M.E. Anilloy Frank

A new Approach to the Identification of Variability in Model-based Embedded Software

Dissertation

vorgelegt an der

Technischen Universität Graz

zur Erlangung des akademischen Grades
„Doktor der Technischen Wissenschaften“
(Dr. techn.)

durchgeführt am Institut für Technische Informatik
Vorstand: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Eugen Brenner

November 2012

“It is a thousand times harder to make simple things than complicated ones”

-Mikhail Kalashnikov

Abstract

Modern development of applications and products largely use Model Based Software Engineering (MBSE) approaches. Large development environments with numerous groups from several domains tend to repeatability in designing which necessitates a usually large number of variants of components in different products and versions. Especially the automotive industry is one that can be characterized by numerous product variants, often driven by embedded software. With the ever increasing complexity of embedded software, the electrical/electronic models in automotive applications are getting enormously large and thus unmanageable.

Software models are hugely hierarchical in nature with numerous composite components deeply embedded within projects comprising typically of Simulink models, implementations in legacy C, and numerous other formats. Hence it is often necessary to define a mechanism to identify reusable components embedded deep within this hierarchy.

The approach presented in this work is model-based middleware architecture, relying on a platform independent definition of selectively targeting the component-feature CF meta-model instead of a global search to improve the identification. We explore the components and their features artifact in a model from an explicit definition of the component node list and the feature node vector respectively.

The proposed approach has been evaluated on an industry use case – a project model developed using the design tool ESCAPE built by the company Gigatronics. ESCAPE is an authoring tool for graphical definition, manipulation, and analysis of functional networks with the ability to manage a wide range of configurations.

Since the approach does not depend on the depth of the hierarchy of the components or on its order, it serves well with all the scenarios, thereby exhibiting a generic nature.

Kurzfassung

Die Entwicklung der Anwendungen und Produkte folgt heute weitgehend den Model Based Software Engineering (MBSE) Ansätzen. Erforderliche Varianten von Komponenten in Produkten, deren Entwicklungsumgebungen aus zahlreichen Gruppen mehrerer Domänen bestehen neigen dazu, sich bei der Gestaltung häufig zu wiederholen. Die Automobilindustrie ist eine solche, die durch zahlreiche Produktvarianten, die oft durch Embedded-Software angetrieben werden, charakterisiert werden kann. Mit zunehmender Komplexität von Embedded Software werden die elektrischen / elektronischen Modelle in Automotive-Anwendungen enorm groß und dadurch praktisch unübersichtlich.

Moderne Modelle sind immer ihrer Natur nach hierarchisch mit zahlreichen Composite-Bauteilen, tief in die Projekte eingebetteten Simulink-Modellen, häufig gemischten Implementierungen in Legacy-C und Matlab, sowie zahlreichen anderen Formaten. Daher ist es oft notwendig, einen Mechanismus zu definieren, um wiederverwendbare Komponenten in dieser tief eingebetteten Struktur zu identifizieren.

Der Ansatz, der in dieser Arbeit vorgestellt wird, ist eine Modell-basierte Middleware-Architektur, die sich auf eine plattformunabhängige Definition des Component-Feature Meta-Modells abstützt, statt auf eine globale Suche, was die Identifizierung deutlich verbeßert. Wir untersuchen die Komponenten und deren Eigenschaften in einem Modell, das aus einer expliziten Definition der Komponenten-Knoten-Liste und des Merkmal-Knoten-Vektors erstellt wird.

Der vorgeschlagene Ansatz wurde auf einem Industrie-Use-Case-Projekts evaluiert, deßen Modell mit dem Design-Tool ESCAPE von der Firma Gigatronics erstellt wurde. ESCAPE ein Authoring-Tool für grafische Definition, Manipulation und Analyse funktioneller Netzwerke mit der Fähigkeit, eine breite Palette von Konfigurationen zu verwalten.

Da der Ansatz weder auf die Tiefe der Hierarchie noch auf der Anordnung der Komponenten beruht, hängt die Anwendungsmöglichkeit nicht vom Szenario ab und zeigt damit einen generischen Charakter.

Extended Abstract

Motivation

More and more platform specific embedded systems use Component Based Software Engineering (CBSE) approaches for the development and deployment of applications on specified target platforms. Increasing numbers of target platforms and new distributed business processes require engineering approaches to specify an application in a platform independent way, while supporting the deployment and dynamic configuration of these application artifacts.

The automotive industry is one that can be characterized by numerous product variants, often driven by embedded software. With the ever increasing complexity of embedded software, the electrical/electronic models in automotive applications are getting enormously unmanageable.

Considering constantly changing requirements within the set of products, the variability needs to evolve. Many embedded systems are implemented including a set of alternative function variants to adapt to the changing requirements. Major challenges are in identifying the commonality of functionality, where the designs involve variability (ability to customize). In addition to variants, versions/releases of functional blocks also play an important role for the effective management over the entire product cycle.

The proposed architecture in this work can be applied to applications built of components, which contain a formal specification of their spatial description (user interface), function (behavior), and naming (user data). Because this configuration is based on the meta-data model used by these software components, this approach exhibits feasibility in migration between different platforms.

Related Work

Recent trends in development of automotive applications and products largely use Model Based Software Engineering (MBSE), an industrially accepted approach. Model Driven Software Development (MDSD) is typically realized in distributed environments. While MDSD facilitates models for the abstract specification of system architectures, their platform specific artifacts are often realized by applying Component Based Software Engineering (CBSE) techniques.

The basic element in these approaches is a software component, which is an execution unit with well defined interfaces. Usage of software components is driven by the requirements of improving reusability of developed software artifacts. The mapping of software components on networked Electronic Control Unit (ECU) is a distinct shift from CBSE. Software components are combined with the help of assembly descriptions. They are specified in the development phase, and are resolved in the deployment phase of a CBSE process.

Most MDSD approaches follow the Model Driven Architecture (MDA) concept. In this concept, beginning with the specification of a platform independent model, this is then transformed to a platform specific model by applying several generators. The layered Meta Object Facility (MOF) approach is used for creating these models. This approach is also used as basis for the Unified Modeling Language.

Distributed systems are based on Service Oriented Architectures (SOA) and Grid Computing. SOAs built on well defined interfaces deal with business processes, distributed over existing and new heterogeneous systems, often from different vendors. While this aspect is also targeted in CBSE, the loosely coupled SOA services are contradictory to the assemblies used in CBSE. Grid Computing has evolved for the distribution of scientific computational tasks, but is also used for distributed data access in heterogeneous networks.

Model-based software components in automotive domain

Here a model-based approach for the distributed business process, with several participants realizing different functionalities in a multi-layered (or multi-tiered) architecture is introduced. The approach addresses the identification process in the development and deployment of components used in the realization of such distributed processes.

Model-based techniques are used to support the usage of platform independent code. The abstract specification of the components is done by domain experts, and the task for deploying these components on different platforms is handled separately by specific platform developers. As a consequence the effort required for porting elements is reduced.

To enable the identification of variability for software components in a distributed system within the automotive domain, we enlist the specifications below:

- *Specification of components by compatibility*

  The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks, whilst using later version of the function may fail to perform as expected.

- *Extract features, identify and specify*

  To enable parallel development, it is necessary to be able to extract features, and to identify and specify the functional blocks in the repository based on architecture and functionality.

- *Usability and prevents inconsistencies*

  A process that tracks usability and prevents inconsistencies due to deprecate variants and version from repository is required.

- *Testing mechanism for validations*

  A testing mechanism for validations in order to maintain high quality for components and its variants has to be established.

- *Mechanism for simplified assistance*

  The developer has to be assisted by a process to intelligently determine whether a functional block or its variant should exist in the data backbone, to avoid redesigning of existing functions, thereby improving productivity.

Implementation
A system to achieve the specification for model-based software components as described in the previous section has been implemented, targeted for modeling automotive E/E systems and networked embedded devices. Sharing of software resources across networked ECU's and the execution of third party components is eased by this approach. Tools for deploying the runtime and for development of the components are described, and examples of their usage are discussed.

Case study
The proposed approach has been evaluated on an industry use case, a project model developed using the design tool ESCAPE built by the company Gigatronics; ESCAPE is an authoring tool for graphical definition, manipulation, and analysis of functional networks with the ability to manage a wide range of configurations. It provides diagnostic tools like dependency analysis, fault back tracking, and support for mathlab/simulink simulation models and various programming language implementations. Another application was to apply our proposed solution to scattered projects that conform to AUTOSAR naming conventions.

Both case studies outline the benefit of using multiple models for the functional specification resulting in a reduction of platform specific code. The results also demonstrate the importance of good tooling support for creating the functional models and identify the performance of the model interpreters as a key aspect required for the application of this approach in industrial projects.

STATUTORY DECLARATION

Acknowledgments

This thesis was written at the Institute for Technical Informatics, Graz University of Technology.

For his encouragement and scientific support I would like to thank my supervisor Eugen Brenner, Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. for his continuous support in my research and valuable discussions on our research project

I also express special thanks to Christian Kreiner, Dipl.-Ing. Dr.techn, who has provided valuable insight and worth-while experiences.

I extend thanks to Christian Steger, Ass.Prof. Dipl.-Ing. Dr.techn. for his support.

I also thank Dr. Ramananda H.S, Asst. Prof. and HOD, SJEC, Mangalore, for his expertise in helping me develop the mathematical model.

Additionally I would like to thank my colleagues at the Institute of Technical Informatics.

Special thanks are due for Mr Pramod and Priyanka who has been a constant moral support.

Finally I am extraordinarily thankful to my wife Vianny and children for their patience and continuous support throughout the years of my studies.

Graz, 2012 Anilloy Frank

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are microcontroller-based systems built into technical equipment mainly designed for a dedicated purpose. Communication with the outside world occurs via sensors and actuators[1]. Although this definition implies that embedded systems are used as isolated units, there is also a trend to construct distributed pervasive systems by connecting several embedded devices as noted by Tanenbaum and van Steen[2].

The current development trend in automotive software is to map software components on networked Electronic control units (ECU), which includes the shift from an ECU based approach to a function based approach. Also according to data presented by Ebert and Jones[3] up to 70 electronic units are used in a car containing embedded software, which is responsible for the value creation of the car and consists of more than 100 million lines of object code.

Reuse of automotive embedded software is difficult, as it is developed for a small ECU that lacks both processing speed and memory of a general purpose machine. Moreover, the complexity of the algorithms is dramatically increasing. In view of this complexity, achieving the required reliability and performance is one of the most challenging problems[4].

Ebert and Jones presents recent data about embedded software, stating that the volume of embedded software is increasing between 10 and 20 percent per year as a consequence of the increasing automation of devices and their application in real world scenarios.

Complexity management has become a vital factor in an organization. To save costs a company needs to minimize internal complexities, where as it is necessary to satisfy the range of customer requirements which determines external complexity. The dynamics involved is due to three major factors

- *Globalization:* For companies to be present in all major markets and to be competitive the requirements of customers with different cultural, technological, economic, and legal backgrounds needs to be incorporated in products.

[1] [Ebert and Salecker, 2009, p.14] ,
[2] [Tanenbaum and van Steen, 2006] ,
[3] [Ebert and Jones, 2009] ,
[4] [Kum et al., 2008] ,

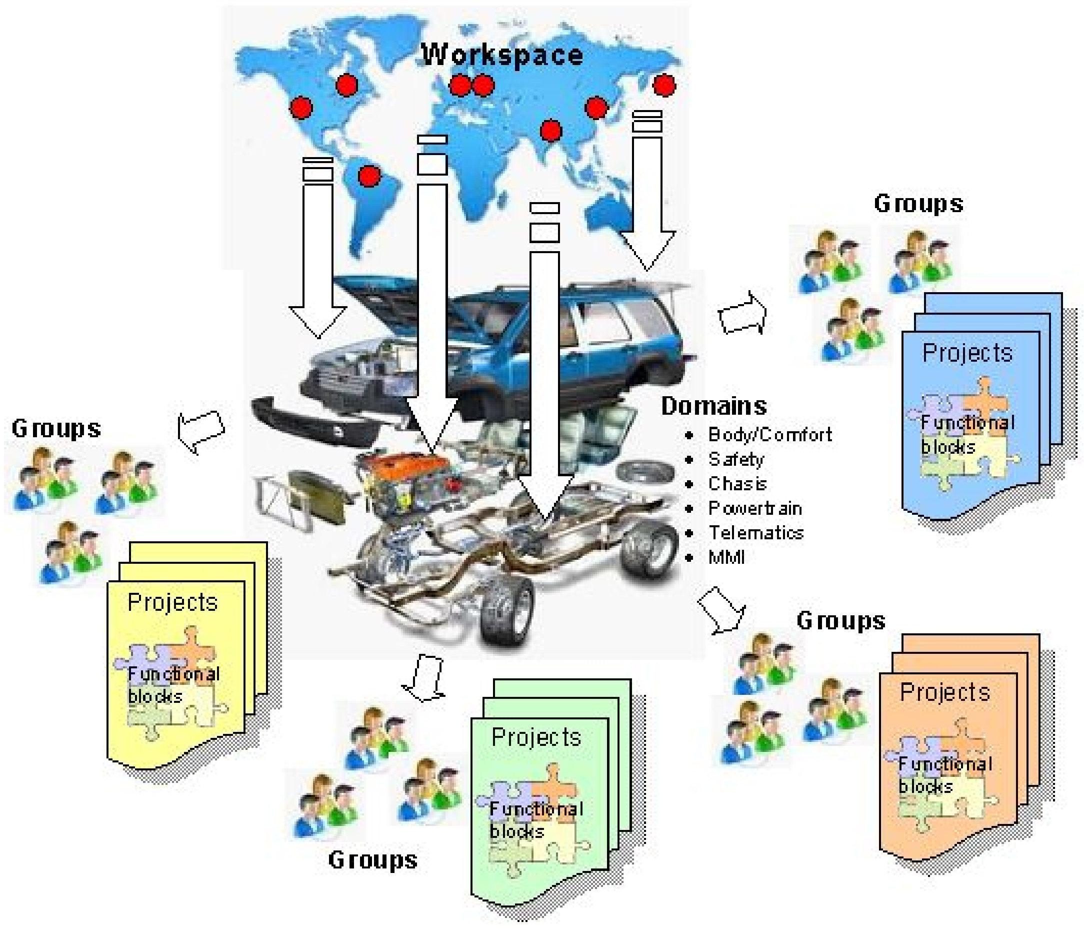


Figure 1.1: Distributed system in automotive domain

- *Evolving Technology:* With a need to reduce the time-to-market, technology is evolving at an extremely fast pace. The trend to launch new products quickly in the market is increasing, which necessitate for enhanced technology as well as convergence of technologies[5].

- *Increasing market influence:* The customers influence to determine a products features and price is inducing the manufacturers to provide more and more product variants.

Figure 1.1 depicts the automotive domain where the design and development of the application is situated at different locations across the globe. Numerous groups working on projects in various domains (e.g. body and comfort, chassis, powertrains, safety, etc.) are distributed at these locations. Many of these groups work on closely related projects where a tendency of repeatability of functional blocks is apparent.

[5] [Bayus, 1994, p.306] , [Poole and Simon, 1997, p.240]

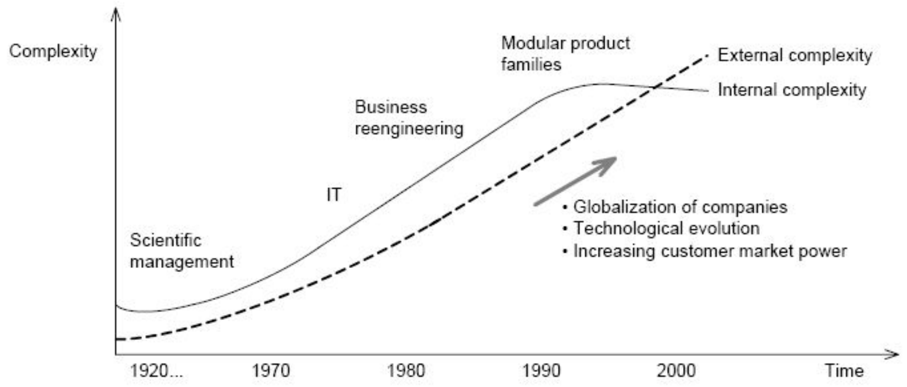


Figure 1.2: Evolution of complexities [Ericsson and Erixon, 1999, p.2]

Figure 1.2 depicts numerous methods and tools introduced in the past to limit the impact of rising external complexity onto internal complexity in manufacturing, information management, and processes.

Usually the product governs processes, manufacturing and information. The product is an interface between external and internal complexity. Dasegning modular products and applying module variants results in product families[6]. The interfaces between these modules need to be clearly specified. To address modular product families from a holistic perspective it needs to be managed in development and realization across the entire lifecycle.

With so many modular product families now being in place, the following observations however, indicate

- *Unsuitable Methodologies:* Modular products families are treated with the same mechanisms as single products, which is unsuitable. Modular product families require a different approach to variant management than single products as interfaces and interactions among modules is crucial.
- *Increasing number of variants:* The number of variants continues to rise and is unmanagable in most companies. Due to cannibalization effects, new variants often do not substantially increase sales but only lead to redistribution from standard to special products. As a result, increased costs are not passed on to the selling price and the profit margin decreases[7].
- *Insufficient decision basis:* Many of the complexity effects cannot be captured using traditional accounting techniques, e.g. overhead calculation. The widely-used method and lack of technical knowledge on the consequences can be misleading when it comes to decisions in variant management.

Planning a standardized architecture within an organization may address a part of these problems and facilitate reuse. With constantly changing requirements within the set of

[6] [Simpson, 2004, p.5]
[7] [P. Child and Wisniowski, 1991, p.5]

products, the variability needs to evolve. Many embedded systems are implemented with a set of alternative function variants to adapt to the changing requirements. Major challenges are in identifying the commonality of functionality, where the designs involve variability (ability to customize). In addition to variants, versions/releases of functional blocks also play an important role for the effective management over the entire product cycle.

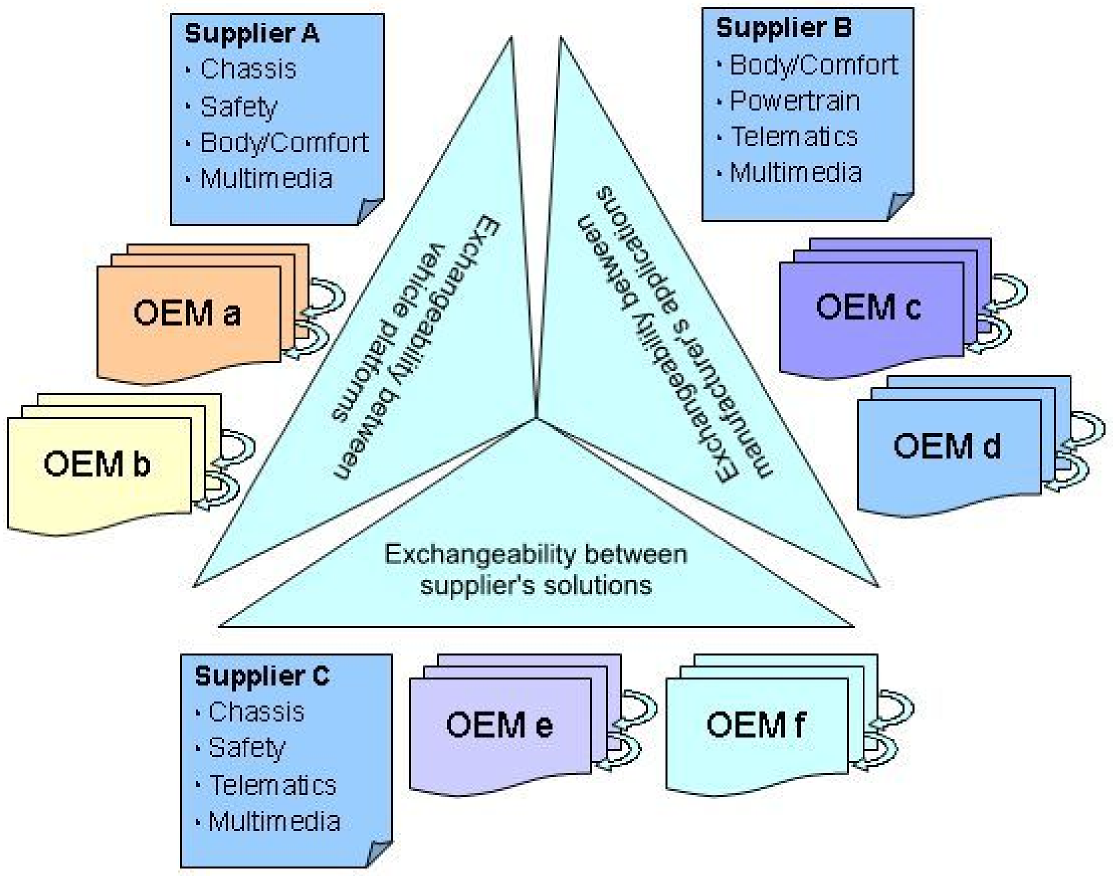


Figure 1.3: External components are a hindrance to variability management

Figure 1.3 depicts a scenario where well established software components tested for performance, safety, and reliability procured from external sources and Original Equipment Manufacturer's (OEM) are causes for a hindrance in managing variability.

