In FODA, selected optional and alternative features are highlighted in the feature diagram for a specific system with the boxes around the name of the selected feature.

External composition rules describe additional dependencies between the features contained in a feature diagram: the features may be described as "*Mutual exclusive With*" or "*Mandatory With*" other features. Supplementary information, such as the explanation of the logical basis of the features (trade-offs, rationales), as well as system feature catalogue (i.e., the register of existing systems and their features), etc., are further parts of the exterior feature definition.

2.3. Feature Diagram Notation in Generative Programming (GP) and other methods

The most popular GP-feature diagram notation differs in some details from the FODA notation and extends it slightly (with Or-features) – see Table 1. Besides a "*one-of-many*"-choice for the strong alternative group, there is an additional kind of feature, so-called Or-features, i.e., an "*n-of-many*"- (nonempty) choice within an alternative or-features group. In the GP-feature diagram notation, each feature name is contained within a box. This makes the presentation of selected options or alternative branches for concrete systems (as in FODA) impossible.

In addition to the features listed above, in Table 1, there are two further feature kinds used in GP [2]: *open*

features and *premature* features. The *open features* are indicated by enclosing the feature name in brackets i.e.: [a Feature]. The open feature is often an arbitrary extendible XOR-Group. The *premature features* are marked with partially dashed lines (at right and bottom) in the feature name-box. Premature features may be used in situations when the feature is premature, i.e., its use in the feature diagram is still not sure and/or the sub-features of premature features will be later modeled in detail. Premature features are used also in situations when the feature in the instance of the notion will be specified in detail later.

The FORM method was developed upon the FODA method by Kang et. al. [8]. In the notation the nodes symbols are the same as in FODA, but all feature names are depicted in boxes - as in GP. There is only strong alternative available, as in FODA (see Table 1). The differentiated feature relationships are composition and generalization/ specialization, as also "Implemented By". The first two are available in other methods - composition explicit, generalization and specialization (also implicit), but the third one is new. "Implemented By" indicates, that a feature is necessary to implement another feature [8].

In FeatuRSEB [4] the UML based notation is introduced for creating feature graphs. The two types of alternative (XOR and OR) are consonant with the enhanced notion of RSEB-variation point. In RSEB [5] the variation point was defined as a point, where a variation may occur. The notion has been further developed in FeatuRSEB and additionally précised in [2]. The binding time (see Section 2.4) maintained in FODA and GP as external description has been integrated in FeatuRSEB- feature model. In FeatuRSEB there are only two possible binding time attributes for variation points: reuse time (XOR) and use time (OR). The overview of modeling features with UML is given in [11].

Jan Bosch uses another feature diagram notation. The notation is as e.g. used in [13], and it is slightly different than in FeatuRSEB (see Table 1). The symbols for the feature nodes are the same as in FODA and feature names are depicted in boxes as in GP. In alternative the empty edges (as in GP) are presented by empty triangles (and filled arcs - by filled triangles) – see Table 1. The external features are introduced as a novel construct. The indication of the binding time is not new, but in comparison to FeatuRSEB more binding times are possible (e.g., compile time).

There are also some further significant extensions to feature diagram notation (not included in Table 1), such as the description of the feature cardinalities, and also arrows indicating the feature associations introduced by Hein et al. [6].

2.4. Information associated with a feature

There is some further descriptive information associated with each concept or feature, such as:

- Semantic descriptions (e.g., models in appropriate formalisms, traceability links),
- Category of feature: concrete, abstract, aspect, etc.
- Rationale (reasons and trade-offs in choosing a feature),
- Stakeholder and client programs (interested in the feature),
- Responsible people/organizations demand (for the feature),
- Exemplar systems (including the feature),
- Constraints and default dependency rules (hard constraints: excludes and requires; weak constraints: default values),
- Binding times, availability sites, binding sites (when, where, by whom the feature is available/may be bound),
- Binding modes (e.g., static, dynamic, reversible),
- Open/closed attribute (extensible/non-extensible alternative features' group),
- Priority (relevance to the project).

It is not required to attach all possible information (which may include much more items than described above) to the portrayed features. Which information should be attached to the feature depends on the concrete domain, the project and the stakeholders.

The optional features as well as the features included within the alternative and strict alternative groups are referred to as the *variable features*. The most relevant information for managing variability purposes is the binding time for the variable features.

2.5. Use of Feature Weights for Variable Features

Each variable feature in a feature diagram may be annotated with a weight symbolizing, e.g., a priority of the variant. There are situations in which it is meaningful to annotate variable features with priorities:

- Domain scoping and domain definition (typicality rates of variable features based on the analysis of known exemplar systems and the target application areas),
- Feature modeling within domain analysis (features' relevance to the project),
- Domain implementation scoping (decisions about which features will be implemented first).

Eisenecker, et al. give an example on the use of priorities for features - not in the feature diagram but within a table containing necessary additional information

Table 1. Comparison of feature diagram notations.

FODA		FORM		GP		Featu-RSEB		J. Bosch		MEANING
	Mandatory		Mandatory f.		Mandatory		Composed of		Composition	Mandatory (if reachable, then f. must be chosen)
	Optional		Optional f.		Optional		Optional f.		Optional f.	Optional (if reachable, may be chosen, or not)
	Alternative		Alternative f.		Alternative		Vp-feature (XOR)		xor-specialization	One-of-many choice from a group
-	-	-	-		Or-features		Vp-f use time bound (OR)		or-specialization	n-of-many choice from a group
-	-	-	-	-	-	-	-		External feature	External Feature
-	-	-	-		Open-feature	-	-	-	-	Open Feature
-	-	-	-		Premature	-	-	-	-	Premature Feature

for each considered feature [3]. The priorities signify features as very important (++) , important (+) , less important (-) and minor (--).

Identical feature sets may be prioritized in different ways for specific customer groups. The principle how some variable features may be described on the basis of fuzzy logic is introduced and discussed in [12].

3. Feature types and relationships

Table 1 contains a comparison of feature types for significant feature modeling methods as:

- FODA,
- FORM,
- Generative Programming (GP),
- FeatuRSEB, and
- Jan Bosch's notation.

In Table 1 the following feature node types are distinguished:

- Mandatory (all methods),
- Optional (all methods),

- Alternative (in all methods),
- OR-Features (GP only); (also FeatuRSEB and Jan Bosch's notation - see below).

In addition, in FeatuRSEB there are two feature types described as "variation points":

- Variation point (OR) and
- Variation point (XOR).

These are not "new" feature types, different than listed above, but are merely describing the fact, that they are parent of the alternative features (XOR) or OR-features (OR). The only difference is that first mentioned - variation point (OR)- is also "use time bound" in FeatuRSEB. The binding time information constrained to this one time (i.e., use time) is not sufficient, because there are many other times, to which the feature may be bound. This kind of information is described in section 2.4 as additional information contained within the feature model.

In Jan Bosch's notation the OR-specialization and XOR- specialization are the same as OR-features (OR-specialization) and the strong alternative (XOR-specialization) respectively. Jan Bosch has introduced a new kind, so called "external feature" (that may also be implemented by external features). This kind of feature

does not fit into the usual classification (listed above), where an alternative, or optional feature may also be an external feature.

The basic relationships between the features are:

- Generalization/ Specialization (all methods)
- Aggregation (Composition) (in all methods)
- “Implemented By” (FORM only)
- Required and Mutex – described in external composition rules.

“Implemented By” (only in FORM) means that a feature is necessary to implement another feature. This kind of information is described as “required “ relationship in FODA and other methods.

4. Conclusion

Specific requirements define the set of conditions or capabilities that must be met by the system or a system component to satisfy a contract, standard or other formally imposed document or description. A conceptual characteristic (e.g., system, component, etc.) is a feature visible to the stakeholder (e.g., users, customers, developers, managers, etc.) and which is used to describe and distinguish system family members. Some features relate to characteristics visible to the end-user, while others relate more to the structure of a system and system capabilities, including also non-functional requirements.

The feature model indicates the intention of the described concept. The set of instances described by a feature model is the extension of the concept. The features are not equal objects - one feature may be part of (different) objects, as also another feature may cover several objects introducing crosscutting aspects into software artifacts.

There is no standard notation for feature diagrams available, the composition rules are maintained separately from the feature diagram. In the paper the most known notations have been compared, and feature types and relationships presented. The presented notations differ only slightly from each other. The basic relationships between the features contained in all methods, such as generalization and specialization, and aggregation (composition), make modeling features with UML-means possible.

5. References

- [1] J. Bosch, *Design and Use of Software Architectures. Adopting and evolving product-line approach.* Addison-Wesley, New York, 2000.
- [2] K. Czarnecki, and U. Eisenecker, *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, New York, 2000.
- [3] U. Eisenecker, M. Selbing, F. Blinn, and K. Czarnecki, *Merkmalsmodellierung für Softwarefamilien.* In *OBJEKTSpektrum* September/Okttober, München 2001, pp. 23-30. – in german language.
- [4] M.L. Griss, J. Favaro, M. D’Alessandro, *Integrating Feature Modeling with the RSEB.* In Proceedings of ICSR98, Victoria, BC, IEEE, June 1998. pp. 36-44.
- [5] I. Jacobson, M.L. Griss, P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley Longman, New York, 1997.
- [6] A. Hein A., M. Schlick and R. Vinga-Martins, *Applying Feature Models in Industrial Settings,* In Proceedings of the First Software Product Line Conference (SPLC1), Denver Colorado August 2000 (P. Donohoe, Ed.). – Massachusetts: The Kluwer International Series in Engineering and Computer Science. pp. 47-70.
- [7] K. Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990. Pennsylvania.
- [8] K.C. Kang et al., *FORM: a feature-oriented reuse method within a domain-specific architectures.* In *Annals of Software Engineering*, V5, 1998. pp. 354-355.
- [9] K. Lee, K.C. Kang, W. Chae, and B.W. Choi, *Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse.* In *Software: Practice and Experience*, Vol. 30, Issue 9, 2000 pp. 1025-1046.
- [10] Object Management Group, *Unified Modeling Language (UML) Specification version 1.3,* June 1999. <http://www.rational.com/uml/resources/documentation/>.
- [11] S. Robak, *Framework for Comparison of Modeling Features in UML.* Feature Modeling Workshop. In conjunction with Generative And Component-Based Software Engineering GCSE’2001, Erfurt 2001.
- [12] S. Robak and A. Pieczynski, *Employing Fuzzy Logic in Feature Diagrams to Model Software Product Lines Variability,* In Proceedings of the 10th IEEE Conference on ECBS 2003, Model-based Development Session and Workshop, Huntsville, Alabama USA – to be printed.
- [13] M. Svahnberg, J. Bosch and J. v Gulp, *On the Notion of Variability in Software Product Lines.* Landelijk Architectuur Congres 2000.

Variability in Multiple-View Models of Software Product Lines

Hassan Gomaa

*Dept. of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444
hgomaa@gmu.edu*

Michael Eonsuk Shin

*Dept. of Computer Science
Texas Tech University
Lubbock, TX 79409-3104
Michael.Shin@coe.ttu.edu*

Abstract

This paper describes how variability is handled in multiple-view models of software product lines, which are depicted using the Unified Modeling Language notation (UML). A multiple-view model for a software product line is an object-oriented domain model which defines the different aspects of a software product line, namely the use case model, static model, collaboration model, statechart model, and feature model, including the commonality and variability. The relationships between the different views are described. The integration of multiple views is achieved by considering relationships among the views in a multiple view meta-model, so that consistency between multiple views is maintained as the multiple-view model evolves. Finally, tool support for the approach is described.

1. Introduction

Several domain engineering methods [1, 2, 8, 11, 12, 13, 14, 17, 19] address modeling commonality and variability in a software product line. Previous papers [5, 6] described how multiple views of software product lines could be modeled using the UML notation [3, 4, 16]. This paper extends the multiple view modeling approach by describing how the different views and variability in those views relate to each other, the underlying meta-model of the multiple views, and tool support for the multiple view modeling approach.

Multiple view modeling of software product lines face greater challenges than single systems, namely how to model commonality and variability. Furthermore, it is important to define how the multiple views relate to each other, for example how variability in one view of the product line relates to variability in a different view. This paper describes the multiple views of software product lines developed using an object-oriented UML based domain modeling method.

A multiple-view model captures different aspects of a software product line, for functional modeling, static modeling, and dynamic modeling. Using the UML notation, the functional view is represented through a use case model in the requirements phase, a static model view

through a class model, and a dynamic model view through a collaboration model and a statechart model. While these views address both single systems and product lines, there is, in addition, a feature model view, which specifically addresses modeling variability in software product lines. In order to explicitly define the relationships among the multiple views, the paper describes the underlying meta-model of the multiple views.

2. Variability in Multiple-View Model Approach of Software Product Lines

A multiple-view model for a software product line defines the different characteristics of a software family [21], including the commonality and variability among the members of the family. A multiple-view model is represented using the UML notation [4] through the use case model, static model, collaboration model, statechart model and feature model views.

For software product lines, it is important to address how variability is modeled in each of the different views. A multiple-view model is modified at specific locations referred to as variation points, which is addressed by

- Variation points in a use case model [11]
- Abstract classes and hot spots [15] in a static model
- Feature modeling [12] and feature dependencies [5] in a feature model

These software reuse concepts are used to deal with variability in the multiple-view model. In addition, alternative decision concepts from single systems are used to model product line variability in collaboration models and statechart models of a software product line:

- Alternative branches and message sequences in a collaboration model [4], which are only enabled if an optional or variant feature is selected.
- Alternative branches and state transitions in a statechart model [10], which are only enabled if an optional or variant feature is selected.

It is important for the multiple views of software product lines to be consistent with each other. In addition, it is essential that as one view is modified at a variation point, the other views are also modified at their variation points so that consistency is maintained.

2.1. Variation Points in Use Case Models

The functional requirements of a system are defined in terms of use cases and actors [11]. An actor is a user type. A use case describes the sequence of interactions between the actor and the system, considered as a black box.

In order to capture the commonality and variability of a software product line, use cases are categorized as a kernel, optional or variant use cases. Kernel use cases are those use cases required by all members of the product line. Optional use cases are those use cases required by some but not all members of the product line. Some use cases may be variant, that is different versions of the use case are required by different members of the product line. The variant use cases are often mutually exclusive [5]. The use case categorization is depicted using the UML stereotype notation, e.g., <<kernel>>.

Variability in a use case model can take place within a use case at variation points [11], which are one or more locations at which a change in a use case could occur. For a use case model, a use case may extend or include another use case at a variation point. The “extend” relationship models a variation of requirements through alternative paths that a base use case might take if appropriate conditions hold. In the extend relationship, the variation point is called an extension point [16]. The “include” relationship models the variation by reusing a common use case used by several other use cases. To model use case variability in product lines, the concept of a feature condition is introduced, which is used to represent an optional feature of the product line. For a given member of the product line, the feature condition is true if the feature is selected.

Fig. 1 depicts variation points of the *Move Part to WorkStation* use case in the factory automation product line [5]. In this example, *Move Part to WorkStation* use case is a kernel use case. *Store and Retrieve Part* is an optional use case, which extends the kernel use case if the feature condition [system has storage] is true. If system has storage (that is, the feature condition is true), *Store and Retrieve Part* extends the *Move Part to WorkStation* use case at extension points -- Store Part and Retrieve Part.

2.2. Alternative Message Sequences in Collaboration Models

In software product lines, once the use cases have been determined and categorized as kernel, optional, or variant, the collaboration diagrams can be developed [5, 6]. Objects in a product line collaboration model can be categorized according to two orthogonal perspectives, the product line perspective and the application perspective such as <<optional, coordinator>>. As with product line use cases, an object can be categorized as a kernel,

optional or variant object to describe commonality and variability of objects in product lines. From an application perspective, an object is categorized depending on the role it plays such as control, algorithm, entity, or interface.

Variability in a collaboration model can be depicted by using an alternative message sequence, which represents a conditional path consisting of objects and messages. A use case can extend a base use case under certain conditions. This change in the use case model requires a change in the corresponding collaboration model through an alternative message sequence. Execution of the alternative sequence is guarded by a feature condition.

Fig. 2 depicts a collaboration model for the *Move Part to Workstation* use case. This diagram shows the system with and without an automated storage and retrieval system. If the system does not have a storage capability, there are four kernel objects involved in the collaboration and one external object. The base message sequence (B1 to B5) depicts the interactions among these objects.

If the system does have a storage capability, corresponding to the *Store and Retrieve Part* optional use case, optional objects *Automated Store and Retrieval System (ASRS) Handler* and *ASRS Forklift Truck* (shaded in gray) are added to the system. The feature condition [system has storage] is used to identify if the system has storage. If the feature condition is true, i.e., the specific product line member supports this feature, then the branch on the collaboration diagram, which is guarded by the feature condition, can be taken.