For achieving large-scale software reuse, reliability, performance, and rapid development of new products, a software product-line (SPL) is an effective strategy. A SPL is a family of products sharing the same assets allowing the derivation of distinct products within the same application domain.

Enabling variability in software consists in delaying decisions at different software abstraction levels, ranging from requirements to runtime. The object-oriented approach to implement variability is based on the development of a frameworks of reusable software components described by a set of classes and by way instances of those classes collaborate.

Model Based Software Engineering (MBSE) is an industrially accepted approach in the automotive applications. Model-Driven Engineering (MDE) is the use of models as the main artifacts during the software development and the maintenance process. Model Driven Software Development (MDSD) is typically realized in a distributed system environment.

While MDSD facilitates models for the abstract specification of system architectures, their platform specific artifacts are often realized by applying Component Based Software Engineering (CBSE) techniques. Models become artifacts to be maintained along with the code, by using model transformations and code generation.

MDE is related with the Object Management Group (OMG) initiatives, Model-Driven Architecture (MDA) and Model-Driven Development (MDD), which argue that the use of models as the main artifact on software development will bring benefits on software reuse, documentation, maintenance, and development time.

Most MDSD approaches follow the Model Driven Architecture (MDA) concept. In this concept beginning with the specification of a platform independent model, it is then transformed to a platform specific model by applying several generators. The layered Meta Object Facility (MOF) approach is used for creating the models. This approach is also used as the basis for the Unified Modeling Language UML.

1.1 Motivation

In the 1960s reuse started with subroutines, followed by modules in 1970s, and objects in 1980s. In 1990 components appeared followed by services in 2000. Software Product lines are currently the state of the art in the reuse of software. Figure 1.4 shows a short history of reuse in software development. The key idea of product lines is old and based on Henry Fords mass customization to provide a effective way for cheap individual cars. Today many different approaches exist to implement a Software Product line.

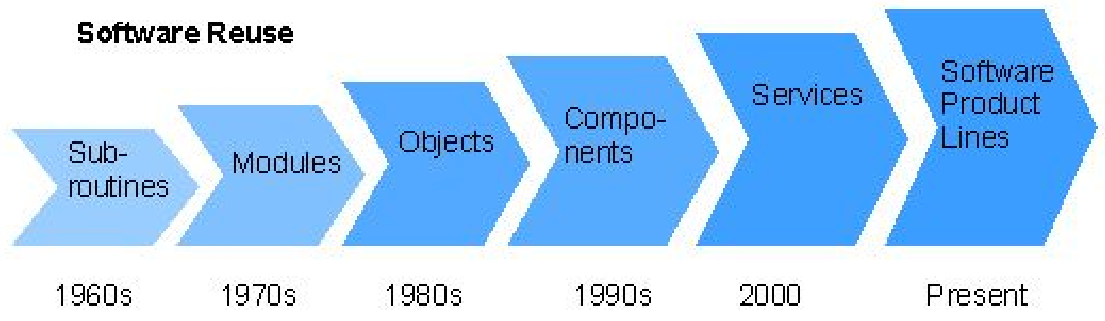


Figure 1.4: Software reuse history [Northrop, 2007]

1.1.1 Software architecture

A layered software architecture is considered[8] for the proposed architecture as depicted on the left side of Figure 1.5.

The definition of software architecture given in the ISO/IEC 42010 IEEE Std 1471-2000: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."[9]

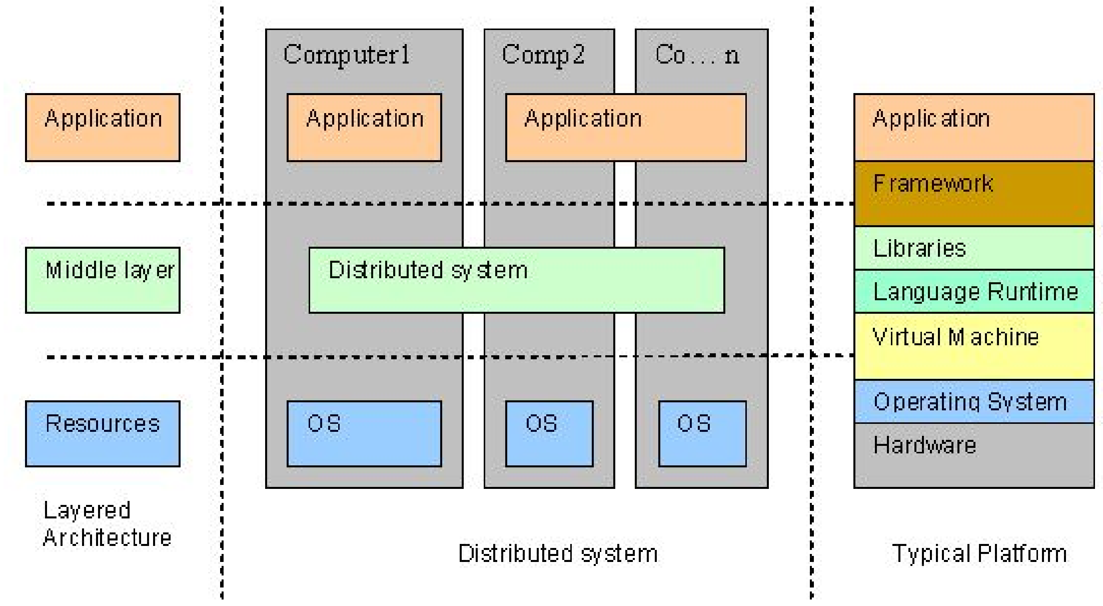


Figure 1.5: Distributed system [Tanenbaum and van Steen, 2006], Typical Platform [Atkinson and Kühne, 2005]

Figure 1.5 also depicts a comparison between proposed layered architecture, distributed systems as proposed by Tanenbaum and van Steen, and the definition of the platform in Model Driven Architectures (MDA) as specified by Atkinson and Kühner. It displays the feasibility of mapping the corresponding artifacts and responsibilities for each layer.

Tanenbaum and van Steen define **distributed systems** as "A distributed system is a collection of independent computers that appears to its users as a single coherent system."[10]

The Kleppe summaries the definition of a **platform** as "A platform is the combination of a language specification, predefined types, predefined instances, and patterns, which are the additional concepts and rules needed to use the capabilities of the other three elements."[11]

8 [Buschmann et al., 1996]
9 [IEEE, 2007, p.3]
10 [Tanenbaum and van Steen, 2006, p.2]
11 [Kleppe, 2008, p.69]

Atkinson and Kühne specify the architecture of a generic platform in MDA[12] as **hardware layer** consists of hardware components (e.g. processor, memory, I/O devices), the **operating system layer** which are augmented services (like file systems, processes, threads), a **virtual machine layer** for the isolation of operating system and hardware components, a **language runtime layer** which contains language constructs expressed as templates of low-level code, a **library layer** of these constructs, **frameworks** and **applications** in the upper layers.

The optional additional predefined functions in the used programming language are contained in the **library layer**, but also middleware solutions are typically delivered as libraries providing several language constructs in the form of an application programming interface (API). In contrast the libraries elements in the **framework layer** contains active control code in a generic way to be used by a family of applications.

Using this layered structure Atkinson and Kühne demonstrate that each layer defines a platform on its own. As a consequence, a generic platform model is applied on every organizational layer leading to a full platform definition with the combination of these platform models.

1.1.2 Reusability through components

Platform evolution is characterized by changes on one or several platform layers of a device, requiring that these changes should not affect the role of this device in the distributed system. Several techniques have been proposed for supporting this evolution.

As the number of elements contained in the platform model of each platform layer needs to be optimized as a consequence of the resource constraints, usage of component based technologies targeting reuse and composition is an essential part of the platform and application development for such devices. Additionally Liggesmeyer and Trapp have noted, that domain specific development of embedded software requires efforts, which should be paid off by the application of the developed framework or platform to related problem domains[13]. This requirement for reuse is solved by component based software development, allowing the composition of individual artifacts relying on well defined interfaces.

The following definition given by Szyperski is often used as the basis for a discussion of CBSE aspects.

Software Component: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party"[14].

Depending on the implementation for the execution environment of a chosen component approach, the notion of a component framework is distinguished from a component platform by Szyperski, with the former defined as a collection of rules and interfaces (contracts) governing the interaction of components. In contrast to component platforms, which are

[12] [Atkinson and Kühne, 2005, p.3]
[13] [Liggesmeyer and Trapp, 2009]
[14] [Szyperski, 2002, p.41]

the foundations for components to be installed and executed on, component frameworks can also be organized hierarchically being components themselves. During platform evolution this fact gets also very important for enabling the adaptation of the component execution environment to changes in the lower platform layers (e.g. changed hardware or operating systems or additionally available programming language constructs).

1.1.3 Domain specific development with models

As noted by Ebert and Salecker [15] embedded software is used for objectives requiring a long lifetime of the system, thus heterogeneity of the underlying platform has to be considered by the embedded software engineer. Also an ever-growing demand for new functionalities and technologies is noted by Liggesmeyer and Trapp[16], requiring efforts of raising the level of abstraction for the development of mobile and embedded software. For embedded systems this is accomplished by the usage of higher level programming languages like C or C++ instead of assembler, and also by the help of Model Driven Development (MDD) techniques.

In Model Driven Software Development (MDSD) models of the software are used as primary artifacts. There is no unique definition of a model, as reported by Muller et al.[17], while comparing several definitions of a model in the literature. But most definitions describe the use case of a model, which is applied to abstractly specify a distinct system. According to Muller et al. the process of modeling also aims to establish representing something by something else. Dealing with the abstraction introduced by such models is provided by model transformators, which are applied at development time in the form of code generators or at runtime in the form of model interpreters as reported by Stahl and Völter in [18].

1.1.4 Evolution in Virtual Organizations

Virtual organizations as defined by Foster and Kesselman.

Virtual Organization: "A collaboration whose participants are both geographically and organizationally distributed."[19]

Each layer contains a number of artifacts, and the artifacts at the same layer are managed by people. Multiple people are involved in the management operation of each layer, as well as multiple artifacts. The state of the system defines the kind of management. Management at development time is performed by designing and implementing the given artifact of a layer in software or hardware. At runtime the administration and usage of available instances of the defined artifacts by a (typically other) person has to be considered.

[15] [Ebert and Salecker, 2009, p.41]
[16] [Liggesmeyer and Trapp, 2009]
[17] [Muller et al., 2009],
[18] [Stahl and Völter, 2006]
[19] [Foster and Kesselman, 2004, p.672]

Bird e .al. notes that "the original idea of a VO as a dynamic group of users with a common goal coming together for a specific, short-lived collaborative venture and then dissolving has never been realized owing to the complexity of deploying and authorizing such a dynamic structure."[20]. Each person and each artifact can be added, replaced or removed at system runtime, constituting a multidimensional evolution leading to the dynamic structure.

This fact is also important in the business domain of automotives. In a distributed system of automotive application the involved people as well as the used artifacts are expected to be organizationally and geographically distributed, which are key features of virtual organizations as defined by Foster and Kesselman.

1.2 Summary

Inspite of all the hype there is a lack of an overall reasoning about variability management.

- SPL approach promotes the generation of specific products from a set of core assets, in which products have well defined commonalities and variation points[21].
- Although variability management is recognised as an important issue for the success of SPLs, there are not many solutions available[22] .
- However there are currently no commonly accepted approaches that deal with variability holistically at the architecture level[23].

Previous sections focus on managing heterogeneity by the layered architecture. This proposed work is used for the identification of their products and for the exchange of data, which has been captured during the lifetime of these products.

1.2.1 Outline of thesis

This chapter introduces a layered architecture for discussing current trends in application and middleware development in the form of Model-Based Development and Component Based Engineering. Considering the fact, that automotive applications are designed increasingly upon distributed systems, support for managing these systems is required. Such support is provided by the concept of Virtual Organizations. Furthermore, the layered architecture has been utilized for demonstrating the heterogeneous aspects in the software development by outlining the various organizational roles and different artifacts.

Chapter 2 focuses on related work in respect of application development with model based techniques, also the latest trends in component based software engineering that have a deterministic role in the classification of commonality for variability identification like

[20] [I. Bird and Kee., 2009, p.41]
[21] [E. Oliveira and Maldonado, 2005, p.5]
[22] [Heymans and Trigaux, 2003, p.5]
[23] [Galster and Avgeriou, 2011, p.5]

structured and unstructured text, semi formal approach for software reuse, component mining, software product line engineering and variability are discussed. Further it also cast an insight of a modeling platform that is used in the case study.

Chapter 3 begins by defining the problem and stating the specification of cases. Based on the challenges discussed the objectives for this thesis are identified and the contributions of the research are enumerated. It further presents an overview of the process and a formal approach (mathematical model) for the process described.

Chapter 4 envisages several key aspects for the implementation of a generic methodology targeting the objectives discussed in Chapter 3 to domain specific models. The proposed work are discussed and associated with the publications presented in Chapter 7.

Chapter 5 summarize the evaluation carried out on the case studies and the results demonstrates the support of the proposed approach for variability identification in the business domain of automotive.

Chapter 6 provides a conclusion and gives an outlook on future work resulting from the insights obtained in this thesis.

Chapter 7 enlists the publications presented at several international conferences during different stage of the work that bears significance to the contribution of this thesis.

Chapter 2

Related Work

The discussion in this chapter envisages the various aspects for variability identification relevant to software modeling tools in the automotive domain. The heterogeneous nature of co-simulation supports cross-domain simulation of models. Each subsystem is modeled and simulated using a domain-specific simulation tool, while the co-simulation platform handles the coupling between these subsystems that enables the holistic simulation of a system. The concept of variability management support suggests itself to be used for different application scenarios. The aspects like structured and unstructured text, semi formal approach for software reuse, component mining, software product line engineering and variability are reviewed. In addition the authoring tool ESCAPE is discussed, which covers one of the use cases.

Because software models are a central aspect in the proposed approach, their usage with respect to the different phases in the lifecycle of a software project is shown. Also existing component models and programming paradigms for embedded software are presented.

2.1 Adding structure to unstructured text

Jonathan I. Maletic and Michael L. Collard[1] proposes that the maintenance and reverse engineering of large legacy software systems is difficult as one has to deal with large numbers of text documents. These documents include source code files in various programming languages, internal documentation, external system documentation (e.g., text, diagrams, tables), and possibility user manuals, bug reports, and version histories.

These documents are all unstructured or semi-structured in nature and techniques for querying, analyzing, and transforming documents with the explicit goal of recovering and identifying metadata from unstructured and semi-structured text. The representation of unstructured and semi-structured text, methods for parsing and translation from raw text and tools for analysis and transformation is needed.

A general approach is to use XML technologies to support storage and extraction of metadata. However, translating unstructured text into XML requires custom parsers

[1] [Jonathan and Collard, 2005]

based on flexible grammar specifications that allows us to skip over uninteresting or ill formed text, are very robust and extremely efficient applied to very large text bases.

For automated feature detection or concept location in raw text, information retrieval methods, namely latent semantic indexing is used to cluster document parts and automatically identify high-level concepts in large document bases.

Analysis begins with the extraction of lexical, structural, syntactical, and documentary information from documents. A lexical approach such as regular expressions can be used. However, robust and efficient regular expressions can be difficult to write especially when determining the matching context. Current compiler technology can address this; however existing compiler-centric parsers take a high level AST (Abstract Syntax Tree) view of a program but ignore the preservation of the original (source-code) text especially with regard to white space, comments, and preprocessor directives. In addition they are not robust and cannot handle code that is either incomplete or has compilation problems.

2.2 Semi-Formal approach to assist software design with reuse

Design with reuse has been accepted as a cost-effective way to software development. Software reuse covers the process of identification, representation, retrieval, adaptation, and integration of reusable software components[2].

Software reuse includes reusing both the artifacts and process of previous software projects, which can be classified as a building blocks approach and as a generative approach. Many technologies are involved in software reuse, for example software libraries, application generators, source code compilers, and generic software templates.

Software reuse deals with two major problems: the retrieval (component representation and library construction) problem and the composition (adaptation and integration) problem.

Most existing approaches have focused on solving only parts of reuse problems. Some works focus on the software components retrieval problem[3] and many of them provide effective automated retrieval systems with reuse libraries. However, few have considered the consistency checking of the retrieved and new components. Some works only deal with software integration and adaptation problem. Some use special specification languages for software component integration and are thus difficult to integrate with existing software component retrieval techniques. Without an approach to assist the entire process of software reuse, the effectiveness of software reuse is limited.

These technologies all involve abstracting, selecting, specializing, and integrating/adapting software artifacts. Abstraction is concerned about what type of software artifacts are reused and what abstractions are used to describe the artifacts. Selection deals with how users can efficiently locate, understand, compare, and select the appropriate artifacts from

[2] [H. Mili and Mili, 1995]
[3] [E. Ostertag and Braun, 1992]

a collection. Specialization helps to tailor generic artifacts for reuse. Integration addresses the problem of how reusable artifacts are integrated to create a complete software system.

Many programmers adopt an ad hoc approach to reuse software system designs and source code through scavenging fragments from the existing software systems and use them as parts of new software system. This approach is often called the design and code scavenging approach. In practice, the overall effectiveness of design and code scavenging is severely restricted by its informality.

Chu et al.[4] proposes a semi-formal approach to software reuse, in which software components are annotated with formal information. The formal information includes the structural, behavioral, and environmental conditions of a reusable component.

The approach consists of the following major steps:

(1) software components are annotated with formal information,

(2) the software components are then translated into predicate transition nets, and

(3) consistency checking of the reusable and new components is carried out using the reachability analysis technique of predicate transition (PrT) nets.

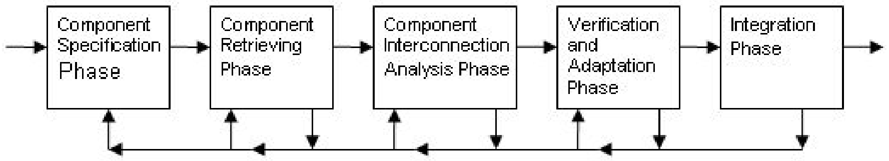

Component Specification Phase
Component Retrieving Phase
Component Interconnection Analysis Phase
Verification and Adaptation Phase
Integration Phase

Figure 2.1: Software design process reuse model

A complex system is composed of a set of possibly semantically interrelated components. A candidate component selected from the reuse library can affect the later selected components that may interact with it. Because of the interrelationship among the reusable components, the results from each phase in the reuse process can be used to improve the performance and preciseness of the other reuse phases. Figure 2.1 depicts the detailed operation as described below:

- **Component Specification Phase:** The specification of a candidate component is either derived from the users requirement statements or collected from other related components in the developed system. The collection of related predicates can be automatically generated from later phases.
- **Component Retrieving Phase:** Based on the specified keywords and predicates, appropriate components are retrieved from the reuse library.
- **Component Interconnection Analysis Phase:** The retrieved component is transformed into a PrT net and then integrated with other components also represented in PrT nets. The integrated PrT net is used to check consistency among the selected components. The consistency checking can be fully automated.

[4] [William C. Chu and He, 2005]

- **Verification and Adaptation Phase:** The retrieved component is checked for any necessary modification and modified accordingly. The needed modification may be revealed during the analysis phase.
- **Integration Phase:** The retrieved or modified components are integrated.