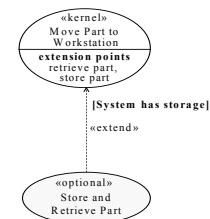


Fig. 1. Variation Points in the *Move Part to WorkStation* use case

2.3. Alternative State Transitions in Statechart Models

A statechart [10] is developed for each state dependent object in the collaboration model, including kernel, optional, and variant objects. Each state dependent object in a collaboration diagram is specified by means of a statechart. Since there can be variants of a control object, each variant is modeled using its own statechart.

Variability in statechart models is represented through an alternative state transition point at which an alternative state transition is triggered if the appropriate condition is satisfied. Each statechart diagram describes each state dependent use case whose corresponding collaboration

diagram contains state dependent control objects. As a state dependent use case evolves, the corresponding statechart model can capture the change in use case behavior by using an alternative state transition.

Fig. 3 depicts a statechart model for *Part Agent with Storage* object in Move Part to Workstation use case, in which alternative state transitions are shaded in gray.

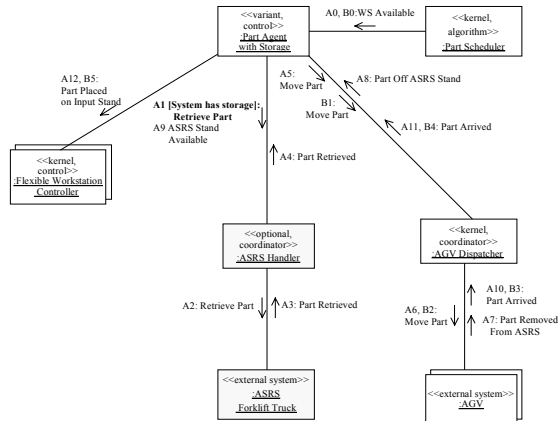


Fig. 2. An alternative message sequence in a collaboration model for move part to workstation use case

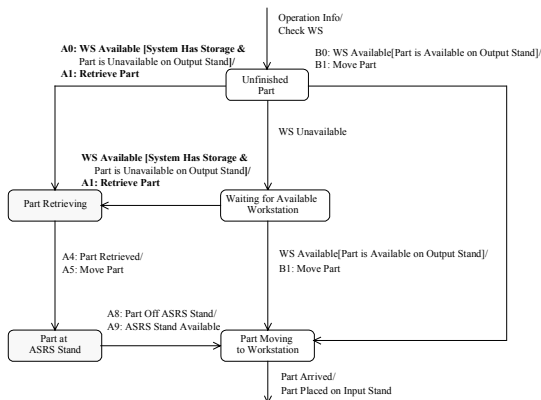


Fig. 3. An alternative state transition in a statechart model for part agent with storage

2.4. Abstract Classes and Hot Spots in Static Models

Variability in class models can be addressed through abstract classes and hot spots [15]. An abstract class is a class with no instance because it declares at least one abstract operation. Each subclass of the same abstract

class can have a different implementation of the abstract operation. A hot spot is a place where class adaptation takes place. Some operations of classes can be implemented or replaced by the subclasses using the inheritance mechanism. Such operations are called hot spots, which provide the flexibility that makes class models capable of evolving.

Fig. 4 depicts a hot spot in *Part Agent* for flexible manufacturing systems. Flexible manufacturing systems can be extended to flexible manufacturing with storage systems. The *Part Agent* class in flexible manufacturing systems provides an operation, `receiveWorkstationStatus` (in `workstationStatus`), for the *Part Agent with Storage* class in flexible manufacturing with storage systems. This operation is a hot spot that is modified in *Part Agent with Storage* class to handle storing a part in the ASRS and retrieving it from storage.

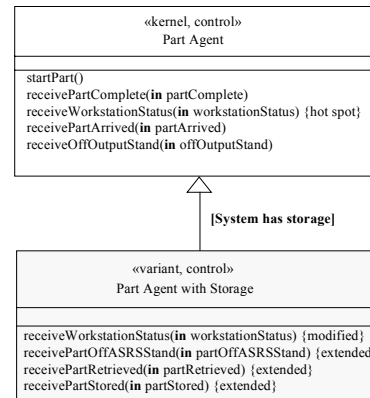


Fig. 4. A hot spot in part agent class

2.5. Feature Dependencies in Feature Models

A feature [12] is an end-user functional requirement, which is used to identify reusable requirements of a software product line [9]. A feature is categorized as a kernel, optional, or variant feature to capture commonality and variability of a software product line.

Variability in feature models [5] can be captured through dependencies among features. Each feature is supported by one or more use cases. In an evolution of the use case model, an alternative use case can extend a base use case under certain conditions. The use case dependency corresponds to a dependency between the features. Each feature is also supported by one or more classes. As a class model evolves at a hot spot, one class can be specialized from another class where the two classes support different features, respectively. This relationship between the two classes corresponds to a dependency between these two features.

Fig. 5 depicts a feature dependency based on a class relationship. The *Flexible Manufacturing* kernel feature is supported by the classes *Part Scheduler*, *Part Agent*,

Flexible Workstation Controller, and *AGV Dispatcher*. The *Store and Retrieve* optional feature is supported by the classes *ASRS Handler* and *Part Agent with Storage*. The feature dependency between the features is reflected in the inheritance dependency between the *Part Agent with Storage* and *Part Agent* classes.

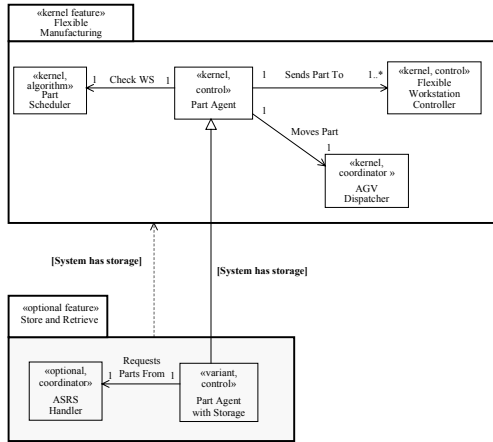


Fig. 5. A feature dependency based on class relationship

3. Multiple-View Meta-Model of Software Product Lines

A multiple-view meta-model describes how a product line view relates semantically to other views, which were informally described in the previous section. A meta-model is a model that defines the meta-classes, their attributes, and relationships for each view of the multiple-view model. A software product line development phase can contain several views, each of which is decomposed into meta-classes. A user-defined multiple-view model is an instance of the meta-model. A multiple-view meta-model is represented using a class diagram in the UML notation.

Fig. 6 depicts the underlying relationships among multiple views in the development phases of a software product line. The views in each phase are:

Requirements Modeling phase:

- Use case model: This model presents the functional requirements of a multiple-view model in terms of actors and use cases.

Analysis Modeling phase:

- Class model: This model addresses the static structural aspects of a multiple-view model through classes.
- Statechart model: This model captures the dynamic aspects of a multiple-view model by describing states and transitions.

- Collaboration model: This model addresses the dynamic aspects of a multiple-view model by describing objects and their message communication.
- Feature model: This model captures the commonality and variability of a software product line by means of features and their dependencies.

Design Modeling phase:

- Consolidated collaboration model: This model synthesizes all the collaboration diagrams developed for the use cases.
- Subsystem architecture model: Based on the consolidated collaboration model, this model addresses the structural relationships between subsystems.
- Task architecture model: This model addresses the subsystems decomposition into tasks (active objects) and passive objects.
- Refined class model: This model addresses the design of classes by determining the operations and attributes of each class.

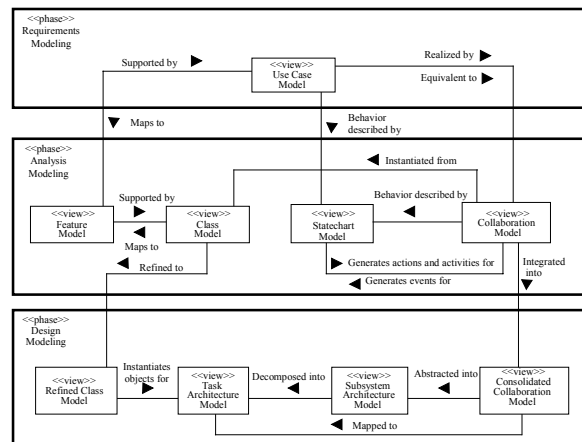


Fig. 6. High-level relationships between multiple views for a software product line

Each of the multiple views has relationships with other views within the same phase, part of which is as follows:

- The behavior of an object in the collaboration model is described by a statechart diagram in the statechart model.
- The statechart model generates actions and activities for the collaboration model.
- The collaboration model generates events for the statechart model.
- A feature in the feature model is supported by classes in the class model. For example, the *Flexible Manufacturing* feature (Fig. 5) is supported by four kernel classes. As the system has storage, the *Storage and Retrieve* feature (Fig. 5) is supported

by the classes *ASRS Handler* and *Part Agent with Storage*.