2.3 Component mining and software product line techniques

In the domain of embedded systems, hardware variabilities often happen in an unplanned way. Old components are often substituted by more current ones, which are cheaper and more efficient, thus producing improved versions of the original product. Possibly, some components were not even in existence when the first version of the product was delivered. Many companies of the embedded systems domain want the evolution of their software to occur with the partial or full reuse of the artifacts of their succeeded applications in order to create new ones with little or no maintenance[5]. In this context, it is the responsibility of the software engineers to choose the most suitable development techniques and technologies to achieve this goal in a given domain for a particular set of new requirements.

Leveson and Weiss[6] mentions that many of the reports about success in software reuse practices have been premature because most of the artifacts created are abandoned due to their aging and degradation, even before a satisfactory return on investment is reached. In order to minimize these effects, they warn that the creation of reusable artifacts must be carried out carefully so as to enable easy and safe maintenance and also to expand the functionality of the systems they are built in. The mining of generic components from legacy codes and their subsequent reconnection to the original systems is a technique that can support system revitalization by extending its functionalities. This can also allow the production of a core of reusable assets to support the development of new similar products. However, if legacy codes and components are developed through different paradigms, the development of gateways is necessary[7], i.e., interface adapters that enable functions of the legacy code to easily access component interfaces.

Software reuse refers to the construction of systems from existing artifacts, rather than developing them from scratch. Different techniques have been researched with the purpose of increasing reuse to higher abstraction levels, for example: Components and Software Product Lines.

Components are context-independent composition elements, implemented for a certain specification, distributed in an autonomous way and that, through their interfaces, add functionalities to the systems they integrate[8].

However, in the domain of embedded systems, components are not usually autonomous elements (run-time components), but codes written in high-level languages, that can be

[5] [B. Graaf, 2003]
[6] [N. Leveson, 2004]
[7] [OBrien, 2005]
[8] [Szyperski, 2002]

connected to the code of an application during the creation of system versions (build-time components). Easily-adaptable generic components can support the creation of reusable solutions for similar requirements in different domains. For such, they have to support the commonalities and variabilities of the domains to which they are applied[9].

The process of mining assets often refers to the non trivial activities of uncovering and extracting potentially useful and reusable artifacts built into the legacy systems. Automation tools can be designed for this purpose, but handling such tools requires the work of experienced software engineers to properly analyze the results and refine mining rules.

Features are properties of a domain, visible to the user, that enable the identification of commonalities between related systems as well as their variabilities. With the purpose of improving the identification of important or special properties of a domain during the analysis phase[10], introduced the feature model in their FODA method (Feature Oriented Domain Analysis). In this model, the features are arranged hierarchically in a tree structure where they are connected by structural relationships, forming groupings. Each feature has its own specifier that defines it as mandatory, optional or alternative. Different notations have been proposed to extend the representative aspect of the feature model in relation to different types of structural relationships.

2.3.1 Activities to mine reusable assets from legacy systems

Usually technologies for traditional software development do not consider the specific needs associated to the creation of embedded software and the usual constraints of this domain, such as memory limitation, power consumption and hardware changes.

The framework of activities foresees the creation of software components from features built into embedded legacy systems. The feature mining process is based on an adaptation of the four steps[11] as depicted in Figure 2.2. It considers the availability of legacy source codes and also the minimal documentation of the existing hardware and software elements, i.e., peripherals, operating system etc.

Step (1): The process begins with a meeting that brings together programmers, users, and other people directly or indirectly involved with the legacy systems. The purpose of the meeting is to identify the current deficiencies and the immediate and future requirements of the systems.

Step (2): Through the information obtained and with customer support, it is possible to come up with a preliminary analyses of the technical and economical feasibility of the project to revitalize the systems, based on the previously identified requirements. These requirements serve as a guide for outlining a suitable strategy for the development of the project, as well as for the establishment of its scope, goals and priorities.

Step (3): This step starts with the creation of a group of technical people, that include one or more domain specialists, to better understand the legacy systems and their documentation and to analyze, as best as possible, the specific concepts of the domain and

[9] [Crnkovic, 2005], [J. Bergey and Smith, 2000]
[10] [Kang et al., 1990]
[11] [Ramos and Penteado, 2008]

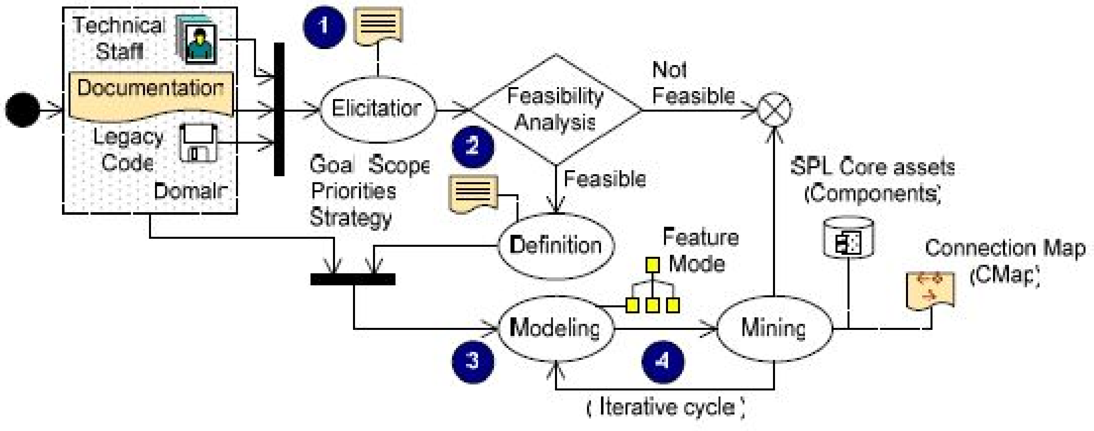
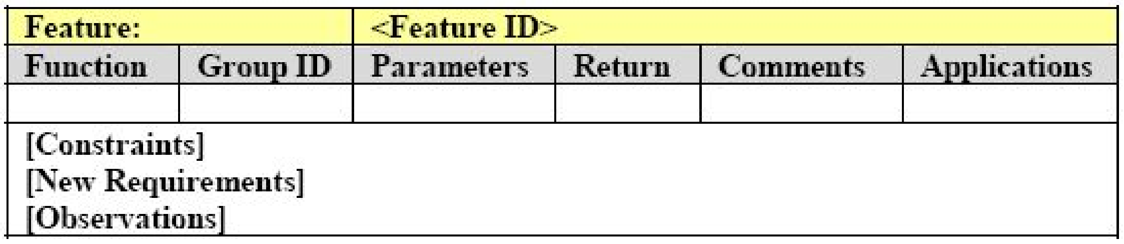


Figure 2.2: Process model to mine reusable assets from legacy codes [Ramos and Penteado, 2008]

its particularities. The priorities of the project help to define the set of knowledge to be acquired initially, which can involve the entire system or just part of it, depending on whether the mining is to be carried out fully or gradually.

Step (4): Based on the information acquired and documented up to this point, the mining activities begin and may vary depending on the goal of the project and on the previously established strategy.

In a gradual component mining process, the feature model of the domain can be simple at first, just detailing the features that represent immediate requirements and current deficiencies of the legacy systems. As the components are developed, refinements and additions of new features can be made to the existing feature model. This process is performed iteratively.



Feature:		**<Feature ID>**			
Function	**Group ID**	**Parameters**	**Return**	**Comments**	**Applications**
[Constraints] **[New Requirements]** **[Observations]**					

Figure 2.3: Connection Map (CMap) and its elements

The legacy code serves as a guide for the design of the generic interfaces of the mined components, which must implement, at least, the actual system functionalities. Techniques such as inheritance and configurable interfaces can be used to extend the system functionalities through the support of domain variabilities. For every feature of the feature model, an inspection in the legacy code must be carried out to identify all the functions directly related to it.

Based on the properties of these functions, a Connection Map (CMap)[12] is built and used to back the design of the generic interfaces of the component related to the feature. When there is a group of similar systems, CMap can map the connection of their codes with the common feature in a unified manner. This increases the generality of the component interface in development, widening the possibilities of its reuse in new products.

The format of the Connection Map (CMap) is presented in Figure 2.3 and the columns refer to its elements.

Feature ID: Feature name in the feature model. In order to facilitate its location, an optional extended ID can be supplied, which includes all the features that precede the current one in the tree branch it belongs to. For example, the feature B child of feature A, can be identified by AB;

Function: Function of the legacy code that fully or partially implements the related feature;

Group ID: Unique number to identify distinct groups of functions. If similar functionalities are performed by different functions in different applications, these functions must be assigned to the same group ID when inserted into the CMap. The revitalized code must provide a generic component interface that supports all variations documented in the group;

Parameters: List of function parameters. The objective of each parameter should be easily identified through its description, written in the form <name> | <data type> as in the UML. For functions without parameters, None is used.

Return: Function return. It follows the same rule described above for Parameters;

Comments: It contains a brief description of the functionality implemented. It can also contain relevant information to help in the feature mining of the legacy codes;

Applications: Each similar legacy application of the domain receives a unique ID at first. Once a function is identified within one or more of these applications, the IDs of these applications will appear in this column of the table. This fact frequently occurs when the adopted reuse technique involves code sharing or duplication.

This list indicates if feature mining involves multiple applications;

Constraints: During feature mining, technical or functional limitations of any kind may appear and should be documented in this section and taken to the domain specialists, who will decide on the best solution for them. Specific meetings with experts can be scheduled to analyze and solve potential issues; New Requirements: When the solution for a particular feature limitation has to be addressed by the component, its description is appended in this section and it will become a part of the requirements for the component implementation.

[12] [Ramos and Penteado, 2008]

Observations: Free text area to facilitate the communication of the team members who are mining a common particular feature. This can include the current status of the task, a list of pending issues etc.

The CMap creation enables a wider understanding of the legacy code through the accomplishment of code inspections, which evaluates the coupling level among the code and each of the features to be mined, facilitating the performance of impact analyses.

2.3.2 Legacy code revitalization through the mining of assets

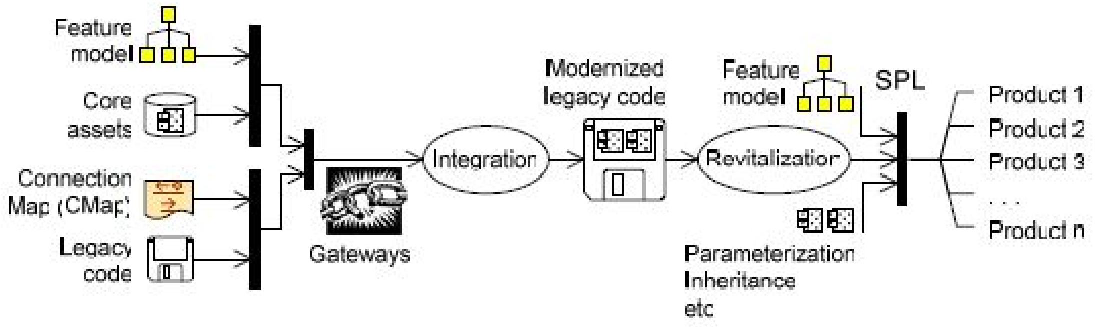


Figure 2.4: Legacy code revitalization through the mining of assets [Ramos and Penteado, 2008]

The resulting artifacts of the asset mining process, described in the previous section, are: a) a feature model, that models the domain features, b) a CMap, that documents the connection of these features with the legacy systems belonging to the domain and c) a core of reusable software components, whose generic interfaces implement, at least, the current functionality of the legacy systems. Although the components can be used independently to create new products, they also can, in the proper manner, be reconnected to the legacy code to improve its structural organization and to aggregate new functionalities to the application, extending the life of the original product. This revitalization approach considers the implementation of gateways, as proposed by[13], which enable the use of different paradigms for designing components.

The CMap and the component interface documentation, produced by the asset mining process, supply enough information to implement gateways that act as interface adapters. In order to rebuild the original systems using the recently created artifacts, legacy functions, now implemented as component methods, must be removed from the legacy code. However, the original function calls, which will be redirected to the components by the gateways, must be maintained.

In this manner, legacy systems revitalization is obtained without making any changes to the code structure and without interfering in the daily activities of the maintainers of the

[13] [OBrien, 2005]

systems. Figure 2.4 shows the process described, highlighting how the mined components can extend original system functionalities or enable the creation of new similar products.

The revitalization, when performed gradually, can facilitate the execution of validation tests of the rebuilt systems, since the changes are focused on isolated features and are entirely implemented in the components. The same existing test cases for the legacy system can be used for the revitalized one, to check if the functionality has actually been maintained.

2.4 Software Product Lines Engineering

Software engineering aims to provide techniques for developing better software products with less resources. Better software refers to several distinct desired characteristics such as correctness, reliability, usability, robustness, comprehensibility, extensibility, and maintainability of software. On the other hand, less resources means less human activity.

A major goal of software engineering is effectiveness in software reuse, the capability of using existing pieces of software in different contexts, bringing advantages on all the characteristics stated above. The state of the art strategy for achieving large-scale reuse is based on the adoption of software product-line (SPL) architectures[14], describing product families and capturing the variability within them. A key issue on SPLs is variability management - the necessary mechanism for obtaining the several distinct products of an SPL.

There are several key motivations for using software product line engineering[15]. The most important are discussed in the following description.

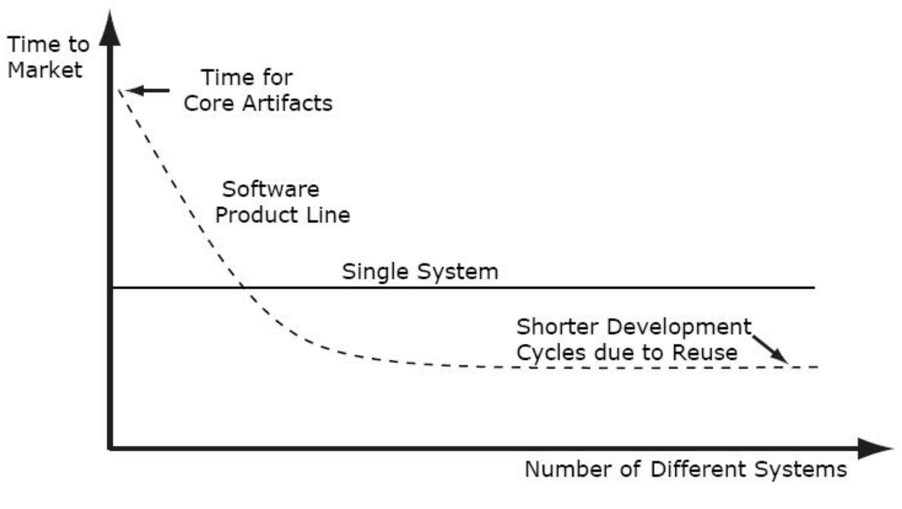


Figure 2.5: Time to market single system Vs. SPLE [Clements and Northrop, 2007]

[14] [Bosch, 2000], [M. Jazayeri and van der Linden, 2000]
[15] [Frank van der Linden Klaus Pohl, 2005]

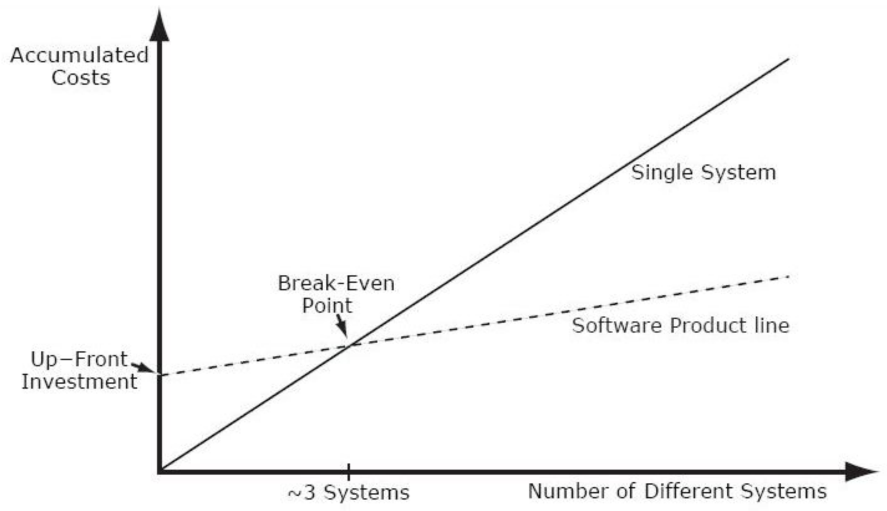


Figure 2.6: Development cost single products Vs. SPLE [Clements and Northrop, 2007]

Reduce development costs An essential reason to apply a new engineering practice is the economical justification which means cost reduction. At the beginning of a SPLE the costs are higher compared to a single system, because of the common platform development and the reusable parts. The following products can be made cheaper because of the commonalities in the product line. This means a company has to make an investment to create the platform before it can reduce the costs per product. Figure 2.6 shows this behavior. At the beginning the accumulated costs are higher, but after the break-even point of approximately three different products, the software product line is the better strategy.

The use of a common platform and the reuse of artifacts in numerous products leads to the next point.

Increase quality The platform is reviewed many times and tested in several products, so finding errors in a product increases quality in all products of the product line.

Reduce time to market This aspect is important in many business areas because reducing the time to market can be a key motivation. Figure 2.5 shows the different traces of time to market with both strategies. Similar to the costs (Figure 2.6) the product line shows the advantage after a certain number of implementations.

Reduce maintenance effort As a result of the architecture the maintenance effort is reduced, because there is only one common platform and the same artefacts for all kind of products.

Coping with evolution and complexity Implementing one new artifact for the platform gives the opportunity to put it in all other products of the SPL to set trends. The reuse of parts reduces the complexity, because the development with higher abstraction of already implemented parts is easier.

Benefits for the customer Last but not least, there are benefits for the customers who get a product which is adapted to their needs with an adjusted price.

2.4.1 Approaches for Software Product Line

Basically there are three different approaches to start with a Software Product Line[16].

Proactive In this case, the Domain Engineering Process is the first step. The development process must take into account all feasible products of the Software Product Line. The complete set of all artifacts is developed from scratch which is of course a risk for the company. This needs predictive knowledge and a clear strategy. After the SPL has been constructed, the new products will quick come to market with a minimum of coding effort by exploring the variability.

Reactive The Reactive Approach starts with just one or few more products. These are used for Domain Engineering and for further products. This leads to lower costs at the beginning of a new project compared to the Proactive Approach, but the architecture and the core artifacts must be robust, extensible and appropriate to future needs.

Incremental / Extractive This approach starts with existing software products or systems. Then reusable artifacts were extracted to create a first version of a Software Product Line. Next products extend the artifacts incrementally.

The choice of an appropriate approach depend on the strategy of the user or company and there is no specific procedure to choose the right one.

2.4.2 SPL Framework

A popular method for implementing SPL variability is through the development of object-oriented application frameworks, a collection of classes implementing the shared architecture and common functionality of a family of applications. Frameworks allow the implementation of the application-specific parts of the different products of the product family, using mechanisms such as aggregation/delegation, inheritance, parameterization, object composition, etc.

The adoption of framework-based SPLs has demonstrated to be successful for mediumsized product families i.e. families representing an application-domain of reasonable size. With the increase of framework size, its learning and usage become naturally more complex. The selection of the adequate parts of the framework for derivation of a specific product and how to customize them becomes a difficult task for the framework user. Nevertheless, large frameworks involve considerable efforts on production and maintenance of documentation by the framework developers.

[16] [Northrop, 2007], [Uira Kulesza and Borba, 2007]

Software Product Line (SPL) consists of a group or family of products that share a common architecture and belong to a particular domain. The purpose of SPLs is to increase the efficiency of development processes by exploring the identification and reuse of commonalities and managing variabilities of related products[17]. Figure 2.7 shows a generic SPL engineering process comprised of two main activities: Domain Engineering creates the core of reusable assets and the SPL development infrastructure, and Application Engineering develops new products, family members, from the available resources[18]. The core of reusable assets of a SPL contains its requirements, domain models, architecture, software components etc.