A view has relationships with another view in different phases, part of which is as follows:

- A use case in the use case model is realized by one or more collaboration diagrams in the collaboration model.
- The state dependent behavior of a use case in the use case model is described by a statechart diagram in the statechart model.
- A feature in the feature model is supported by use cases in the use case model.
- If there is a use case dependency between two use cases that support two different features respectively, the use case dependency between the use cases maps to a feature dependency between the two features.

Each view in Fig.6 is decomposed into meta-classes. The meta-model for the each view and its meta-class attributes are described in [18, 20].

Consistency checking rules are defined based on the relationships among meta-classes in the meta-model. The rules resolve inconsistencies between multiple views in the same phase or in different phases, and define allowable mapping between multiple views in different phases. To maintain consistency in the multiple-view model, rules defined at the meta-level must be observed at the multiple-view model level. Consistency checking is used to determine whether the multiple-view model follows the rules defined in the multiple-view meta-model.

As an example of consistency checking, there is a relationship between Class in the class model and Feature in the feature model (Fig. 6), which is “each optional class in the class model supports only one optional feature in the feature model.” The optional class “Part Agent with Storage” supports the optional feature “Store and Retrieve” in the multiple-view model (Fig. 5) where “Part Agent with Storage” class and “Store and Retrieve” feature are respectively instances of Class and Feature meta-classes in the multiple-view meta-model. For the multiple-view model to remain consistent, this meta-level relationship must be maintained between instances of those meta-classes, that is, “Part Agent with Storage” class and “Store and Retrieve” feature. Consistency checking confirms that each optional class in the class model supports only one optional feature in the feature model.

4. Tool Support for Multiple-View Modeling Approach

In order to support the multiple-view meta-modeling approach, a proof-of-concept prototype, the Product Line UML Based Software Engineering Environment (PLUSEE) has been developed, which built on experience gained in previous research [7, 8]. A domain model addressing the multiple views of a software product line is developed and checked for consistency among the multiple views.

There are two different versions of the PLUSEE prototype, using Rational Rose and Rational Rose RT CASE Tools respectively as the interface to this prototype. Fig. 7 depicts this proof-of-concept prototype. A domain engineer captures a multiple-view domain model consisting of use case, collaboration, class, statechart, and feature models through the Rose tools, which save the model information in a Rose MDL file. From this MDL file, the domain model relations extractor extracts domain relations, which correspond to the meta-classes in the meta-model. Through the domain relations extractor, a multiple-view model maps to domain model relational tables. Using these tables, the consistency checker checks for consistency of the multiple-view model by executing the consistency checking rules described in Section 4. After the domain engineer has produced a consistent multiple-view model, an executable model is developed using Rose Real-Time. The Rose RT executable model is based on message communication between active classes, which execute statecharts.

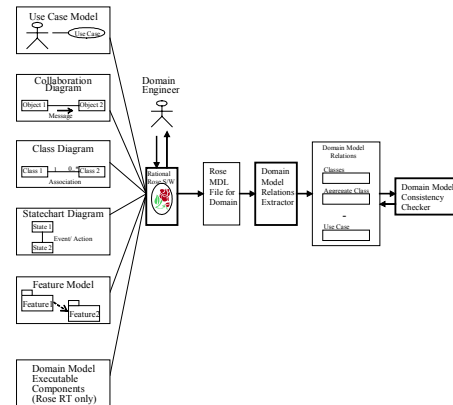


Fig. 7. Product Line UML Based Software Engineering Environment (PLUSEE)

5. Conclusions

This paper has described how variability is handled in a multiple-view modeling approach for software product lines. The approach integrates the multiple views by defining relationships among the different views using a multiple-view meta-model. To support this approach, a proof-of concept tool was developed.

The meta-model depicts life cycle phases, views within each phase, and meta-classes within each view. The relationships between the different views are described. Consistency checking rules are defined based on the relationships among meta-classes in the meta-model.

An important advantage of the multiple-view modeling approach is that it permits the evolution of software product lines by explicitly modeling the variation points in each view where evolution can take place and by defining the relationships between these variation points.

References

- [1] Atkinson, C., Bayer, J., Muthig, D., *Component-Based Product Line Development: The Kobra Approach*, Proceedings, 1st International Software Product Line Conference, 2000.
- [2] Colin Atkinson, Joachim Bayer, and Oliver Laitenberger and Jorg Zettel, "Component-Based Software Engineering: The Kobra Approach," ICSE Software Product Line Workshop, 2000.
- [3] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, Reading MA, 1999.
- [4] H. Gomma, "Designing Concurrent, Distributed, and Real-Time Applications with UML," Addison-Wesley, 2000.
- [5] H. Gomma, "Object Oriented Analysis and Modeling for Families of Systems with the UML," Proc. International Conference on Software Reuse, Vienna, Austria, June 2000.
- [6] H. Gomma, "Modeling Software Product Lines with UML", Proc. Software Product Lines Workshop, International Conference on Software Engineering, Toronto, Canada, May 2001.
- [7] H. Gomma, L. Kerschberg, V. Sugumaran, C. Bosch, and I Tavakoli, "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," *J. Automated Software Engineering*, Vol. 3, Nos. 3/4, August 1996.
- [8] H. Gomma and G.A. Farrukh, "Methods and Tools for the Automated Configuration of Distributed Applications from Reusable Software Architectures and Components", IEE Proceedings – Software, Vol. 146, No. 6, December 1999
- [9] M. Griss, J. Favaro, M. D'Alessandro, "Integrating Feature Modeling with the RSEB", Proc. International Conference on Software Reuse, Victoria, June 1998.
- [10] Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin, March 1996.
- [11] I. Jacobson et al., "Software Reuse: Architecture, Process and Organization for Business Success," Addison Wesley, 1997.
- [12] K. C. Kang et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
- [13] B. Keepence, M. Mannion, "Using Patterns to Model Variability in Product Families", IEEE Software, July 1999.
- [14] Maurizio Morisio, Guilherme H. Travassos, and Michael E. Start, "Extending UML to Support Domain Analysis," Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00), Grenoble, France, 11-15 September, 2000.
- [15] Wolfgang Pree, "Design Patterns for Object-Oriented Software Development," Addison-Wesley, 1995.
- [16] J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," Addison Wesley, Reading MA, 1999.
- [17] Software Engineering Institute, "A Framework for Software Product Line Practice – Version 3," <http://www.sei.cmu.edu/plp/>, Carnegie Mellon University, 2000.
- [18] Hassan Gomma and Michael E. Shin, "Multiple-View Meta-Modeling of Software Product Lines" the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Maryland, December, 2002.
- [19] David M Weiss and Chi Tau Robert Lai, "Software Product-Line Engineering: A Family-Based Software Development Process," Addison Wesley, 1999.
- [20] Michael Eonsuk Shin, "Evolution in Multiple-View Models in Software Product Families," Ph.D. dissertation, George Mason University, Fairfax, VA, 2002.
- [21] Parnas D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

Managing Knowledge about Variability Mechanism in Software Product Families

Kannan Mohan and Balasubramaniam Ramesh
Department of Computer Information Systems
J. Mack Robinson College of Business
Georgia State University
kmohan@cis.gsu.edu, bramesh@gsu.edu

Abstract

Effective design of product platform architecture is a prerequisite for a flexible product family development. In platform designs, managing points of variability is critical to facilitate effective proliferation of product variety. Selecting an appropriate mechanism to incorporate variation points is considered to be crucial in achieving the capability of the product platform to change depending on variability needs. We suggest that the capture of structured knowledge about the selection and use of variability implementation mechanisms to handle different types of variability would play a key role in guiding further variability implementation. We discuss the use of a knowledge management system that can be used to capture variability mechanism related knowledge. Using scenarios from a case study, we illustrate the use of this system in capturing structured knowledge about variability scenarios, specifically describing the choice of the variability implementation mechanism.