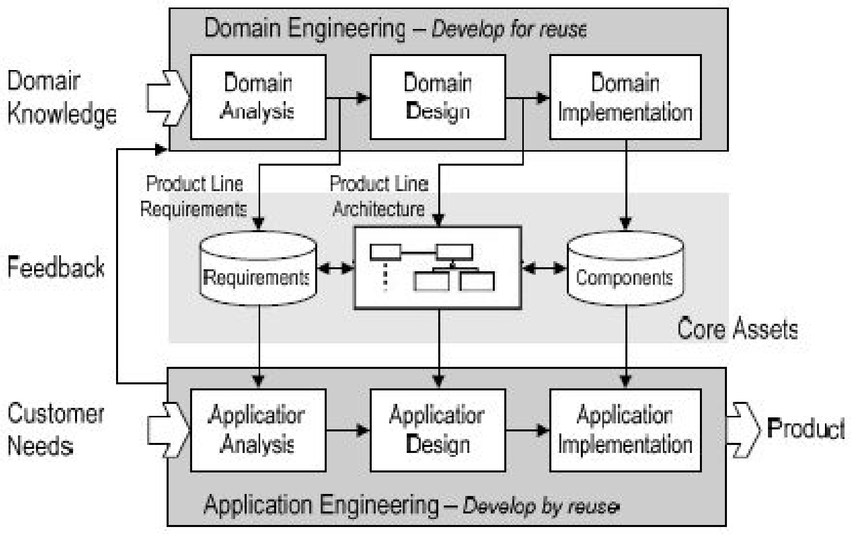


Figure 2.7: Generic SPL engineering process [Ziadi, 2003]

The paradigm of software product line engineering differentiates between three processes[19]. These processes are all iterative and partially parallel. Domain Engineering is the process which is responsible for creating the platform, defining and realizing all the commonalities as well as variabilities of the Software Product Line. The second process is called Application Engineering where all the applications are built by using the domain artifacts and exploiting the variability of the software product line. The management is the third process dealing with the economic aspects of the Software Product Line. This part of the development is handled as own process in the SEI framework as depicted in Figure 2.8 whereas it is a part of domain engineering in the other framework.

[17] [Atkinson et al., 2001]
[18] [Ziadi, 2003]
[19] [Frank van der Linden Klaus Pohl, 2005]

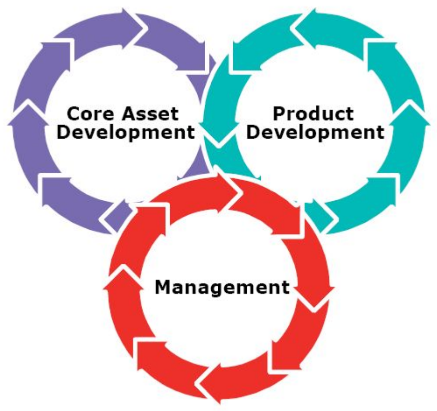


Figure 2.8: SPLE framework SEI [Northrop, 2002]

2.4.2.1 Management

The management process involves economic aspects of the software product line. The business strategy and the product portfolio are the main parts. The company goals are the base for the so called *product roadmap* determining the ongoing and future set of product types, the commonalities and the variabilities. The roadmap also defines a schedule of market introduction. For software product line success the management must be closely linked to the other processes[20]. This behaviour is illustrated in Figure 2.8
The management also controls the iteration between application and domain engineering to stay on the roadmap. In traditional software engineering the management is responsible for creating a single result in a defined amount of time, so product management in product lines differs in some points from single systems:

- The goal is to generate a complete product portfolio (product variants / roadmap).
- The product variants are similarly.
- The platform for the product variants is crucial because it effects all products.

An often found definition for product management in software product line engineering is: *Product management is the sub-process of domain engineering for controlling the development, production, and marketing of the software product line and its applications.*[21]

The products must make the best use of the domain artefacts and variabilities and the domain artefacts must be feasible for the products of the roadmap. Therefore the management is involved in the whole process of software product line engineering[22].

[20] [Frank van der Linden Klaus Pohl, 2005], [Northrop, 2002], [Paul C. Clements and McGregor, 2005]
[21] [Frank van der Linden Klaus Pohl, 2005]
[22] [Paul C. Clements and McGregor, 2005]

2.4.2.2 Domain engineering

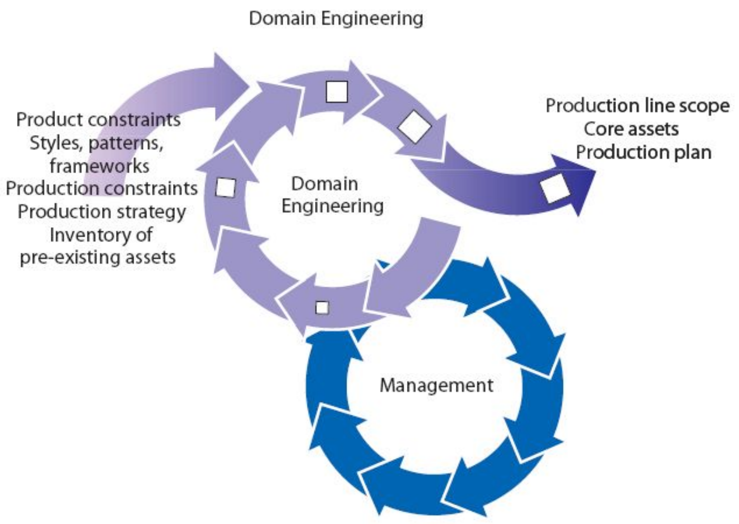


Figure 2.9: Core asset development [Northrop, 2002]

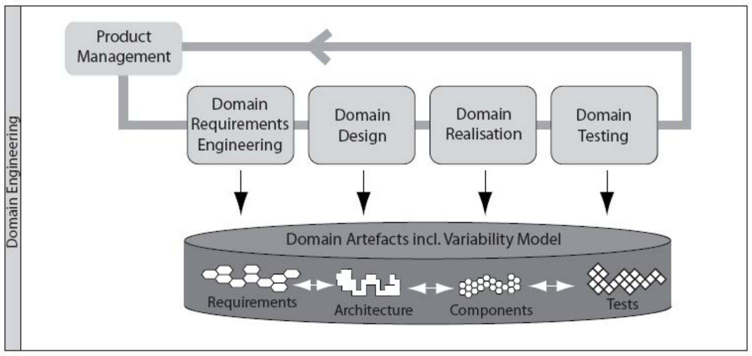


Figure 2.10: Domain engineering [Frank van der Linden Klaus Pohl, 2005]

The process domain engineering defines the commonality and the variability of the software product line. In cooperation with the management process the set of applications will be selected. The reusable artifacts can be constructed to determine the variability. Figure 2.10 and Figure 2.9 illustrate Domain Engineering and Core asset development respectively[23].

The input for Domain Requirements Engineering is the roadmap from the management which causes reusable requirements as output. The format of this requirements is not specified and may be textual or anything else. These requirements are not for special applications, but for all possible applications of the product line. In this process, the

[23] [Frank van der Linden Klaus Pohl, 2005], [Northrop, 2002]

chosen approach to start with a product line must put into practice. The Domain Design sub process takes all requirements as input and creates a reference architecture. In this stage technical reasons may influence the internal variability. After defining the reference architecture Domain Realization is done. Reusable components can now be designed and implemented. It is possible to test the implemented components against their specification to reduce the later tests of the whole application. This sub process is called Domain Testing. All discussed single sub processes are iterative.

Figure 2.9 shows the same processes in another way. The output are again reusable artifacts, which the company uses to implement all products of the roadmap. Production constraints influencing the development may be external or company specific standards which must be applied to all products[24].

2.4.2.3 Application engineering

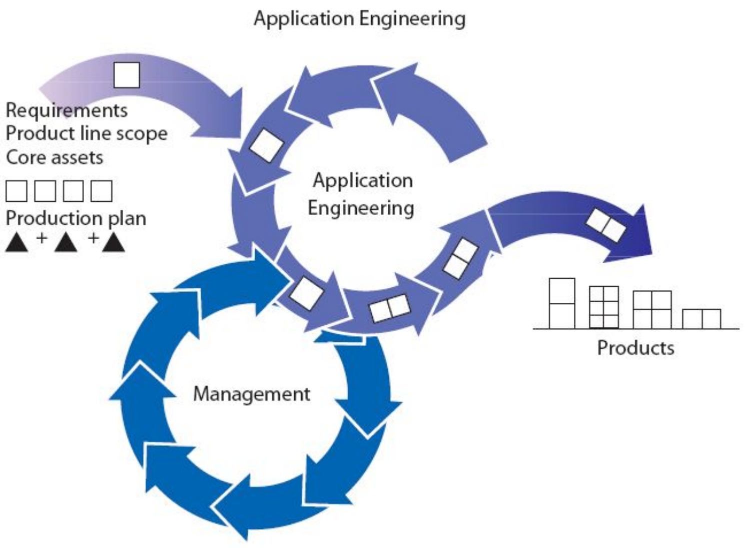


Figure 2.11: Product development [Northrop, 2002]

The Application engineering process tries to achieve a high reuse of the domain artefacts during creation of a new application[25]. Exploring the variabilities of the Product Line should lead to many resulting applications. Figure 2.11 and Figure 2.12 illustrates the two views on application engineering.

This process is also divided in sub processes as illustrated in Figure 2.12. Application Requirements engineering is the first step to develop the specification of the application requirements. Differences between the provided reusable domain artifacts and the needed requirements can be found because the domain artifacts are the only input for this sub process with the product roadmap kept in mind. The result of this sub process is a requirement specification for an application. Modifying the reference architecture to fit the

[24] [Northrop, 2002]
[25] [Frank van der Linden Klaus Pohl, 2005], [Northrop, 2002]

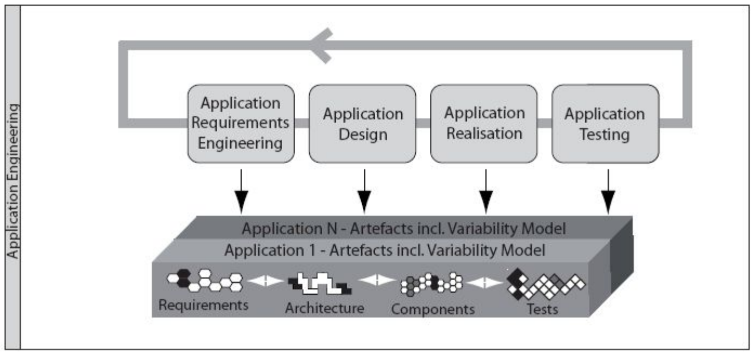

Application Engineering
Application Requirements Engineering
Application Design
Application Realisation
Application Testing
Application N - Artefacts incl. Variability Model
Application 1 - Artefacts incl. Variability Model
Requirements
Architecture
Components
Tests

Figure 2.12: Application engineering [Frank van der Linden Klaus Pohl, 2005]

needs for a specific application is done in Application Design. The reference architecture and the specification are the input parameters and the output is a adjusted architecture for a single application. Now the application is created in the sub process Application Realization. Reusable components are selected and configured. It is also possible to implement application specific parts. The result is a running application which will be tested in the last sub process called Application Testing. Figure 2.12 illustrates the set of applications resulting of Application Engineering in the Software Product line. The Figure 2.11 gives another view on Application Engineering by taking again the domain artifacts in respect to the roadmap to create customized products. Here an explicit connection to the management is shown.

2.5 Variability

Frank van der Linden Klaus Pohl defines variability as[26]

"*Documenting and managing variability is one of the two key properties characterizing software product line engineering. The explicit definition and management of variability distinguishes software product line engineering from both singles-system development and software reuse.*

Variability is defined during the Domain Engineering process where it is refined in all subprocess illustrated in Figure 2.10. Variability describes the ability of domain artifacts to be used in different applications of the product line roadmap[27]. Two important terms are variability subject which is a variable item in the real world and variability object, as a specific instance of the subject. Further abstractions which are interesting related to variability in software Product Line Engineering are variation points and a variant. This is the representation in a context of the variability subject (variation points) and the variability object (variation point).

The variation points and the variants are used to define the variability of a Software Product Line. When creating a SPL, first all variation subjects must be declared, then the

[26] [Frank van der Linden Klaus Pohl, 2005]
[27] [Bachmann, 2005], [Jan Bosch and Pohl, 2001], [Pohl et al., 2005]

variation points have to be worked out and finally the according variations. To document the variability the following questions have to be answered[28]:

What varies? The mapping between the real world and the variation points should be documented.

Why does it vary? The reason can be internal like technical constraints or external like laws, standards or needs from stakeholders.

How does it vary? All possible variants should be documented and linked to the domain model elements.

From whom is it documented? There is a difference between extern and intern. Some documents are only for internal use and some are relevant for the stakeholder.

Figure 2.13 shows a graphical notation to model variability. It defines the variation points, the variation and the dependencies between them. Dependencies are divided in variability and constraint dependencies.

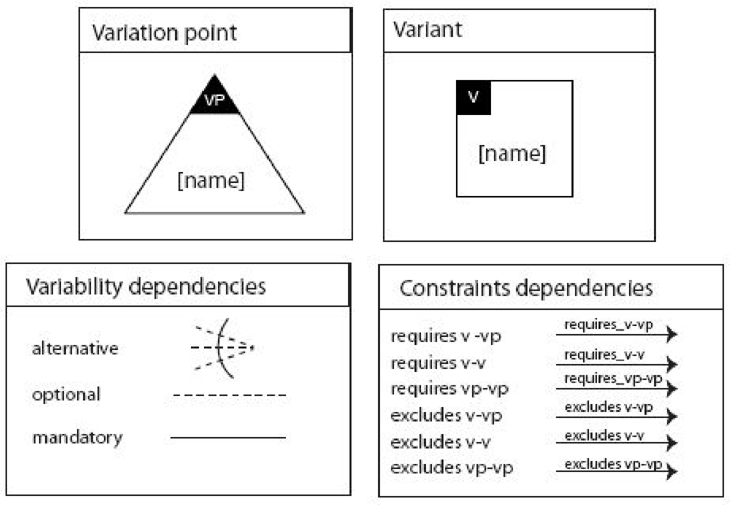


Figure 2.13: Graphical notation for variability[Frank van der Linden Klaus Pohl, 2005]

In big projects there are thousands of variation points and variations, so the organization and optimization is a challenging research area. Variation management is a main-criteria when selecting a tool for big Software Product Lines.

2.5.1 Variability management process interaction with the Product Lines (PL) development process

Figure 2.15 presents the interaction between the core asset development process[29], represented by the activities vertically aligned on the left, and the variability management

[28] [Pohl et al., 2005]
[29] [E. Oliveira and Maldonado, 2005]

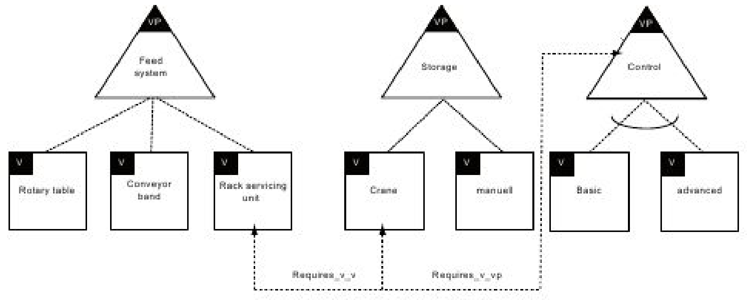


Figure 2.14: Example of graphical notation

process, represented by the activities defined inside the right rectangle. The variability management process activities are executed by the PL manager. It is an iterative and incremental process that runs in parallel with the core asset development.

After the execution of each activity of the core asset development, the variability management process is executed, thus progressively taking as input the output artifacts of the core asset development. As the activities are executed, the number of variabilities tends to increase. As the process is iterative, variability updates are allowed from any activity of the process. The input and output artifacts of the activities are defined as follows. However, note that the input artifacts are made available according to the progress of the core asset development activities. The proposed process consists of the following activities:

- Variability tracing definition, which takes the use case and the feature models as input and generates the variability tracing model as output;

- Variability identification, which takes the use case, the static type, and the feature models, plus the component model as input and generates the same artifacts with the variabilities identified as output;

- Variability delimitation, which takes the use case, the static type and the feature models, plus the component model as input and generates the same artifacts with the variabilities limited as output; and

- Identification of mechanisms for variability implementation, which takes the static type model and the component model as input and generates the variability implementation model as output.

In addition, the process is supported by a metadata model which describes the relationships among the PL artifacts. The process consumes artifacts from the PL core asset as well as producing information for it. They feed the variability management process and return to the core asset the variabilities identified and limited. However, there are models such as the variability tracing and implementation models that are originated in the variability management process.

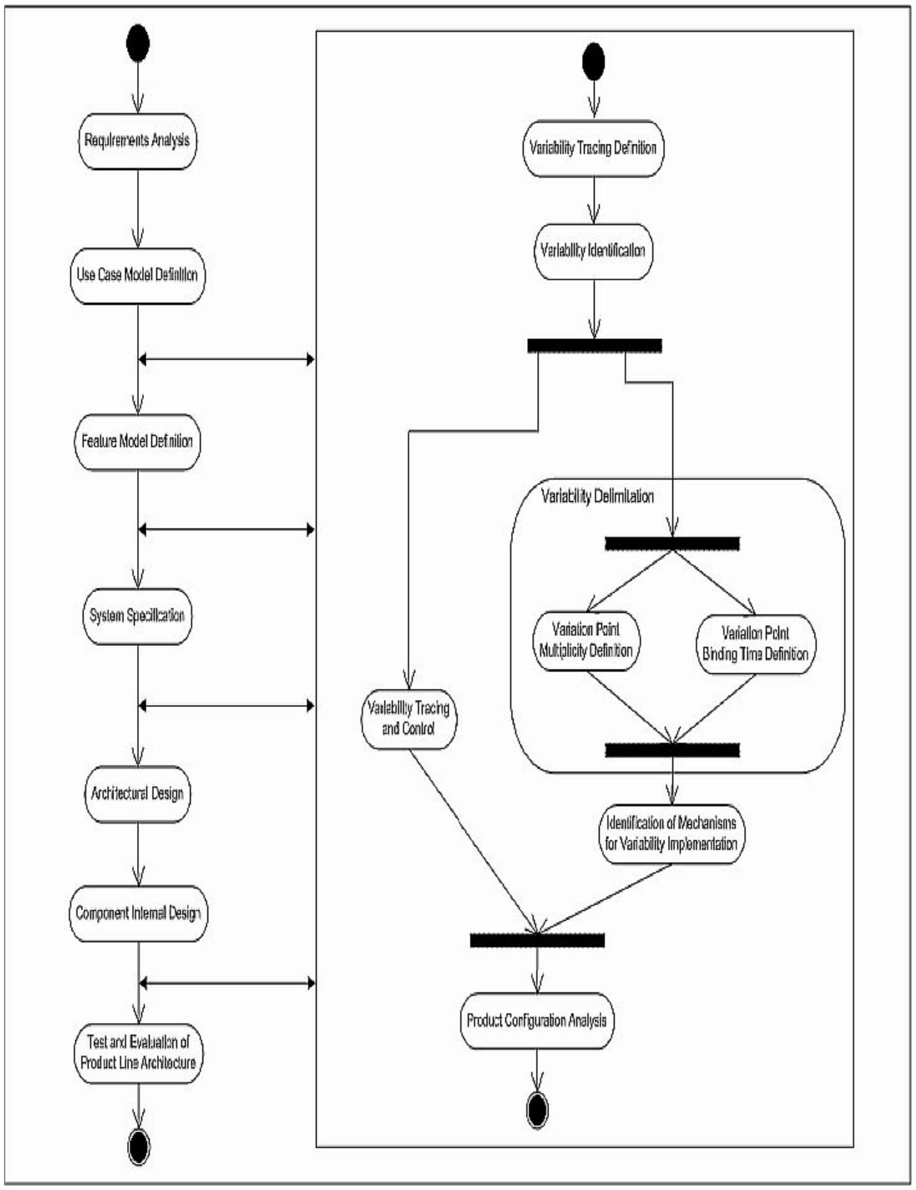

Requirements Analysis
Use Case Model Definition
Feature Model Definition
System Specification
Architectural Design
Component Internal Design
Test and Evaluation of Product Line Architecture
Variability Tracing Definition
Variability Identification
Variability Delimitation
Variation Point Multiplicity Definition
Variation Point Binding Time Definition
Variability Tracing and Control
Identification of Mechanisms for Variability Implementation
Product Configuration Analysis

Figure 2.15: The variability management process and its interaction with the PL development process[E. Oliveira and Maldonado, 2005]

2.6 ESCAPE® architecture

ESCAPE is an authoring tool for graphical definition, manipulation and analysis of functional networks with ability to manage a wide range of configurations. It provides diagnostics tools like dependency analysis, fault back tracking, and support for mathlab/simulink simulation models and various programming language implementations The modeling can be performed without regard to their subsequent implementation in software (SW) and hardware (HW)[30].