1. Introduction

Customers expect products to be specifically tailored to suit to their needs and at no extra cost. Product family engineering facilitates proliferation of variety by exploiting the commonality and variability among the variety of products demanded by the customers in a particular domain [1]. Flexible design and development of product families highly depend on effective variability management. Product variety can be achieved by careful design and development of a flexible product platform and then plugging in different components depending on the customer-specific requirements. Variation points are incorporated into these product platforms so as to delay design decisions to later phases of the development life cycle. These design decisions ultimately depend on variable requirements. Selecting an appropriate mechanism to incorporate such variation points is

considered to be crucial in achieving variety in the product family. Svahnberg et al. [2] warn that many factors affect the introduction of variability into a software product family. Here, we propose the use of a knowledge management system to capture structured knowledge related to variability mechanism selection and implementation. We argue that such a capture would facilitate effective mechanism selection for future variability scenarios.

2. Software Product Families

A product family is a set of products that share certain common aspects and have predicted variabilities. A software product family is defined as a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market segment or mission [3]. Products are clustered as a family based on their commonality, as it is easier to analyze, design and manage the family as a related set of elements rather than concentrating on each member of the family separately [4].

2.1. Variability

Variability refers to the ability of the system to change depending on customer specific requirements. Variation points are specific locations in design artifacts where the behavior of the system can be changed [5]. It is defined using differentiation index, which measures where differentiation occurs within the process flow [6]. Coplien et al. [7] describe variability as an assumption that is true for some elements in a set of objects, or an attribute with different values for at least two different elements from the set of objects. Identification of the points of variability is crucial in proliferating variety from a single service platform. These variations are triggered by customer-specific requirements that could indicate changes in the environment or lead to algorithmic changes in the domain [8]. Bachmann et al. [9] argue that the existence of a collection of alternatives that provide a list

of potential solutions to the development team, is a cause for variability. Feature-based recognition of variability, proposed by Lee et al [8], suggests that variability should be analyzed in terms of product/service features. Since it is highly difficult to elicit all variable requirements completely from the customer, the platform should be designed in such a manner so as to accommodate changes necessitated by evolving requirements. The use of appropriate variability mechanism plays a crucial role in effective application engineering.

3. Implementing Variability in Product Families

Process-oriented issues associated with the design and development of product families play a key role in addressing variability concerns. Cugola et al. [4] argue that following a process that lends itself to the anticipation and identification of all possible family members during the early phases of the development life cycle would be an ideal method for product family development. A wide variety of possible options for finishing the development of a family of products demands the capture of changes that have been planned and those that have been anticipated [9]. Bachmann et al. [9] categorize and describe variability at the architectural level, and prescribe architectural solutions for the various types of variations. Keepence et al. [10] provide a pattern-oriented solution to model variability in product families. They suggest the use of design patterns to model discriminants.

3.1. Variability Realization Mechanisms

Past research has discussed the use of a variety of mechanisms to realize variability. Anastasopoulos and Gacek [11] identify various implementation approaches and discuss the problems and advantages in using these approaches. They enumerate the use of common object-oriented techniques like aggregation and inheritance, and other techniques like dynamic link libraries and conditional compilation, in handling variability. They emphasize that identifying a suitable approach is a critical issue in implementing variability.

Svahnberg et al. [2] have emphasized the importance of the suitability of variability mechanism to a specific variation need. They advance a taxonomy of variability realization techniques. They describe the various factors that need consideration in the process of choosing an appropriate mechanism. Here we propose a model that structures the knowledge related to variability

implementation mechanism. Also, we illustrate the instantiation of this model in our prototype system for specific scenarios from a case study.

4. Knowledge Support for Variability Mechanism Selection

Through a case study, we have developed a model that identifies certain factors that play a critical role in the choice of variability mechanism and how they are related to each other. We argue that when a developer is in the process of selecting a variability mechanism, these factors can be used to guide him/her in documenting the various aspects related to the choice of mechanism. Such documentation can be used to provide with heuristics in selection of mechanisms for future variability scenarios.

The knowledge model that represents the various elements related to variability mechanism selection is shown in Figure 1. Variability, which is the capability of our system to handle changes depending on customer-specific requirements, is driven by the feature model. Such variability is implemented by specific variability realization mechanisms. Use of specific mechanism is constrained by factors like support provided by the programming language to be used and performance.

We have developed a knowledge management system to support the process of variability realization mechanism selection. This system supports capture and use of knowledge structured according to the model shown in Figure 1. In the sections below, we describe the capabilities of the prototype system and illustrate its use in a specific scenario drawn from our case study.

4.1. Illustrating the Use of our KMS

The knowledge management system can be used by designers to document the use of realization mechanisms to implement variability. The underlying model that is used to structure the knowledge capture can be tailored according to project specific needs. Here, we use an example from a case study to illustrate the capture of knowledge related to variability mechanism selection. We have conducted a case study with an organization that is involved in the development of a family of warehouse management systems. Figure 2 shows a specific scenario associated with the selection of a design pattern, viz., the strategy pattern to be used to realize the variability in material handling. Since the model underlying the knowledge capture can be tailored, we can also represent a feature model and link specific features to variability mechanisms used.

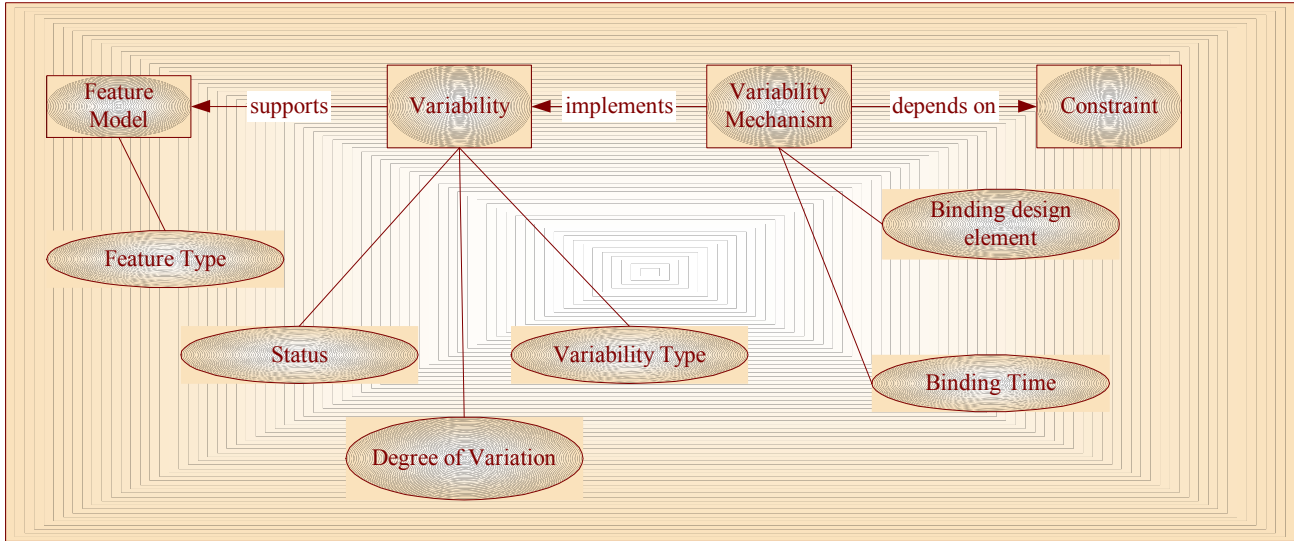


Figure 1: Conceptual Model: Knowledge related to Mechanism Selection

(Oval shape indicates attributes of the rectangles to which the ovals are linked)

The warehouse management system (WMS) has the capability to interact with a material handling system (MHS). Material handling requires equipment to be stored in specific locations in the bays/aisles. Different customers use different methods to handle their equipment. Some customers use a completely automated robotic system that should be commanded from the material-handling module in WMS while some might use a simpler system. The activities required for these different systems and the algorithms used to direct the semi or completely automated equipment handling systems vary considerably. The rest of the material handling system should remain independent of the equipment handling system that handles the location and storage processes. Hence, the designers chose to use the strategy pattern here in order to allow variation of the processes and algorithms independent of the rest of the material handling system. Strategy pattern is used when there is a need for algorithms to vary independent of the clients that use the algorithm [12]. The intent of the strategy pattern is to define a family of algorithms, encapsulate each one and make them interchangeable. A common interface supports algorithms used by specific clients. These algorithms are implemented differently for different clients in Concrete sub-classes. This is a very appropriate pattern to be used here, as different

algorithms might be suitable for different types of equipment handling depending on the customer-specific requirements. Further, new types of material storage handling that a new customer might request can be easily accommodated with this design.