ESCAPE provides modeling platform for design and analysis of distributed embedded real time systems of any degree of complexity and size. From the mission level at which functions and features can be defined, tested and documented by using an executable specification via the solution level, where functional solutions can be reused and partitioned to graphically defined architectures of buses and devices to the implementation level where the final software is generated in todays existing environments as well as the electrical schematics.

ESCAPE supports 3 different views:

- FSB (Functional structure builder) facilitates Functional Modeling to build the structure of the model,
- FTB (Function type builder) provides Solution Modeling defining hardware and software types, and
- HSB (Hardware Structure Builder) is Architecture design modelling which allows networking ECUs and mapping the software functions.

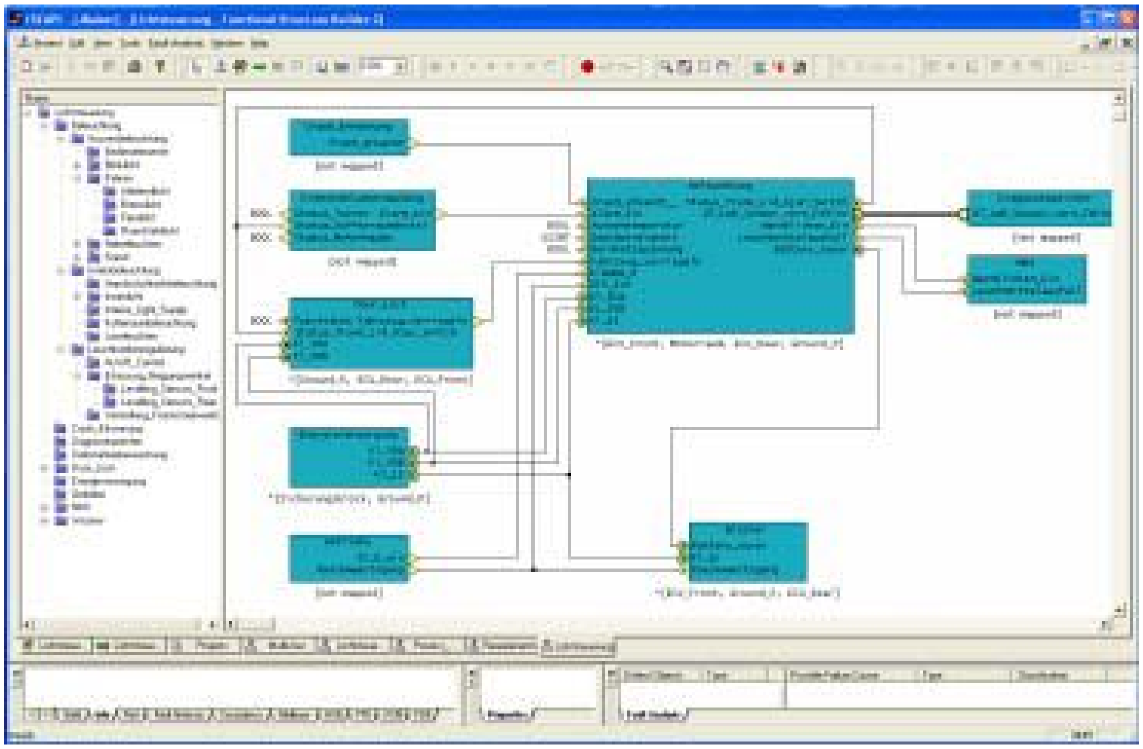


Figure 2.16: Functional structure builder view

[30] [GIGATRONIK, 2009]

Figure 2.16 depicts the FSB view. The project is displayed in the form of a tree that represents the product model structure. The left pane displays the hierarchy of compound functions within a project. Groups or teams can work independently on a sub-tree for the development of functional parts of the product model. The right pane displays the schematic of the functional structure, and leafs of the hierarchy are the instances of the hardware- and software-types defined in the FTB. These sub-trees can then be integrated into a single large model in FSB. It also provides tools to trace the forward and backward impact on the model.

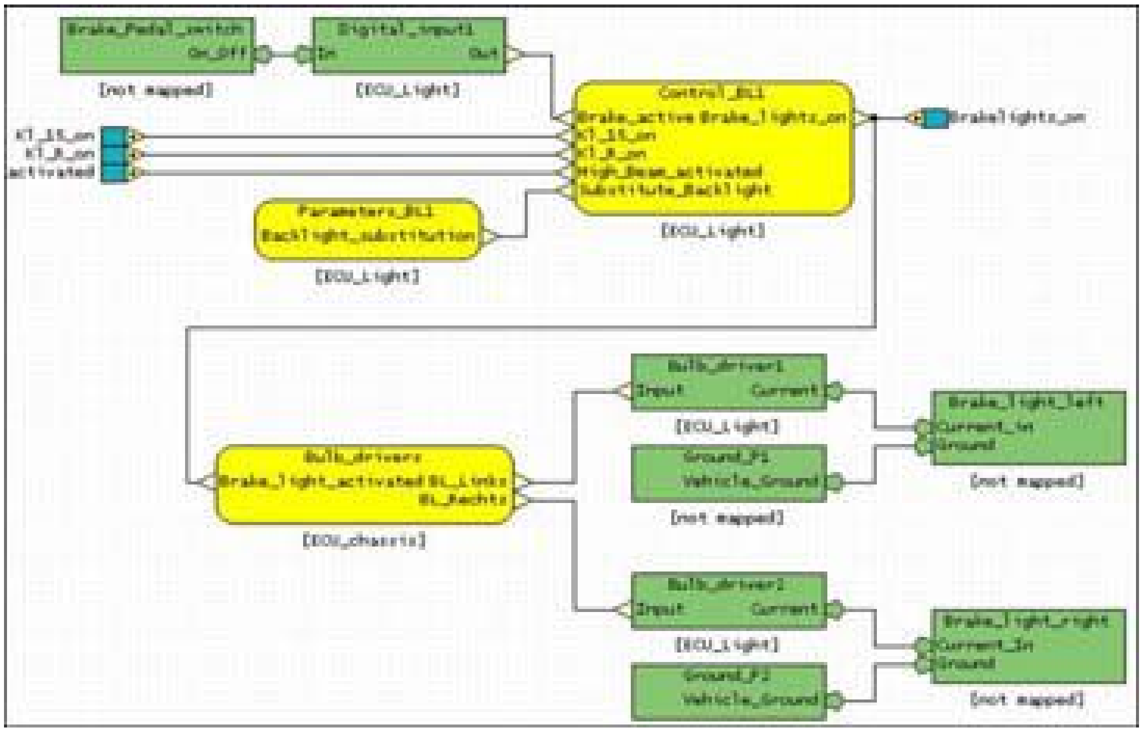


Figure 2.17: Functional type builder view

Figure 2.17 depicts the FTB view. The left pane of the FTB view displays the hierarchy of hardware types and software types. These types can have basic- and user-defined subtypes. The user-defined hardware types can have subtypes like input hardware (e.g. sensors, switches), output hardware (e.g. actuators), and control hardware which can be further sub-typed to any depth grouped by similar hardware domain, hardware logic, and actual hardware type. The right pane of the FTB view shows the definitions of these types. Similarly the basic software types include integer, boolean, float, double, etc., and the user defined software types can have subtypes to any depth grouped by domain related user-defined software types based/derived from basic types, software parameters, etc.

Figure 2.18 depicts the HSB view for System architecture design and optimization. The HSB view shows the hierarchy of networked ECUs interconnected by bus systems. It provides a graphical design view to the technical architecture of a distributed embedded system, the HSB (hardware structure builder). The HSB enables to create devices, buses,

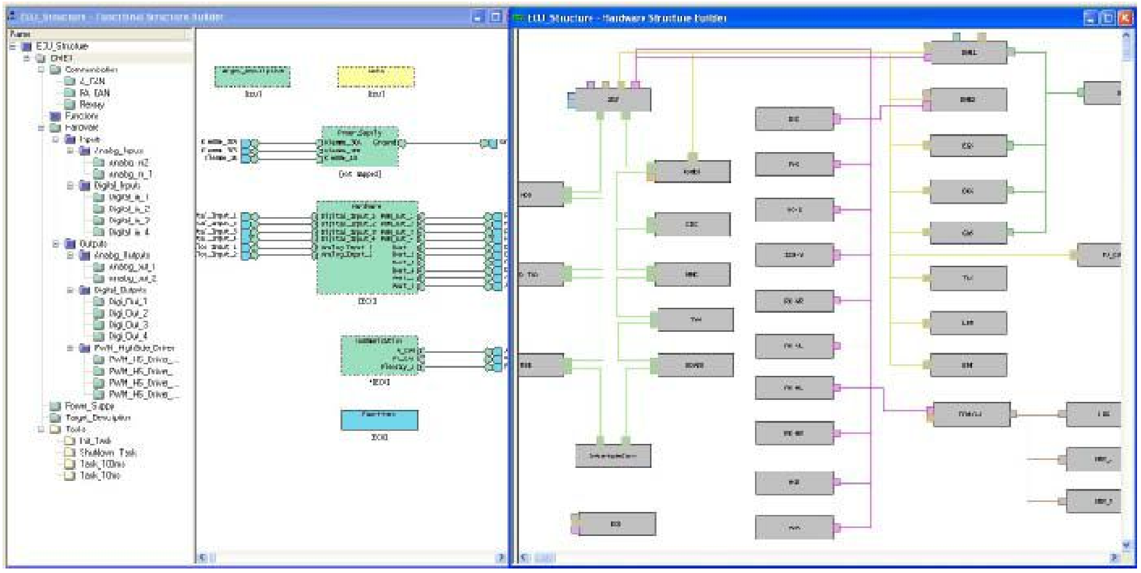


Figure 2.18: Hardware structure builder view

messages and protocols. Existing message catalogs can be transformed into ESCAPE-promessage types, in terms of AUTOSAR PDUs. Thus the back-annotation of messages to logical functions can be achieved. The ECU structure view enables the user to look inside an ECU and to configure it via drag & drop mechanisms. Functional elements mapped to the ECU can be mapped to the real resources of an ECU just by dropping into a graphical representation of the ECUs resource, e.g. a task or an input/output. ESCAPEpros dependency analysis supports work on the logical design as well as on the physical design. Signals are generated automatically by mapping functions and solution elements to components in the HSB. Signals can be mapped into messages and ESCAPEpro automatically carries out a bus load calculation. Protocols like SAE J-1939 are supported. ESCAPEpro works independently from bus technologies. Support for buses like CAN, LIN and FlexRay can be loaded as a plug-in at any time needed.

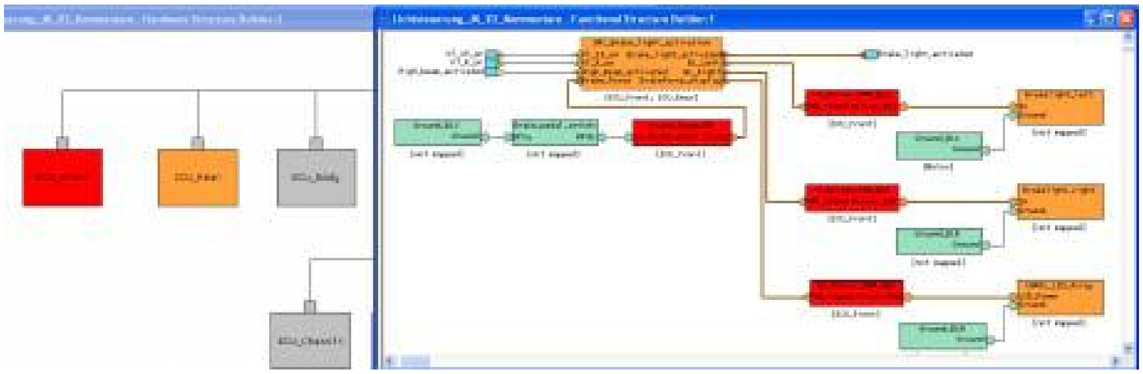


Figure 2.19: Failure analysis

Failure effects analysis Figure 2.19 depicts the failure mode effects analysis. Any logical element, sensor, actuator, HW-channel, SW-module, ECU, bus cable or signal can be marked faulty. The result displays the analysis graphically across all hierarchies and views. The left picture shows an analysis, where an ECU has been marked faulty, the logical view marks red all elements mapped to that ECU and marks orange the elements depending on information from that ECU (secondary faults).

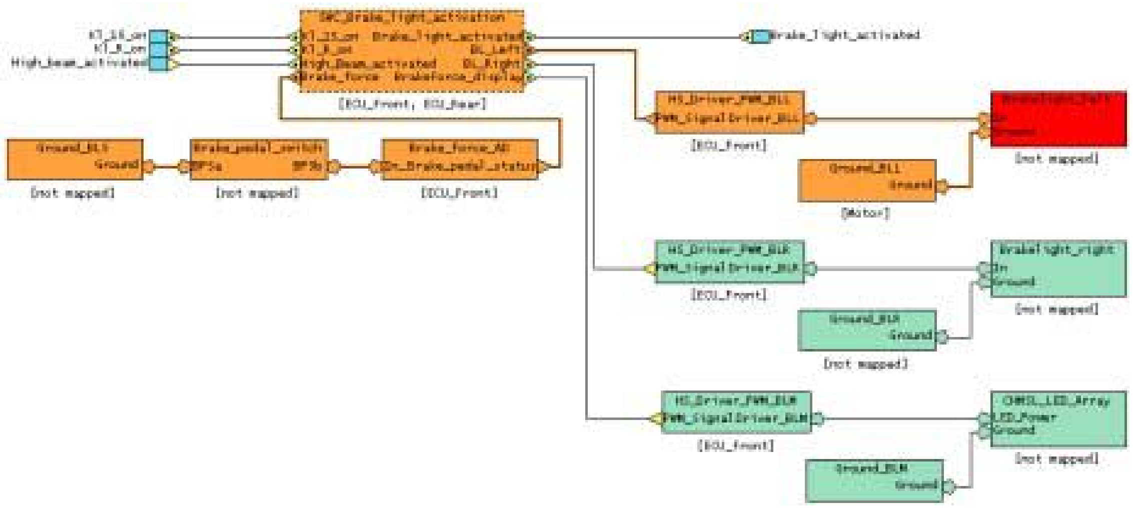


Figure 2.20: Fault back tracking

Fault back tracking - model based diagnosis and remote debugging Figure 2.20 depicts a fault back tracking analysis. Any logical element, sensor, actuator, HW-channel, SW-module, ECU, bus cable or signal can be marked faulty. It analyses the possible causes for a fault and displays the result of the analysis graphically across all hierarchies and views. The picture on the right shows an analysis, where a bulb has been marked faulty. The information needed to carry out this analysis (dependencies) can be derived automatically e.g. from existing Simulink models. It can upload actual process values from a plant e.g. via TCP/IP and display the values in the functional model. If the embedded system supports forcing, the user can force values into the plant directly out of his logical architecture model view.

2.7 Summary

Aspects like structured and unstructured text, semi formal approach for software reuse, component mining, software product line engineering and variability that form the basis and are crucial in the specification and development of heterogeneous cross-domain simulation models were discussed in this chapter. These aspects have a deterministic role in the classification of commonality for variability identification. The later section cast an insight of a modeling platform that is used in the case study.

Chapter 3

Concept

A Model Driven Development approach for system development allows heterogeneity in the target platform with the state-of-the-art code generation for the executable artifacts. This approach lacks support for reconfiguration of these artifacts because the implementation is bound to a target platform at the development time.

These models contain an abstract description of the software specification and therefore foster reuse of the described software artifacts on multiple platforms. Hence it is often necessary to define a mechanism to identify reusable components within the hierarchical models with numerous composite components deeply embedded.

Models confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications, and AUTOSAR® were considered, although this concept is not limited to the automotive domain alone.

We start by analyzing the deficiencies of the tool to handle variants, the project structure and the textual representation of the model structure. Furthermore, we state the specification of cases and form a concept to define commonalities to extract an element list that facilitate the identification of variability. Based on the adaptation of a formal mathematical model presented in this chapter is the implementation and evaluation of the proposed strategy.

3.1 Definition of the problem

Referring to the description in Section 2.6, as the depth of hierarchy in FSB and FTB grows, the probability for redesigning similar functions and user-defined types, which may be variants of existing types increases, simply because the mechanism to manage variants is lacking.

This tool displays inadequacy in handling variants. Therefore the need arises to develop patterns (parameters and procedures) for extracting object features, and to develop a mechanism to manage and support variability.

3.1.1 SPLE approach

To achieve a variability management mechanism we apply the SPLE approach.

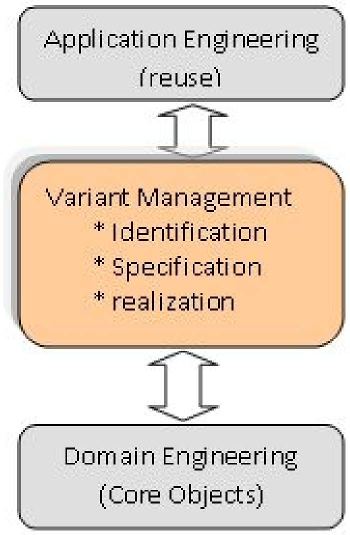


Figure 3.1: Variability managment layer in product lines

Figure 3.1 illustrates a basic SPLE mechanism that can be categorized into domain engineering and application engineering. Domain engineering involves design, analysis, and implementation of core objects, whereas application engineering is reusing these objects for product development[1].

A layer for variant management is introduced in between to achieve variability handling. Activities on the variant management process involves variability identification, variability specification, and variability realization[2].

- A variability identification process will incorporate feature extraction and feature modeling.
- A variability specification process is to derive a pattern.
- A variability realization process is a mechanism to allow variability.

3.1.2 Analysis of project structure

Figure 3.2 illustrates an example to demonstrate the development phase for two products A and B. Both products reuse functions from a data backbone. The development phase

[1] [Bachmann and Clements, 2005], [Bosch, 2000]
[2] [Burgareli et al., 2009]

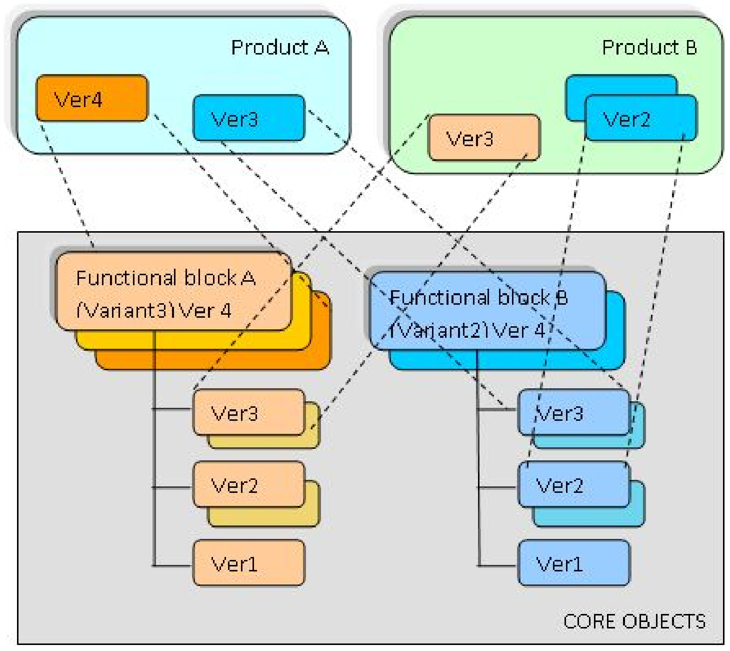

Product A
Ver4
Ver3
Product B
Ver3
Ver2
Functional block A (Variant3) Ver 4
Functional block B (Variant2) Ver 4
Ver3
Ver2
Ver1
Ver3
Ver2
Ver1
CORE OBJECTS

Figure 3.2: Example describing Variants, Versions and Products

may also include the development of software functions in the repository itself, which may have numerous versions. The software development for product A is tested with the Ver3 of functional block B and a variant of Ver4 of functional block A. Similarly product B also uses different version and variants of functional blocks from the data backbone.

A certain version/release of a variant of a functional block suitable in a product may exhibit an improper behavior, when a different version or variant is selected. Even a more improved version of the same variant may not yield a reliable result.

When handling variants and versions we can state the following specification of cases:

Case 1: The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks, whilst using later version of the function may fail to perform as expected.

Case 2: To enable parallel development, it is necessary to be able to extract features and to identify and specify the functional blocks in the repository based on architecture and functionality.

Case 3: A process that tracks usability and prevents inconsistencies due to deprecate variants and version from repository is required.

Case 4: A testing mechanism for validations in order to maintain high quality for components and its variants has to be established.

Case 5: The developer has to be assisted by a process to intelligently determine whether a functional block or its variant should exist in the data backbone, to avoid redesigning of existing functions, thereby improving productivity.

3.1.3 Extending variability management layer

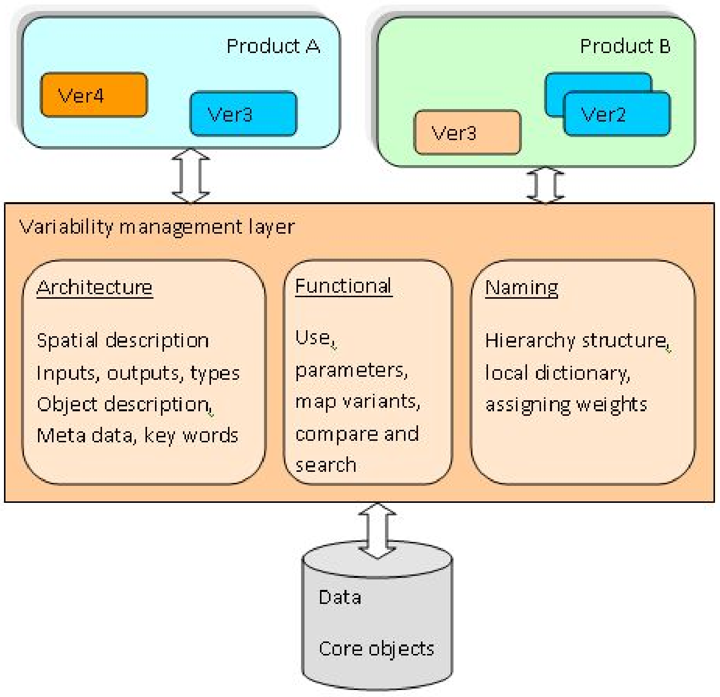


Figure 3.3: Variability management layer

Extending the variability management layer described in Section 3.1.1. Decision points, choices and constraint dependency rules describe variability. In the definition of variability, we have both variability subjects and variability objects. Variability subject may be understood as decision points, and variability objects as choices. Typically several thousand decision points and choices are required. Constraint dependency rules are: requires or excludes decision points, requires or excludes choices, and choice requires or excludes decision points[3].

Based on the specification of the cases broadly the decision points, choices, and constraint dependency rules pattern for feature extraction to extract spatial, functional and name are depicted in Figure 3.3.

The proposed method is to introduce a layer that provides the capability for variability management as shown in Figure 3.3, which enhances both readability and clarity in representation of variability. It offers the user an option for the configuration of all information

[3] [Osman et al., 2010]

related to variation points, insertion of variants, the definition of their types, and their storage in a data base. All that information is made available so that the user can select and create new variants.

A. Architectural features: Variability may only be in some spatial features of software functions like a different parameter or a different data type for inputs or outputs, etc. Thus in an interactive tool one of the techniques to identify software functions can be based on the architectural features.
To enhance accessibility the architectural features supported are:

- Structural design rules, design convention, consistent ways: these could include the number of inputs and outputs, their data types, and their default values,
- Parts in different sub trees which could be similar: a group of blocks having same functionality,
- Brief description about the object, comments, description convention: meta data, keywords and other textual description about the object.

B. Functionality of the objects: Though the spatial features may provide a certain amount of accessibility, there can be numerous functions with different functionality having the same or similar spatial features. Thus identifying software functions based on spatial features alone is not sufficient and therefore identifying functionality becomes essential. Also it can further narrow down the search. Defining architectural features is a relatively simpler process than defining functionality. Describing functionality is an extremely complex process with interrelation between numerous parameters, logic tables, data types, states, etc.

To further narrow down the accessibility, functionality features supported are:

- Methods to identify identical blocks: based on functional use,
- Representing hierarchy in structure of variants: based on parameters,
- Possibility to map variants,
- Comparing and searching: rules for comparing functionality and search for the functional block.

C. Naming conventions: Naming conventions of functional blocks, parameters, etc., are usually long and cumbersome, which are not just difficult to remember but also tedious to construct. Eg. a typical convention for naming defined as: project_group_year_functionalBlock_type_number looks like

BRGD_CODCA_2010_FNRS_USINT_14357

BRGD_CODCA_2010_FNRS_USINT_14486

BRGD_CODCA_2010_FNRS_USINT_14527

These names are neither user friendly nor meaningful. But to assist the user to identify, search, and construct these names, comfortably displaying them as hierarchy, as well as having a procedure to navigate and simplify the construction of such names will enable the user to quickly build long names uniformly over the entire project. Often used activities that are supported by our solution are:

- Building a local dictionary of all names used in the project, if they are not the standard words,
- Using some characters as delimiters and displaying suggestions to the next level in the hierarchy from the existing names, as the user types new names or edits existing ones. In addition suggestions of thesaurus, synonyms, antonyms, etc. are given,
- Automatically navigate through hierarchy levels as the corresponding suggestions are selected, or build the names as one traverses through the hierarchy,
- Assigning weights depending on related names, most often used names, frequently used names, etc.

3.1.4 Textual and graphical representation

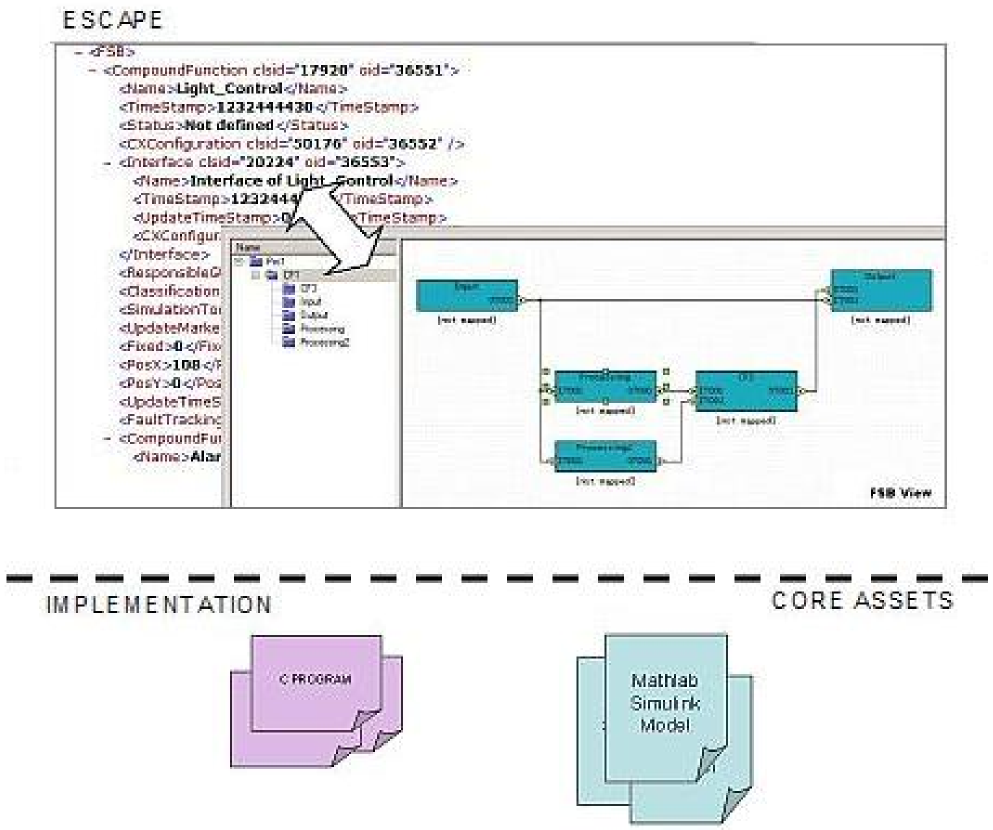
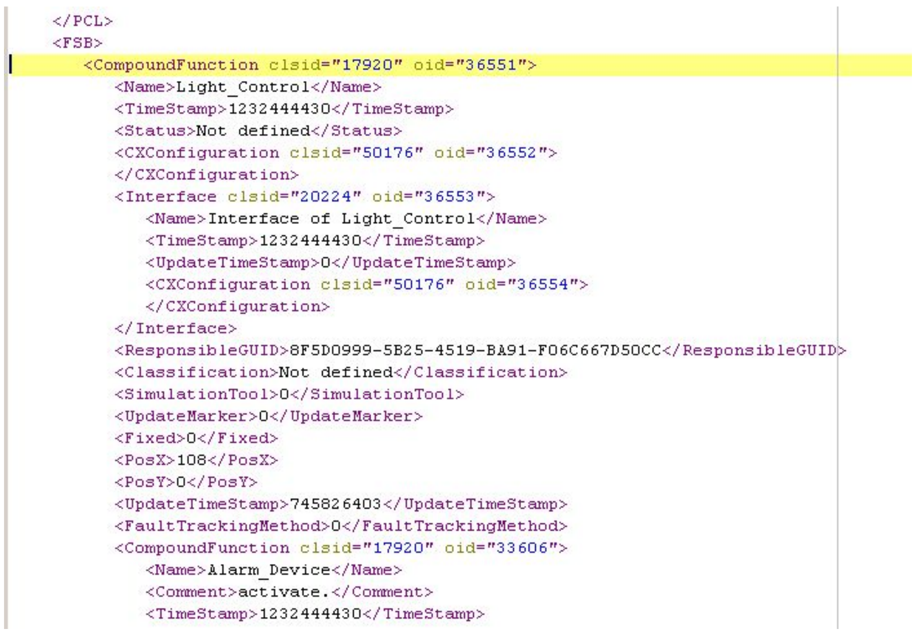

```
- <FSB>
  - <CompoundFunction clsid="17920" oid="36551">
      <Name>Light_Control</Name>
      <TimeStamp>1232444430</TimeStamp>
      <Status>Not defined</Status>
      <CXConfiguration clsid="50176" oid="36552" />
    - <Interface clsid="20224" oid="36553">
        <Name>Interface of Light_Control</Name>
        <TimeStamp>1232444</TimeStamp>
        <UpdateTimeStamp>0</TimeStamp>
        <CXConfigur
      </Interface>
      <ResponsibleG
      <Classification
      <SimulationTo
      <UpdateMarke
      <Fixed>0</Fix
      <PosX>108</P
      <PosY>0</Pos
      <UpdateTimeS
      <FaultTrackinc
    - <CompoundFu
        <Name>Alar
```

Figure 3.4: Mapping textual and graphical representation

An analysis of the models exhibits a common architecture. Figure 3.4 depicts the textual representation that underlies every graphical models. The textual representation is usually in XML format which strictly validates to a schema. A heterogeneous modeling environment may consist of numerous design tools each with its own unique schema to offer integrity and avoid inconsistencies. Projects developed have to strictly validate to the schemas of these tools respectively.

3.1.5 Identifying commonalities in textual representations



```
</PCL>
<FSB>
    <CompoundFunction clsid="17920" oid="36551">
        <Name>Light_Control</Name>
        <TimeStamp>1232444430</TimeStamp>
        <Status>Not defined</Status>
        <CXConfiguration clsid="50176" oid="36552">
        </CXConfiguration>
        <Interface clsid="20224" oid="36553">
            <Name>Interface of Light_Control</Name>
            <TimeStamp>1232444430</TimeStamp>
            <UpdateTimeStamp>0</UpdateTimeStamp>
            <CXConfiguration clsid="50176" oid="36554">
            </CXConfiguration>
        </Interface>
        <ResponsibleGUID>8F5D0999-5B25-4519-BA91-F06C667D50CC</ResponsibleGUID>
        <Classification>Not defined</Classification>
        <SimulationTool>0</SimulationTool>
        <UpdateMarker>0</UpdateMarker>
        <Fixed>0</Fixed>
        <PosX>108</PosX>
        <PosY>0</PosY>
        <UpdateTimeStamp>745826403</UpdateTimeStamp>
        <FaultTrackingMethod>0</FaultTrackingMethod>
        <CompoundFunction clsid="17920" oid="33606">
            <Name>Alarm_Device</Name>
            <Comment>activate.</Comment>
            <TimeStamp>1232444430</TimeStamp>
```

Figure 3.5: Textual representation of projects in XML

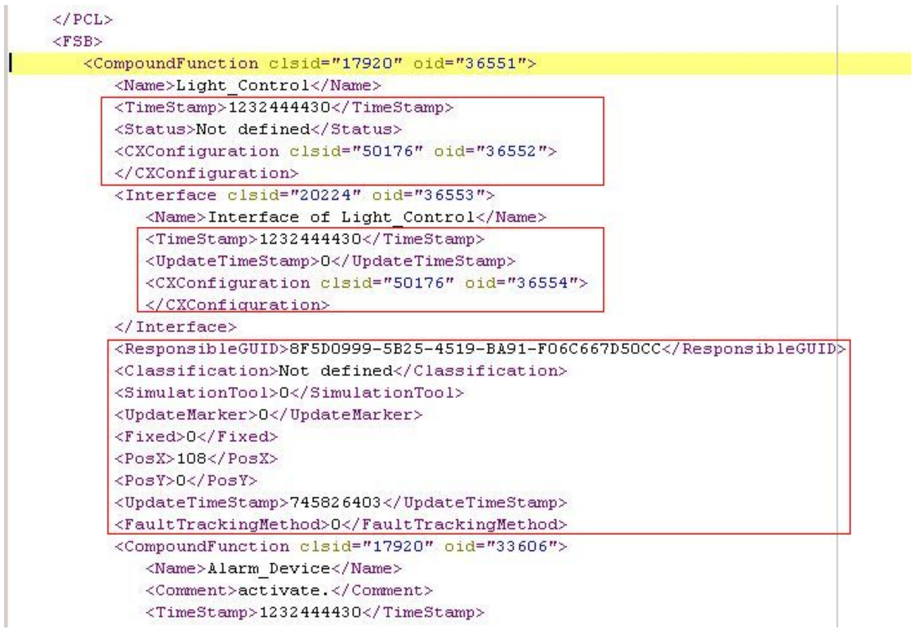
Figure 3.5 illustrates a the textual representation of models mainly in XML format.

A closure examination of the nodes in the textual representation of models depicted in Figure 3.6 reveals some interesting information. The nodes marked in red rectangles provide important information regarding the identity, specification, physical attributes, etc. of a component, but are insignificant from the variant perspective.

Based on the challenges discussed and the concluded related work presented, the following objectives for this thesis can be derived.

3.2 Objectives of this thesis

Objective 1: Support heterogeneous models containing hierarchically embedded software components containing the complete specification of specific functionality to foster reuse.



```
</PCL>
<FSB>
  <CompoundFunction clsid="17920" oid="36551">
    <Name>Light_Control</Name>
    <TimeStamp>1232444430</TimeStamp>
    <Status>Not defined</Status>
    <CXConfiguration clsid="50176" oid="36552">
    </CXConfiguration>
    <Interface clsid="20224" oid="36553">
      <Name>Interface of Light_Control</Name>
      <TimeStamp>1232444430</TimeStamp>
      <UpdateTimeStamp>0</UpdateTimeStamp>
      <CXConfiguration clsid="50176" oid="36554">
      </CXConfiguration>
    </Interface>
    <ResponsibleGUID>8F5D0999-5B25-4519-BA91-F06C667D50CC</ResponsibleGUID>
    <Classification>Not defined</Classification>
    <SimulationTool>0</SimulationTool>
    <UpdateMarker>0</UpdateMarker>
    <Fixed>0</Fixed>
    <PosX>108</PosX>
    <PosY>0</PosY>
    <UpdateTimeStamp>745826403</UpdateTimeStamp>
    <FaultTrackingMethod>0</FaultTrackingMethod>
    <CompoundFunction clsid="17920" oid="33606">
      <Name>Alarm_Device</Name>
      <Comment>activate.</Comment>
      <TimeStamp>1232444430</TimeStamp>
```

Figure 3.6: XML nodes which do not signify variability

Breaking down the models into several components and logical clustering of components of the modeled software is not targeted. In contrast the proposed methodology enables the identification of commonalities of components in heterogeneous models. For deployment and reuse purposes several partial models are treated as one artifact. Furthermore the architecture should support reuse of these artifacts for the development of new functionalities.

The challenge of the realized system of artifact heterogeneity should be based on existing component technologies that provides mature techniques, that are a consequence of the application independent and generic definition of the system specific components and ensures the portability of the proposed system on other platforms.

Objective 2: Enable dynamic configuration.

To envisage each subsystem is modeled and simulated using a domain-specific simulation tool, while the co-simulation platform handles the coupling between these subsystems that enables holistic simulation of a system.

The challenge for identifying variability of software components validating to numerous schemata of respective simulation tools and dynamically loading of plug-ins for specific set of components adhering to respective schemata at execution time in model interpretation architecture.

Objective 3: Enable shared usage of resources.

A scenario depicting the concept of virtual organization should have a clear method to tackle resource access, validation and verification of specific models.

3.3 Contributions of this thesis

The following contributions are claimed as outcomes of the research in this thesis regarding the objectives presented in the previous section .All contributions have been evaluated by prototypes during the case studies summarized in Chapter 5 as well as in the specific papers presented in Chapter 7.

Contribution 1: Model-based Variability Management for Complex Embedded Networks.

The concept of Model-based Variability Management is proposed in the publication available in Section 7.1, which contemplates on the definition of a problem and specification of the cases. Furthermore the concept specified is used for feature extraction to extract spatial, functional, and name for the realization of new functionality. These models has been evaluated for data models in the publication in Section 7.4.

Contribution 2: A generic approach to envisage the identification of variability.

The primary mechanism for determining commonality, allowing dynamic extension in the identification of variability of software components which are embedded in hierarchical model confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications, and AUTOSAR®. The approach is based on the adaption of a formal mathematical model presented in the publication in Section 7.3

Contribution 3: An approach to visualize, navigate and simplify the unintelligible naming conventions.

Mapping highly indecipherable naming conventions and transposing to hierarchical structures using predetermined delimiters, to assist the user to identify, search, and construct these names, comfortably displaying them as hierarchy, as well as having a procedure to navigate and simplify the construction of such names.

3.4 Overview of the process

We propose a semi-automatic variant identification layer. A component list and feature vector is derived manually from the schema of the project; a collection of elements that represent components and their descriptive features that significantly contribute to the identification of the component variant respectively. For projects developed using several modeling tools and simulation tools numerous lists can be derived for each distinct schema.

The basic concept to identify variability is depicted in Figure 3.7. The left side is a set of projects which have software components hierarchically embedded within. This projects validate to the corresponding schema. The middle layer is an identification layer with three functional blocks. A set of component list is derived from the node list in this schema. Similarly a feature vector is derived from the schema that corresponds to components. The second block is a customized parser that generates a relevant lexicon from the set of software components within a project. The third block is a set of rules (viz. mandatory, optional, exclude) to govern variability identification.