This particular scenario is captured by instantiating the conceptual model shown in Figure 1. Our system is capable of supporting feature models. Specific parts of a feature model can then be linked to particular mechanism use and related justifications. Here, a section of the feature model for the WMS is shown. Alternative material handling systems varying in the levels of automation are shown to trigger the need for variability. Various variability attributes are also captured. For instance, the degree of variation attribute describes that the variability is in the scale of automation, providing the commonly used number of levels. Variability type is shown to be alternative, which means the choice of a particular level of automation could result in some components being replaced by others. The feature under consideration, viz., communication with the material handling system, is described as a mandatory feature, as this is one of the features that identifies the system, i.e., it is very essential for the WMS.

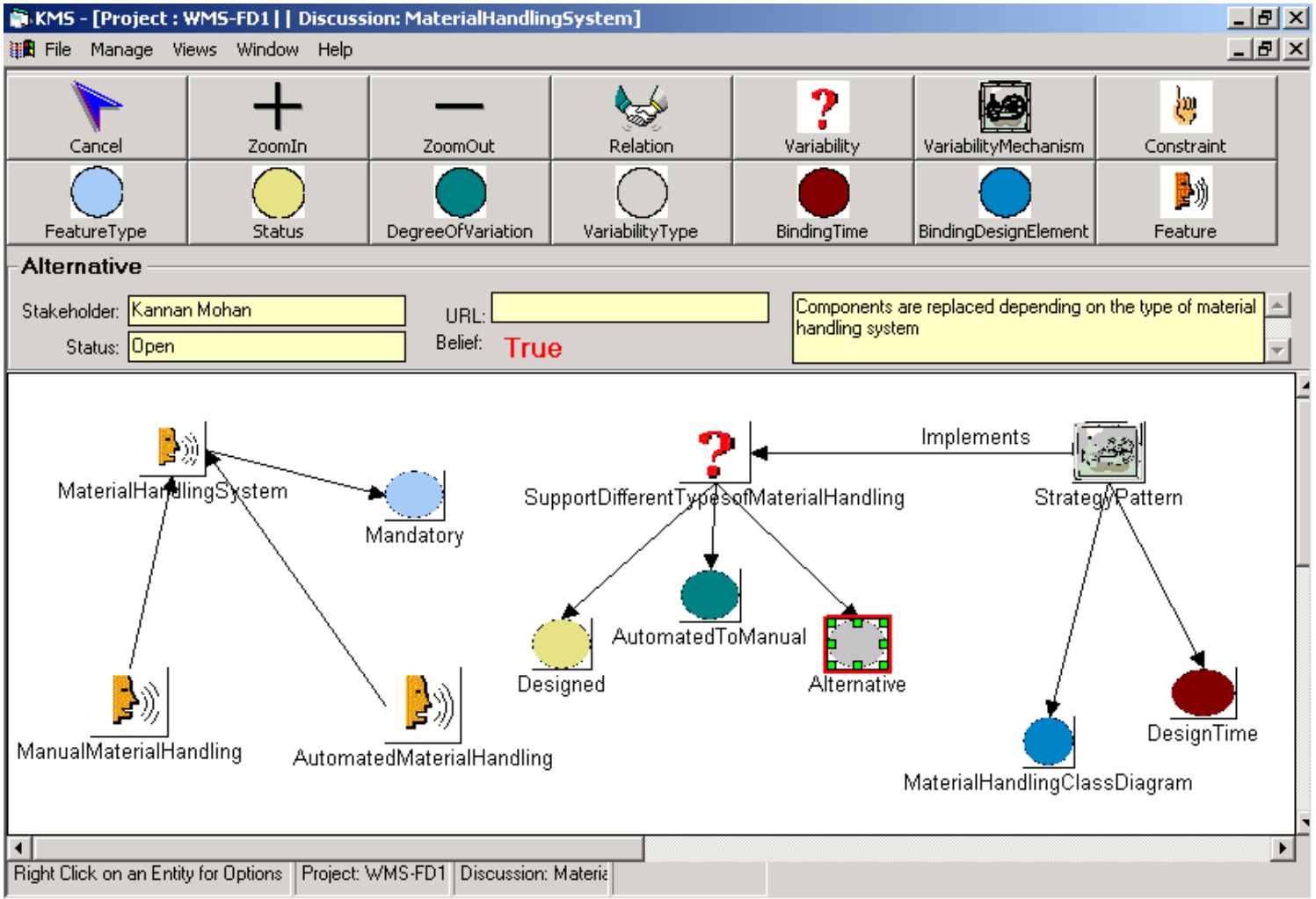


Figure 2: Selecting the Strategy design pattern to handle different types of material handling systems

Our knowledge management system can also be used to represent links between these knowledge elements capture and the design artifacts that are related to these. In the above example, a particular UML class diagram has the strategy pattern implemented to realize the variability in material handling system communication. Our system provides the interface to access the various elements in UML models developed using Rational Rose. Through this interface we can link any knowledge element to any design element from a UML model. We are currently working on extending the prototype system with the capability of using the heuristics gained from such knowledge captures to guide future variability mechanism selection based on the various attribute values of features and variability.

5. Discussion

In this paper, we have discussed the use of a knowledge management system in capturing knowledge

about variability mechanism selection. This prototype can be used to capture various factors that are considered while deciding on a mechanism to implement variability. Such a capture would be valuable in deriving heuristics to guide future mechanism selection. Explicit capture of factors affecting the choice of a variability mechanism would bring forth any constraints or conflicts in mechanism selection. Close integration of our prototype system with work process and productivity tools like Rational Rose, Microsoft office suite, and Groove, a communication/collaboration tool tend to reduce the overhead in switching from their work environment to capture knowledge and link it to appropriate design artifacts.

6. References

- [1] J. Pine, *Mass Customization: The new Frontier in Business Competition*. Boston, MA: Harvard Business School Press., 1993.

- [2] M. Svahnberg, J. v. Gulp, and J. Bosch, "A Taxonomy of Variability Realization Techniques," Blekinge Institute of Technology, Sweden 1103-1581, 2002.
- [3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Upper Saddle River, NJ: Addison-Wesley, 2002.
- [4] G. Cugola and C. Ghezzi, "Program Families: Some Requirements Issues for the Process Languages," presented at 10th International Software Process Workshop, Dijon, France, 1996.
- [5] J. Bosch, *Design and Use of Software Architectures*: Addison-Wesley, 2000.
- [6] M. Martin, W. Hausman, and K. Ishii, "Design for Variety," in *Product Variety Management*, T.-H. Ho and C. S. Tang, Eds. Norwell, MA: Kluwer Academic Publishers, 1998.
- [7] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, vol. 15, pp. 37-45, November/December 1998.
- [8] K. Lee, K. C. Kang, E. Koh, W. Chae, B. Kim, and B. W. Choi, "Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice," in *Software Product Lines: Experience and Research Directions*, P. Donohue, Ed. Norwell, MA: Kluwer Academic Publishers, 2000.
- [9] F. Bachmann and L. Bass, "Managing Variability in Software Architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 126-132, 2001.
- [10] B. Keepence and M. Mannion, "Using Patterns to Model Variability in Product Families," *IEEE Software*, vol. 16, pp. 102-108, 1999.
- [11] M. Anastasopoulos and C. Gacek, "Implementing Product Line Variabilities," presented at Symposium on Software Reusability, Toronto, Ontario, Canada, 2001.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

Towards a component-based, model-driven process supporting variability of real-time software

Vieri Del Bianco Luigi Lavazza
CEFRIEL - Politecnico di Milano
P.zza Leonardo Da Vinci, 32
20133 Milano, Italy
{delbianc|lavazza}@elet.polimi.it

Abstract

Model-driven techniques for the development of component-based real-time software are available. These techniques have also been integrated with formal methods, thus providing the developer with the degree of confidence that is needed when dealing with real-time, safety-critical applications.

Here we report an initial discussion concerning the applicability of such techniques to the management of variability.

1. Introduction

Real-time software is often safety-critical. This suggests that formal methods are used (especially in the specification phase) in order to guarantee that the development has a sound base. Another consequence of the criticality of real-time software is that the development process tends to be quite expensive (for instance, due to the need for extensive testing). It is therefore desirable to be able to reuse as much as possible the code already developed and tested. For this purpose component based development is a promising technique.

A problem –which occurs particularly often for real-time embedded software– is that the same software is used in different contexts, because the user needs evolve in time, but also because different users have different needs at the same time. In both these cases, in order to be able to manage the variations in an effective and economic way we tend to modularize the system in order to have a stable core and some additional components that can vary from version to version of the system.

Here we discuss how to adapt the techniques that were developed for the development of non varying (or slowly varying) software to the development and evolution of software which is subject to variability.

Best practices in software development call for precise modeling of user needs and consequently for rigorous

specifications of the system to be developed. In order to verify and validate specifications, several methods have been proposed, including model checking. This techniques requires that specifications are written in a suitable formal language. Since formal languages are not very easy to use, in past years we developed a technique that allows the user to write the specifications of the system as a UML model, and then translates this model into a formal notation, thus enabling the application of formal methods. For this purpose we defined an extension of UML, called UML+ [2][5][6]. In order to make the specifications more modular (hence easier to manage), to allow an incremental approach to specification verification, and to facilitate the transition to the implementation phase, UML+ models can be structured in terms of components (or capsules [7]). A component-oriented version of UML+ is also available [8].

A possible process that exploits the UML+ notation and tools is depicted in Figure 1. The idea is that the specifications are written in UML+ and then they are verified by means of a set of tools that can be applied to models derived from the UML specifications. It is interesting to note that the verification of specifications requires that the environment where the system operates must be modeled as well. This can also be done by means of UML+. In Figure 1 we stress the need for verifying the specifications in several different ways (including simulation). This need descends from the nature of real-time software, which is often safety-critical. However it is well-known that errors in the specifications are likely to be very expensive, thus the possibility of verifying UML+ models contributes also to keep the cost of development under control.

2. Supporting variability

The UML+ notation and process were originally conceived to support development of real-time software without considering variability issues. Therefore it is

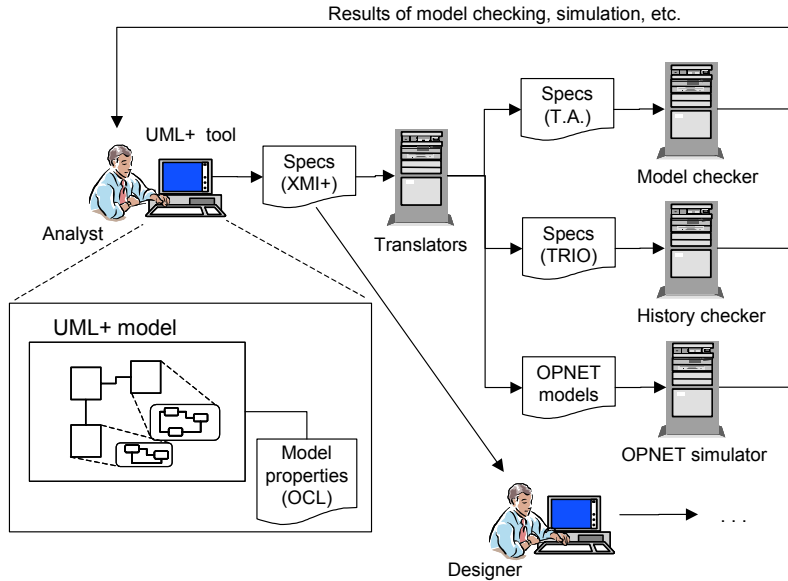


Figure 1. The specification process exploiting UML+.

interesting to discuss the applicability of UML+ in managing software variations.

We consider first the notation, and then the process in its two main phases: specification and design/implementation.

2.1. Notation

The first question we have to answer concerns the notation: is UML+ adequate to support variability? In order to answer this question we observe that with our notation the component (or capsule) is the elementary unit of the model. Thus most variations can be represented in terms of components. In other words, it is possible to define a taxonomy of variations where differences between two versions of a system are described in terms of components. For instance, here is a preliminary list of possible variations:

- Component A is replaced by component A', having the same interface. In general this means that A and A' behave differently under some respect.
- A component B is removed from the model. This typically means that the components that were previously connected with B change their behavior accordingly.
- Two components that are connected together change the way they communicate. This generally means that they also change their behavior.
- ...

A taxonomy of variations is needed in order to make the management of variations easier. E.g., it will be possible to identify variations patterns, and thus to define suitable management criteria for each pattern.

In general the UML+ notation proved to be flexible enough to represent all the variations we spotted so far.

2.2. Process: specification phase

When dealing with variations the analyst will generally build a new specification on the basis of:

- New requirements.
- Existing specifications (i.e., compositions of components), generally associated with a set of proved properties.
- Fragments of specifications, possibly single components or small compositions of components, generally associated with a set of proved properties. These are particularly interesting, because they can help in building specifications incrementally. In fact, when using such a fragments in a bigger composition it is generally not necessary to include a detailed description of every single component belonging to the fragment: often an abstract description of the fragment (featuring the properties that have been proven) is sufficient, and makes the resulting specification smaller and easier to verify.

In any case, it is likely that the resulting specification will have large parts in common with the existing specification, but in general it will feature some new components, some modification in the system structure or in the communications among components, etc.

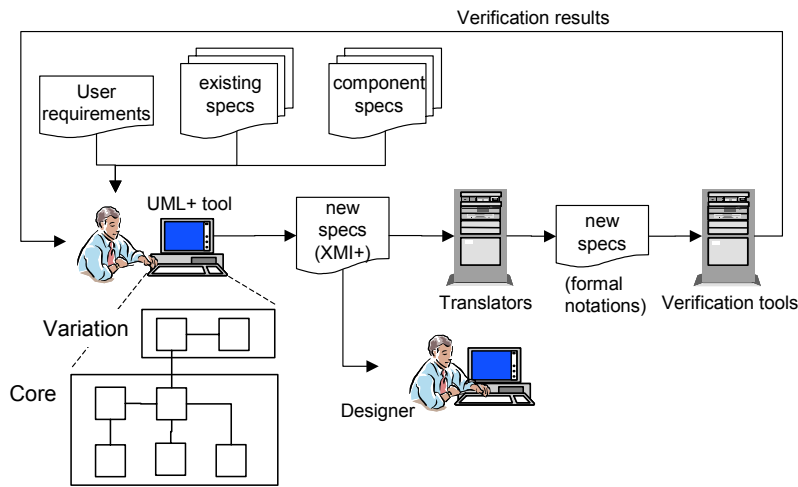


Figure 2. The specification variation process exploiting UML+.

In general the new specification can be verified against user requirements by means of the set of formal techniques mentioned in the previous section. The specification process is synthetically described in Figure 2. Note that in Figure 2 it is explicitly represented that there is a part of the model (the core) which is common to several systems and a part which is specific of the considered system.

- The abstract description of the common part contributes to avoid the state explosion problem, which affects some of the verification techniques employed, namely model checking.

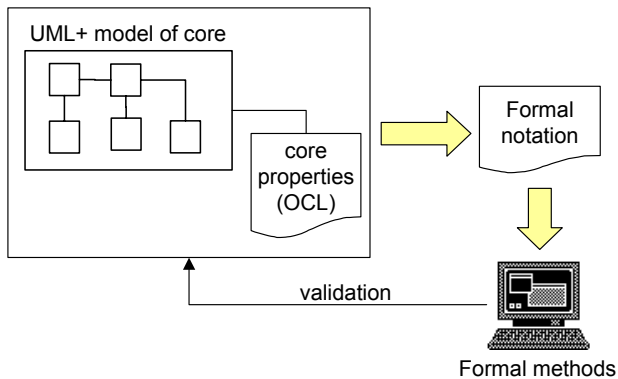


Figure 3. Verification of the properties of the core.

As already mentioned, it is interesting to note that the verification approach can be incremental. In fact the core can be thoroughly verified (as depicted in Figure 3), and a more abstract version of it can then be used to specify the variations (as depicted in Figure 4). Of course, the larger the core (that is, the smaller the variations) the more efficient can be the verification. This approach is effective under several respects, especially:

- The core of the system is explored in depth, thus increasing the probability of finding possible errors in the more critical part of the system.

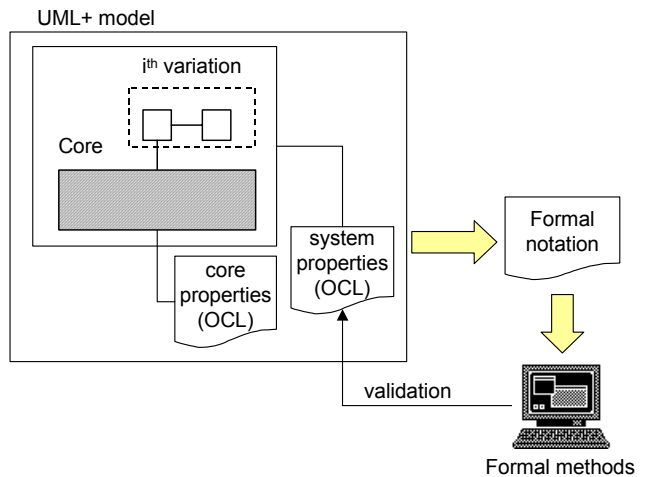


Figure 4. Verification of the properties of a system exploiting the knowledge of the properties of the core (which are represented in an abstract way).