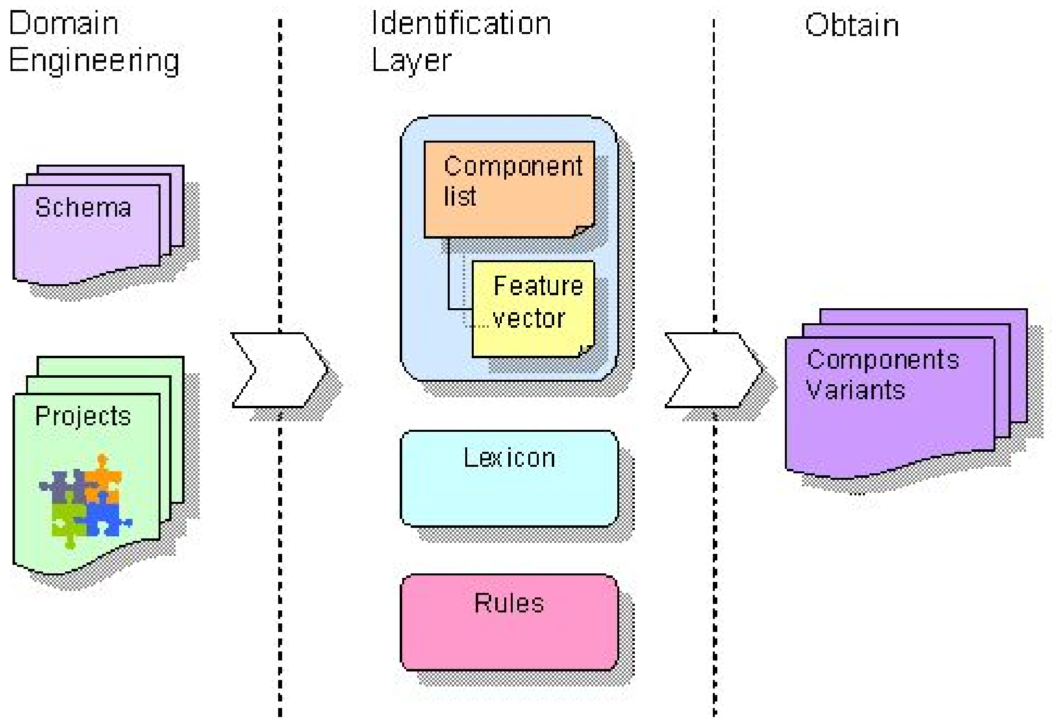


Figure 3.7: Basic concept

3.4.1 Implementation for homogeneous systems

The higher level model is a collection of a hierarchical structure of sub-models. In a distributed business process models are developed and simulated at functional level. Simulating distinct models is not enough as models are mostly interdependent. It necessitates cross-domain modeling and simulation.

The basic concept can be extended to obtain a working model for identification. The work flow is depicted in Figure 3.8. The top layer represents the domain or core assets.

The middle layer is a semi-automatic variant identification layer. A component list and feature vector is derived manually from the schema of the project; a collection of elements that represent components and their descriptive features that significantly contribute to the identification of the component variant respectively. A lexicon is generated fully automatic by processing the elements within the project files that match the component-feature list along with weights based on the frequency of the words. The rules govern the process of variant identification, when the model is explored in the application layer and returns a set of components which may be variants, have semblance or are in some way associated to the expected component.

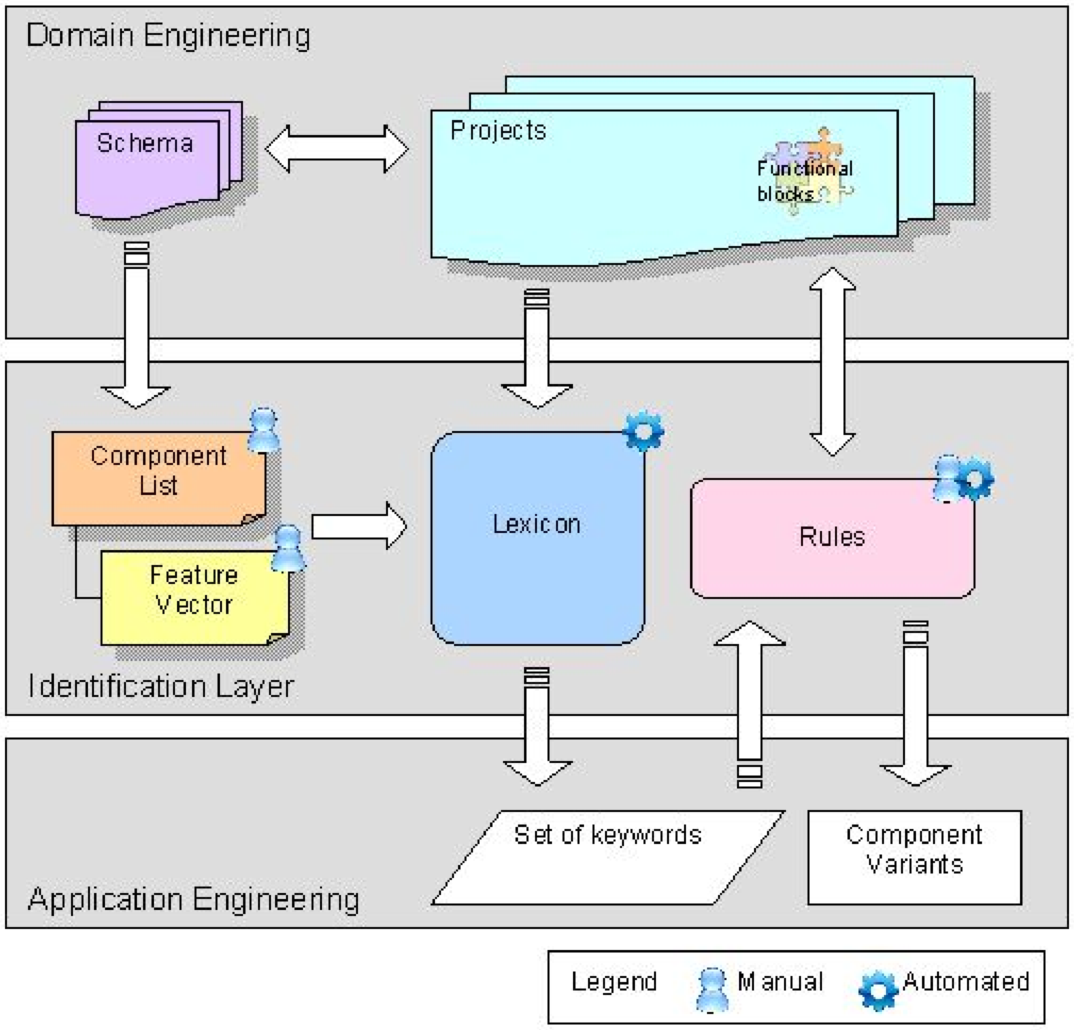


Figure 3.8: Concept extended to homogeneous systems

3.4.2 Concept extended to heterogeneous systems

The work-flow of the concept can be further extended to adapt a heterogeneous environment which consist of projects developed using several modeling tools and simulation tools.

Numerous projects developed using different modeling tools and simulation tools having different schemas to illustrate the components, features and other information. In such scenario the identification layer may be partitioned as illustrated in Figure 3.9. A separate component list and feature vector is derived from each distinct schema.

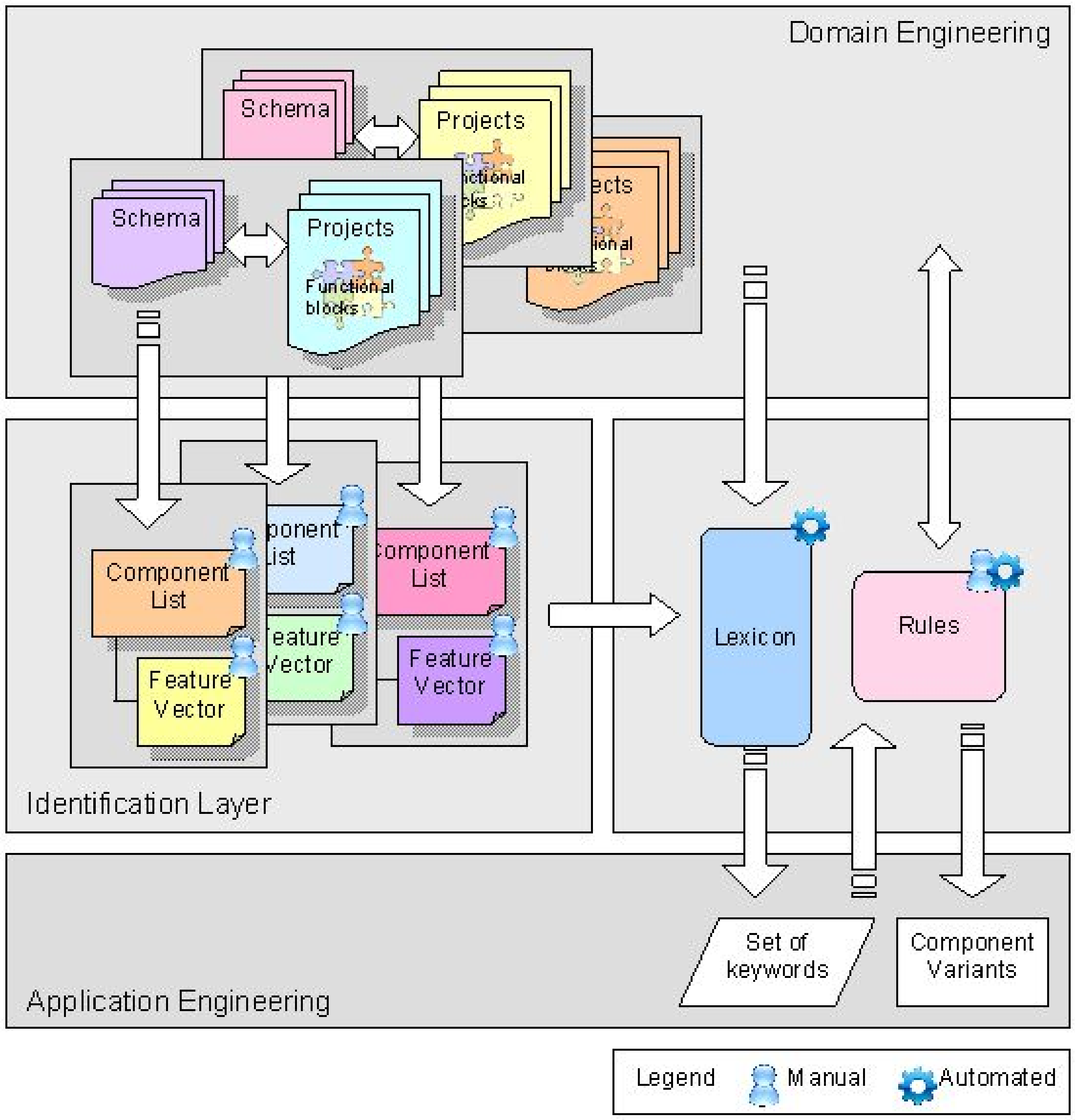


Figure 3.9: Concept extended to heterogeneous systems

3.5 Mathematical model

The formal representation of such a model is complex. The software model is composed of a set of functions, which further contain sub-functions and so exhibiting a hierarchical structure. The software models can be defined as

$$P = \{E, \Gamma\} \tag{3.1}$$

$P = \{p_1, p_2, ...p_n\}$ is a finite set of models consisting of elements that form the functional modeling (the abstract specification of the components), solution modeling (the implementation of the components), and architecture design (deploying and mapping these components on different platforms). In addition it also contains elements that are general rationale and do not signify any of these functionalities.

$E = \{e_1, e_2, ...e_m\}$ is a finite set of elements that constitutes elements providing general information (viz., id, time stamp, date, owner, etc.), elements that form components, elements within the components that represent features. Some of these elements may be categorized as elements that describe variability or that contribute to signify variants.

$\Gamma = \{\gamma_1, \gamma_2, ...\gamma_o\}$ is a finite set of elements which describes complex relationships that reflect information relationships, inheritance flow, and message exchanges.

Each of these models validate to a schema; and there is an isomorphic mapping relationship between the elements of the schema and the models.

We define a schema S as a set of formulas that specify integrity and constraints

$$S = \{N, C\} \tag{3.2}$$

The schema defines the structure, entities, attributes, relationships, views, indexes, packages, procedures, triggers, types, sequences, synonyms and other elements.

$N = \{n_1, n_2, ...n_k\}$ denotes a finite set of nodes or elements in a schema that describes integrity, whereas $C = \{c_1, c_2, ...c_j\}$ denotes a finite set of elements in a schema that describes constraints, and further to adapt a heterogeneous environment which consists of projects developed using several modeling and simulation tools.

$S = \{s_1, s_2, ...s_i\}$ is a finite set of schemata each representing a modeling or simulation tools.

At user reconfiguration level, the software model is represented in an abstract form, consisting of modules, functions, relationship, information, inherited flow, and message flow. Subdividing the set of nodes N and the set of constraints C into general elements and elements that signify

$$N = \{n, \eta\} \tag{3.3}$$

$$C = \{c, \upsilon\} \tag{3.4}$$

$\eta = \{\eta_1, \eta_2, ...\eta_p\}$ and $\upsilon = \{\upsilon_1, \upsilon_2, ...\upsilon_q\}$ are a finite set of nodes and constraints respectively that signify components, features, functions, relations, whereas, $n = \{n_1, n_2, ...n_r\}$ and $c = \{c_1, c_2, ...c_s\}$ are a finite set of nodes and constraints respectively that signify all other nodes.

Targeting all nodes in the model that are isomorphically mapped to η and υ leads to a set of nodes that can be viewed as a Significant Node (SN). As the functions are hierarchical the software model may be viewed as a Significant Node Mesh (SNM).

SN can be defined as

$$SN = \{C_m, F_c, N_c, R\} \tag{3.5}$$

where $C_m = \{C_{m1}, C_{m2}, ...C_{mn}\}$ is a finite set of all components defined on the set P, $\forall\ C_{mi} \subset C_m$ and $i = 1, ...m$, C_{mi} is a finite set including all components of p_i, and is a subset of C_m. $F_c = \{F_{c1}, F_{c2}, ...F_{co}\}$ is a finite set of all features defined on the set P, $\forall\ F_{cj} \subset F_c$ and $j = 1, ...o$, F_{cj} is a finite set including all features of p_i, and is a subset of F_c. N_c and R denotes the set of naming conventions and the set of relations respectively.

Let S_N denote the nodes in model P and M denotes the nodes in schema S. Then there is a map (function) τ from S_N into M, defined such that $\tau(n)$ is the definition (or rule) of $n \in S_N$ in M.

$$\tau : S_N \rightarrow M \tag{3.6}$$

Let S_c be an element of S representing a component c. Let E_C be the subset of the schema S which is extracted manually such that each element represents a variant component.

$$E_C = \{S_c \in S : c \text{ represent a component}\} \tag{3.7}$$

Let E_F be the subset of a S which is extracted manually such that each element represents a feature of the component c.

$$\begin{aligned} E_F = \{E_{F_c} \in S : E_{F_c} &\text{ represents a feature} \\ &\text{of the component } c\} \end{aligned} \tag{3.8}$$

$E_F(i, c)$ denotes the i^{th} element of E_F of a component c.

Let C_1 be the subset of C such that all elements of C_1 are represented in E_C.

$$C_1 = \{c \in C : \tau(c) \in E_C\} \tag{3.9}$$

Let F'_c be the subset of F_c such that element of F'_c are represented in E_F.

$$F'_c = \{f \in F_c : \tau(f) \in E_F\} \tag{3.10}$$

Let $F'(i, c)$ be the i^{th} element of F'_c, where i is an integer.

Let V be the specification set. Then the proposed method will return a result set R_{sn}

$$R_{sn} = \left[\bigcup_{c \in C_1} \left(c\left(\bigcup_{i} F'(i, c)\right)\right)\right] \bigcap V \tag{3.11}$$

and the absolute number of elements in the resultant set R_{sn} is

$$|R_{sn}| = \left| \left[\bigcup_{c \in C_1} \left(c \left(\bigcup_i F'(i,c) \right) \right) \right] \bigcap V \right| \tag{3.12}$$

On the other hand, the result set R_g obtained by global search is

$$|R_g| = |N \cap V| \tag{3.13}$$

where N is the set of nodes in the project.

The general search depicted by Equation 3.13 is a set that contains all nodes which match the specification set irrespective of whether they exhibit relevance to the components or not, resulting in a large set. On the other hand the approach of selective targeting of significant nodes will match only with the nodes that are components and the features that describe variability, thereby returning a closely matched result set depicted by Equation 3.12.

Comparing Equation 3.13 and Equation 3.12

$$|N \cap V| \geq \left| \left[\bigcup_{c \in C_1} \left(c \left(\bigcup_i F'(i,c) \right) \right) \right] \bigcap V \right| \tag{3.14}$$

Hence we conclude an improved result set using the proposed approach as depicted by Equation 3.14.

3.6 Summary

The approach presented peeks into the structure of the models and the specification of cases are stated based on the reasoning. The objectives for this thesis are identified from the challenges discussed and the contributions of the research are enumerated.

Further an overview of the process is presented, which is extended to homogeneous and heterogeneous systems.

An adaptation of a mathematical model that describes the formal approach of the process is formed to envisage the problem and prove theoretically the proposed strategy.

The proposed approach has been evaluated on an industry use case project model developed using the design tool ESCAPE built by the company Gigatroniks.

Chapter 4

Implementation

In this chapter several key aspects of the implementation are discussed. The focus is to describe the architecture of the identification layer, which forms the intermediate layer for adapting the core assets from domain engineering into application engineering.

The related approaches put on view a need for a generic methodology in identification of software components developed using several design tools.

4.1 Component list and feature vectors

As the project structure for each tool is well defined and strictly validate with corresponding schemata, these schemata can be the basis for deriving the list that can identify components.

4.1.1 Component list

An example of the list of elements that characterize components derived manually from the schemata for design tool ESCAPE is

> "CompoundFunction HWFunction SWFunction Parameter" + " StructureElement SWBubbleType ParameterType ParameterTypeTerminal IntDataType FloatDataType TimeDataType AliasDataType VariantDataType HWFunctionType TypeInterface FunctionTypeTerminal HWTypeTerminal" + " StructureElement DeviceMapping DeviceType BusCAN BusSegment MappedFunction"

The list is a delimited string with space or any other delimiter.

A tool supporting multi-functional structures like ESCAPE which has three views FSB, FTB and HSB. Each view can have an independent list

- Component list for FSB

  "CompoundFunction HWFunction SWFunction Parameter"

- Component list for FTB

  "StructureElement SWBubbleType ParameterType ParameterTypeTerminal IntDataType FloatDataType TimeDataType AliasDataType VariantDataType HWFunctionType TypeInterface FunctionTypeTerminal HWTypeTerminal"

- Component list for HSB

  "StructureElement DeviceMapping DeviceType BusCAN BusSegment MappedFunction"

Similarly, in a heterogeneous modeling environment each modeling tool will have its own schemata, and a corresponding list may be derived for each tool.

4.1.2 Feature vector

Similarly, the elements that characterize features of the software components are also derived manually from the schemata, which forms the feature vector and are enlisted below

> "Name LongName DEScription ConnectionSegment SourceTerminal SinkTerminal Interface CompoundTerminal HWTerminal SWTerminal Input DataType"

4.1.3 Naming convention

Moreover, the naming convention within an organization also lead to ambiguity in identification of components when the number is large.

A list for a naming convention for distributed business process is illustrated below

> "WorkSpace DOMain GRouP PRoJect FunctionBlock PartNo VARiant"

A sample of Names confirming to AUTOSAR in an organizational setup is illustrated below (text in bold are the names specified in AUTOSAR), There are more then 10,000 such definitions in AUTOSAR[1].

> FV_BodyAndComfort_Access_P7834_**ATWSLHFD**_AS45781.0003,
> Description: Anti Theft Warning System, Left hand front Door
>
> FV_BodyAndComfort_Access_P7834_**ATWSRHFD**_AS45782.0003,
> Description: Anti Theft Warning System, Right hand front Door
>
> FV_BodyAndComfort_Access_P7834_**ATWSLHRD**_AS45783.0004,
> Description: Anti Theft Warning System, Left hand Rear Door

[1] [Autosar, 2011]

FV_BodyAndComfort_Access_P7834_**ATWSRHRD**_AS45784.0004,
Description: Anti Theft Warning System, Right hand Rear Door

FV_BodyAndComfort_Access_P7838_**CLDLRB**_CL45785.0002,
Description: Central Locking, Door Lock Rearlid and Backlite

FV_BodyAndComfort_Access_P7838_**CLDLRB**_CL45785.0003,
Description: Central Locking, Door Lock Rearlid and Backlite

FV_BodyAndComfort_Access_P7838_**CLKPM**_CL45786.0002,
Description: Central Locking, KeyPad Manager

FV_BodyAndComfort_Access_P7838_**CLTF**_CL45787.0001,
Description: Central Locking, Tank Flap

FV_BodyAndComfort_Visibility_P7834_**ELBLM**_BL45788.0003,
Description: Exterior Lights, Brake Light manager

FV_Chassis_**ESC**_P7847_**ABSEBD**_ESC45792.0004,
Description: ElectronicStabilityControl, Antilock Braking System, Electronic Brake force Distribution

FV_Chassis_**ESC**_P7847_**ABSCBC**_ESC45793.0003,
Description: ElectronicStabilityControl, Antilock Braking System, Cornering Brake Control

FV_Chassis_**ESC**_P7847_**TCS**_ESC45794.0002,
Description: ElectronicStabilityControl, Traction Control System

FV_Chassis_**ESC**_P7834_**YRC**_ESC45795.0003,
Description: ElectronicStabilityControl, Yaw Rate Control

FV_Chassis_**ESC**_P7842_**BAS**_ESC45797.0003,
Description: ElectronicStabilityControl, Brake Assist

FV_Chassis_**Susp**_P7844_**BRHC**_SUSP45801.0004,
Description: Suspension, Body Ride Height Control

FV_Chassis_**Susp**_P7844_**RCWFC**_SUSP45804.0004,
Description: Suspension, Ride Control and Wheel force Control

If the names in the above sample are transformed to a hierarchical structure by splitting with "_" as delimiter, they are more perceptible.