In order to support the process described above we do not need specific notations or tools: UML+ and the associated tools can be used also in the incremental development process. Nevertheless, in order to make the process efficient, some support to process management is needed. In particular, we need a configuration management system that is able to treat the variability concepts. It is also important to be able to manage

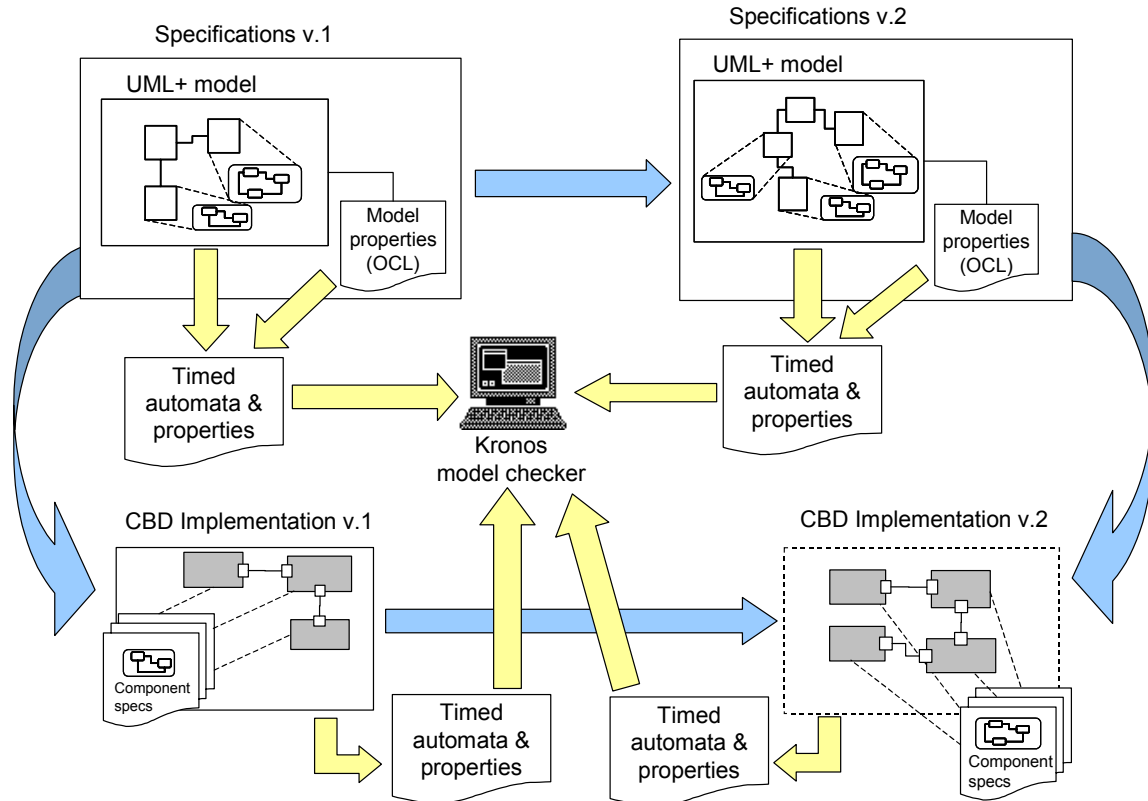


Figure 3. Variability and verifiability.

traceability relations between requirements, specification fragments, and components. Examples of such relations are: dependencies among components, usage relations, property associations, etc. The support system should integrate CM and traceability management, and should be able to treat the information at different granularity levels. For instance a single requirement could be associated with an aggregation of components, or vice versa a set of related requirements could rely on a single component.

2.3. Process: design and implementation phases

In the implementation phase it is crucial to provide the developers with a tool that verifies the consistency of the implementation with its formal specifications.

The main problem here is that the syntax and semantics of the design/implementation models (asynchronous models) usually differ from the specification models (synchronous models). In order to guarantee that the implementation preserves the specifications' properties we have to formalize the mapping between the notations used in the two phases. This formalization is in progress.

Components simplify the problem. A component, in our approach, is described by a model (containing Timed Statecharts [9] in specifications, and standard UML [1] statecharts in implementation) and the requirements the

component must satisfy (typically expressed in some sort of logic language). The latter are often referred to as "abstract specifications".

Model checkers verify that the specifications model satisfies the specifications requirements. The idea is to extract the implementation requirements from the specification model: if it is possible to verify that the actual implementation model satisfies the implementation requirements, as a consequence it is proved that the system verifies the specifications requirements. For bigger projects these proofs can be repeated for every design step or refinement.

Variability increases the complexity of the process. In fact we have to consider refinements at different levels. The component approach helps to limit the scope of the verifications. For instance, if we modify the model of a component while maintaining the requirements unchanged, the rest of the systems will not be affected: we will have to consider only the implementation refinements dependent on the change (down propagation). On the contrary, if we modify the abstract specification of a component, we will have to verify that every component or system using the changed component still satisfies its requirements: if so the change will not propagate further, otherwise we can iterate the process (up propagation).

Figure 5 describes the complete process (with only two levels of refinement: specifications and implementation).

It is quite clear that the transition between two levels of representation is a delicate step. We are currently exploring different ways to maintain the needed properties in the passage:

- Black box approach. The model of a component specifies it completely. We are able to verify properties on a model. We would like to (automatically) verify if a model is a refinement of a more general one. Bisimulation theories should help us to verify whether a model is a refinement of another (or just the same model).
- Model-properties approach. A model can be completely characterized by the properties it verifies. If it is possible to generate the characterizing properties from a model, it would be possible to check the refined model with the properties of the previous one.
- Guided refinement/change approach. We could demonstrate and verify that under specific refinement operations the properties are maintained. We could also demonstrate and verify that under specific change operations some properties are maintained, and other are modified in a predictable way; thus we won't need to verify the changed component against all the properties, but only on the properties we can't predict.

These approaches are not mutually exclusive; on the contrary, together they can successfully deal with different aspects of the process. In the analysis phases formal bisimulation is the preferred solution. On the contrary in the passage from specification to implementation the guided refinement seems the only plausible approach, since it appears to be extremely difficult to apply bisimulation to models expressed in different formalisms. Moreover, the design and programming languages are usually more expressive than those used in analysis, and are not exhaustively verifiable; in other words, no model checkers are available, the properties cannot be verified, but only proved manually.

3. Conclusions

The development of real-time software can profitably exploit well-established techniques like model-based (more precisely, UML-based development), formal methods, component-based development.

The management of variability in real-time software can also benefit of the mentioned techniques. Nevertheless, several issues need to be explored.

In this paper we sketched our approach to this problem. Discussion is open, and the research agenda for the problems mentioned throughout the paper is still in a draft state.

References

- [1] OMG:OMG *Unified Modeling Language Specification* Version 1.4. September 2001.
- [2] Lavazza, L., Quaroni, G., Venturelli, M.: Combining UML and formal notations for modelling real-time systems. In: Gruhn, V. (ed.): *Proceedings of ESEC/FSE 2001*, Vienna, Austria, September 10-14, 2001. ACM Press, New York (2001).
- [3] Ghezzi C., Mandrioli D., Morzenti A.: TRIO, a logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, Vol.12, n.2. Elsevier Science (May 1990).
- [4] Yovine, S.: Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, Vol.1, N.1/2. Springer, October 1997.
- [5] Del Bianco, V., Lavazza, L., Mauri, M.: An introduction to the DESS approach to the specification of real-time software. *Technical Report RT01002. CEFRIEL (2001)*. Available from <http://www.cefril.it>, "WISE" section.
- [6] Del Bianco, V., Lavazza, L., Mauri, M.: A Formalization of UML Statecharts for Real-Time Software Modeling. *The Sixth Biennial World Conference On Integrated Design Process Technology (IDPT 2002)*. Pasadena, California, June 23-28, 2002.
- [7] Selic B., Gullekson G., Ward P.T., *Real-Time Object-Oriented Modeling*, Wiley, 1999
- [8] Del Bianco, V., Lavazza, L., Mauri, M.: "Towards UML-based formal specifications of component-based real-time software", *Fundamental Approaches to Software Engineering (FASE03), ETAPS European Joint Conferences on Theory And Practice of Software*, April 2003, Warsaw, Poland.
- [9] Y. Kesten, A. Pnueli, Timed and Hybrid Statecharts and their Textual Representation, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium, 1992*.