FV
    _BodyAndComfort
        _Access
            _P7834
                _ATWSLHFD
                    _AS45781.0003,

```
                _ATWSRHFD
                    _AS45782.0003,
                _ATWSLHRD
                    _AS45783.0004,
                _ATWSRHRD
                    _AS45784.0004,
            _P7838
                _CLDLRB
                    _CL45785.0002,
                    _CL45785.0003,
                _CLKPM
                    _CL45786.0002,
                _CLTF
                    _CL45787.0001,
        _Visibility
            _P7834
                _ELBLM
                    _BL45788.0003,
    _Chassis
        _ESC
            _P7847
                _ABSEBD
                    _ESC45792.0004,
                _ABSCBC
                    _ESC45793.0003,
                _TCS
                    _ESC45794.0002,
            _P7834
                _YRC
                    _ESC45795.0003,
            _P7842
                _BAS
                    _ESC45797.0003,
        _Susp
            _P7844
                _BRHC
                    _SUSP45801.0004,
                _RCWFC
                    _SUSP45804.0004,
```

4.1.4 Algorithm to identify components within projects

Using the string described in Section 4.1.1 that characterize the software components nodes list within a project, the following algorithm can be devised.

```
componentListString $\leftarrow$ string described in Section 4.1.1;
```

```
**Nodes** $\leftarrow$ doc.GetElementsByTagName("*");
for each **Node** in the **Nodes** (Length($L_n$) $\geq 1$), do
    if **Node.name** in **componentListString**, then
        **componentList** $\leftarrow$ **Node.name**;
```

The order O for matching the software components is 1.

The prototype dataset used for evaluation of this algorithm contained a total of 32909 nodes, of which only 1583 matches were the software components.

Similarly, using the string described in Section 4.1.2 that characterize the features within software components, the following algorithm can be devised.

```
**featureVectorString** $\leftarrow$ string described in Section 4.1.2;
**Nodes** $\leftarrow$ **componentList**;
for each **Node** in the **Nodes** (Length($L_c$) $\geq 1$), do
    if **Node.name** in **featureVectorString**, then
        **featureList** $\leftarrow$ **Node.name**;
```

The order O for determining the corresponding features within the software components is 1.

From the prototype dataset a total of 13353 nodes matches to the feature vector were found.

The results are summarized in Table 5.2 in Section 5.1.

4.2 Lexicon

A simple customized parser has been devised which automatically extracts words from the text within the software components and features that match the component list and feature vector respectively.

```
**lexiconList** $\leftarrow$ NULL;
**Nodes** $\leftarrow$ **componentList** $\cup$ **featureList**;
for each **Node** in the **Nodes** (Length($L_{cf}$) $\geq 1$), do
    wordList $\leftarrow$ split(**Node.innerText**, delimiter) ;
    for each **word** in the **wordList** (Length($L_w$) $\geq 1$), do
```

```
            if **word** not in **lexiconList**, then
                **lexiconList** ← **word**;
                **lexiconList.frequency** ← 1;
            else
                **lexiconList.frequency** ← **lexiconList.frequency**+1;
    End For;
```

A more sophisticated parser that discards non-words will further improve the Lexicon.

The Lexicon assists the user to choose from a set of relevant words along with their frequencies thereby improving user experience.

4.3 Rules

In every case a full match of software components to specification sets is not desired, but in many instances specification sets contain elements that are mandatory (contains all), optional (one or more) and exclude (omit). Providing rules to execute these features enhances performance in the identification process.

```
ruleContainAll ← Specification subset with Contain-all elements ;
ruleOptional ← Specification subset with Optional elements ;
ruleExclude ← Specification subset with Exclude elements ;
Nodes ← componentList ∪ featureList;
for each Node in the Nodes (Length($L_{cf}$) $\geq$ 1), do
    wordList ← split(Node.innerText, delimiter) ;
    for each word in the wordList (Length($L_w$) $\geq$ 1), do
        if word not in ruleExclude, then
            if word in ruleContainAll, then
                variantList ← word;
            elseif word in ruleOptional, then
                variantList ← word;
End For;
```

Using the rules enables to narrow down to a more realistic list of variants that matches the specification set.

4.4 Transforming naming convention

Using the string described in Section 4.1.3 that characterize the naming convention within an organization, The scattered software components can be organized by splitting the names along a delimiter and transformed into a hierarchically structure, the following algorithm can be devised.

```
nameConv ← List described in Section 4.1.3;
SWcompNameList ← doc.readCompName("*");
for each SWcompNameConv in SWcompNameList (Length($L_{nc}$) $\geq$ 1), do
    SWcompNameSplit ← split(SWcompNameConv.name, delimiter) ;
    for each SWcompNamePart in SWcompNameSplit (Length($L_{sn}$) $\geq$ 1), do
        if not exist SWcompNamePart0, then
            RootElementNode ← SWcompNamePart;
        else
            ParentElementNode ← RootElementNode;
            for each SWcompNamePart in ParentElementNode.ChildNodes (Length($L_{cn}$) $\geq$ 1), do
                if not exist SWcompNamePart, then
                    ParentElementNode.addChildNode ← SWcompNamePart;
                else
            ParentElementNode ← ParentElementNode.ChildNodes;
    End For;
```

This algorithm can be further extended to assist the user to identify, search and construct these names comfortably displaying them as hierarchy and a procedure to navigate and to simplify the construction of such names, that will enable the user to quickly build long names uniformly over the entire project.

Chapter 5

Evaluation and case study

A prototype of the architecture presented has been implemented. The implementation was evaluated on different case studies of models in the automotive domain. Case studies targeted the design of model-based software components on an industry use case project model developed using the design tool ESCAPE and a second case study targeted the execution of specific paradigm based on the naming convention of AUTOSAR®.

Two case studies were done for use cases in the development stage of the E/E software models of the automotive domain. One case study was conducted on an industrial prototype, which consisted of an application assisting developer to identify variants. The second case study considered the stakeholders of a Virtual Organization and its external suppliers.

Both case studies were based on the same domain model (containing the automotive specific data structures). The first one realized a workflow for determining variants resembling the specification set. The second case study implemented a workflow to amalgamate the distinct components.

5.1 Variability identification by selective targeting

The first case study was realized in the model development stage of the automotive domain. The tool ESCAPE is used for supporting the various key functions of modeling (authoring tool for graphical definition, manipulation and analysis of functional networks for a wide range of configurations, diagnostics tools like dependency analysis, fault back tracking and support for mathlab/simulink simulation models and various programming language implementations) The hierarchal nature of models camouflages visibility of the deeply embedded software components thereby impeding their reuse. Many component variants remain disregarded and the identification of such components would greatly enhance development.

A traditional method may be restricted to a specific case and not deliver a generic identification. Recent trends should support different stakeholders. For an analysis of these stakeholders different levels of interests are defined.

Schema

Description	**Count**
Total elements collection	171
Components list	23
Features vector	12

Table 5.1: Summary of schema for the data set

Project

Description	**Count**	**Category**
Total elements	32909	all
Components	1583	23
Features within components	13353	12

Table 5.2: Summary of project data set of case studies

Projects: People working in a group are assisted by the same tools for the development of models. All of these projects have same schema and with guaranteed quality of service.

Domains: Groups working with projects of different domains using the same or different development tools, e.g. EAST-ADL (Architecture Description Language), ATESST (Advanced Traffic Efficiency and Safety through Software Technology) AUTOSAR (AUTomotive Open System ARchitecture), etc. Here determining commonality gets more important. Another use case of targeting this level of interest is the developer's knowledge of information.

Virtual Organization: External OEM's supplying software artifacts which are well proven and tested over a period of time. These software IPs can be black boxes and using them can be crucial at the same time it is an important requirement (e.g. for scheduling the deliverables).

The concept supports a two phases strategy. The component list and feature vector are selected manually from respective schemas of projects. These lists form the basis for matching the specification set. Although the model based interpretation was considered, because of the missing basic framework the user interface was realized using a traditional programming approach. Nevertheless the design of the user interface components was standardized by the application of Object-oriented design techniques.

A prototypical application was realized using the C# .Net framework. A dialog for entering specification set related information, thus allowing the input of specific information.

The summary of the schema for the project data set is depicted in Table 5.1 and Table 5.2 respectively. The Components List is a subset of the elements within a schema which describe components. A sample component list is illustrated in Table 5.3a. A similar subset of elements which describes the features within the components forms the Feature Vector illustrated in Table 5.3b is also derived from the schema. Elements which do not contribute to describe variability are ignored.

Component List

Description	**Count**
CompoundFunction	58
HWFunction	182
SWFunction	46
Parameter	6
StructureElement	50
SWBubbleType	130
ParameterType	5
ParameterTypeTerminal	8
IntDataType	14
FloatDataType	2
TimeDataType	1
AliasDataType	1
VariantDataType	4
HWFunctionType	46
TypeInterface	181
FunctionTypeTerminal	580
HWTypeTerminal	91
StructureElement	50
DeviceMapping	10
DeviceType	4
BusCAN	3
BusSegment	3
MappedFunction	108
	1583

(a) Component List

Feature Vector

Description	**Count**
Name	7500
LongName	0
Description	0
ConnectionSegment	537
SourceTerminal	538
SinkTerminal	538
Interface	292
CompoundTerminal	269
HWTerminal	292
SWTerminal	302
Input	1543
DataType	1542
	13353

(b) Feature Vector

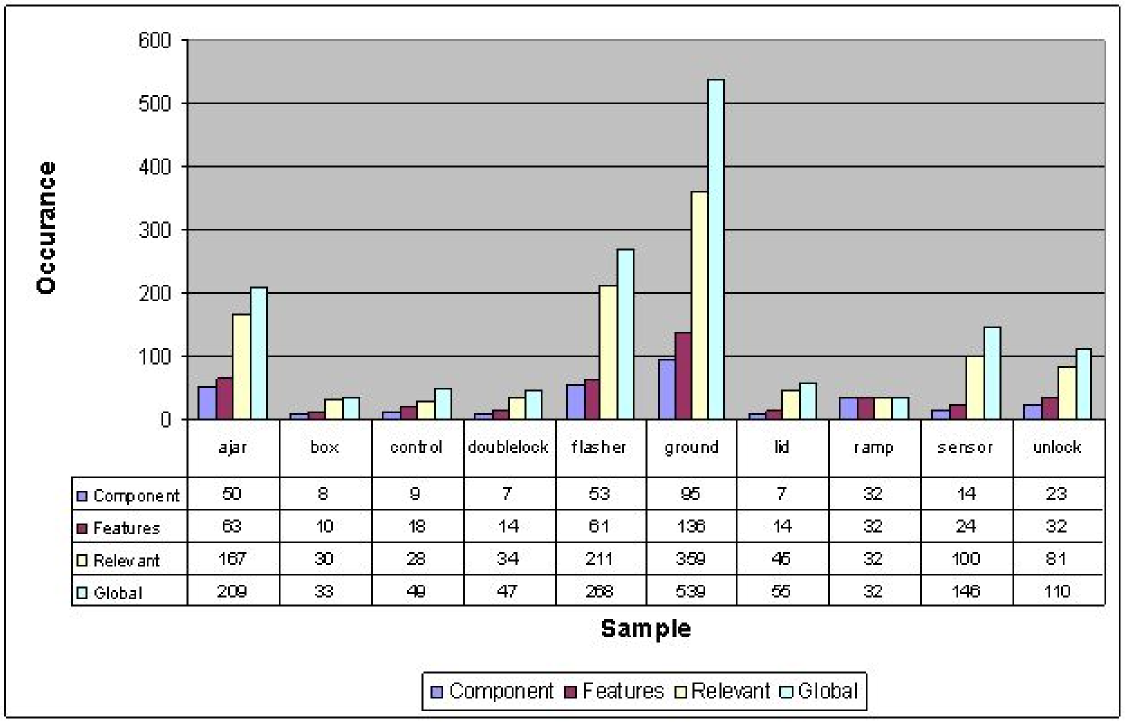
Table 5.3: Component List and Feature Vector derived from Schema

Every Project that validates to the schema refers to the same Component List and Feature Vector derived from the schema to identify components and features.

The specific project data set which was used to verify the implementation is depicted in Table 5.2. The project consisted of a total of 32909 elements. Of these elements a total of 1583 elements signify components which were categorized into 23 categories as enlisted with the component list. A total of 13353 elements signify features which were categorized into 12 categories.

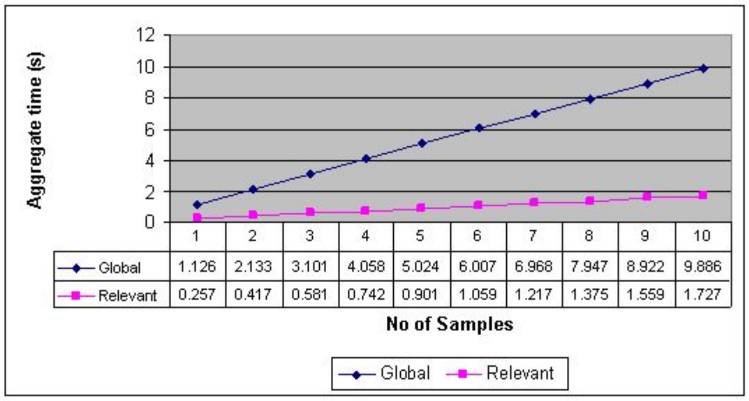
The experimentation has been carried out over a large sample of data, as the results established were similar, only a few are illustrated here.

Three different approaches were adopted to determine the performance with respect to the matches and the time.



	ajar	box	control	doublelock	flasher	ground	lid	ramp	sensor	unlock
Component	50	8	9	7	53	95	7	32	14	23
Features	63	10	18	14	61	136	14	32	24	32
Relevant	167	30	28	34	211	359	45	32	100	81
Global	209	33	49	47	268	539	55	32	146	110

Figure 5.1: Occurrence graph for a single element specification set (Sample set 1)



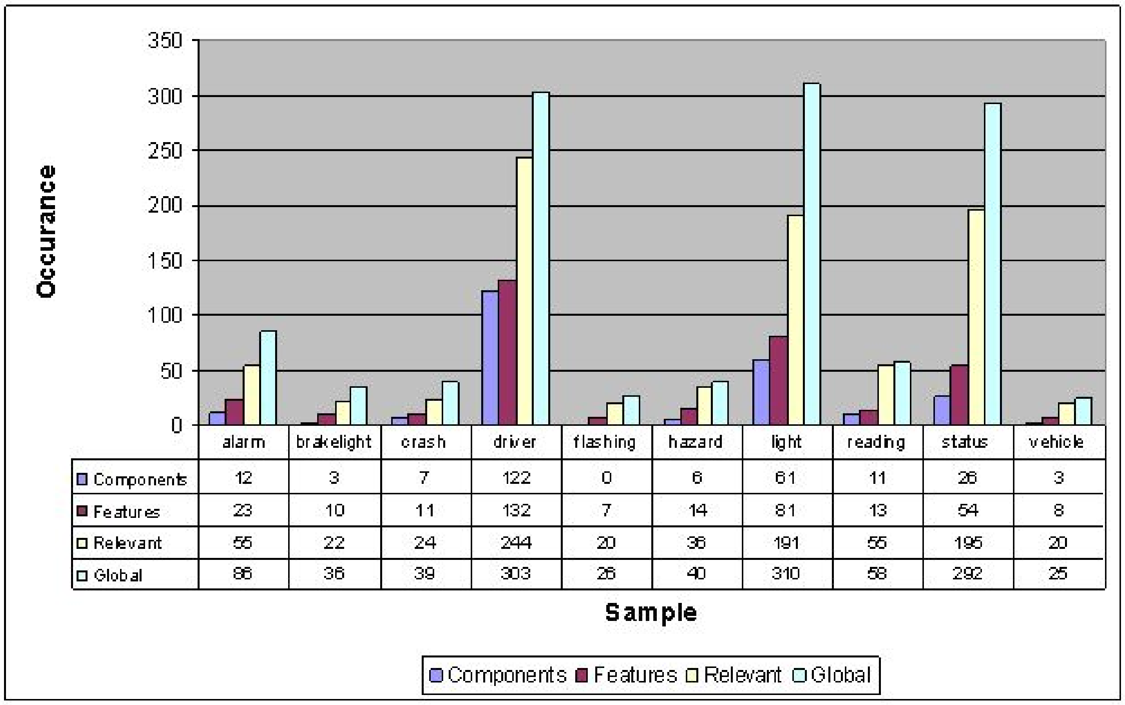
	1	2	3	4	5	6	7	8	9	10
Global	1.126	2.133	3.101	4.058	5.024	6.007	6.968	7.947	8.922	9.886
Relevant	0.257	0.417	0.581	0.742	0.901	1.059	1.217	1.375	1.559	1.727

Figure 5.2: Time graph for a single element specification set (Sample set 1)

5.1.1 Evaluation using a single element specification set

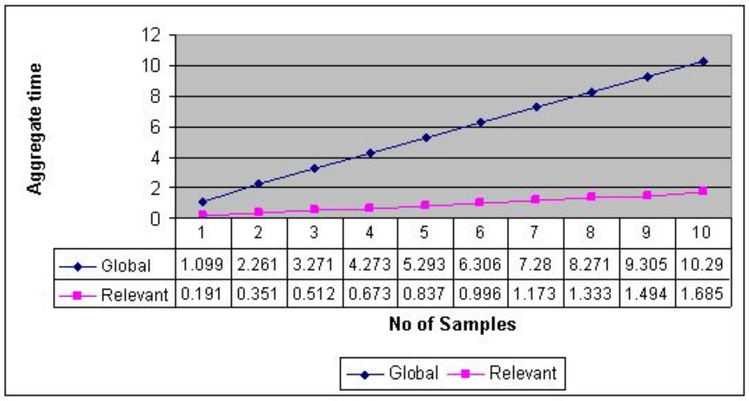
The first experiment was conducted using a single element specification set. A group of ten sets were input to determine the result set in comprehensive search and selective search. Figure 5.1 and Figure 5.3 are two such sets illustrated here.

The global search results match all occurrences of the set whether they are components or not. The relevant search results match all occurrences within a component even if



	alarm	brakelight	crash	driver	flashing	hazard	light	reading	status	vehicle
Components	12	3	7	122	0	6	61	11	26	3
Features	23	10	11	132	7	14	81	13	54	8
Relevant	55	22	24	244	20	36	191	55	195	20
Global	86	36	39	303	26	40	310	58	292	25

Figure 5.3: Occurrence graph for a single element specification set (Sample set 2)



	1	2	3	4	5	6	7	8	9	10
Global	1.099	2.261	3.271	4.273	5.293	6.306	7.28	8.271	9.305	10.29
Relevant	0.191	0.351	0.512	0.673	0.837	0.996	1.173	1.333	1.494	1.685

Figure 5.4: Time graph for a single element specification set (Sample set 2)

they occur multiple times in a component. Searches with “Component only” match if the elements are components. Searches result with “Features” match components even if some features match.

In all cases the pattern of result set displayed similar behavior.

Observations

- The comprehensive (global) search yields a result set that contains every occurrences of the specification set even if these nodes do not characterize a component.

- The nodes representing components yields a result set which is somewhat realistic. Though these do not epitomize the complete set desired. This is often observed when the component nodes do not match, but their features collectively match the specification set.
- Whereas these nodes along with the features set yields a more elaborate result set. A match contained by any node in a set of features would result in representing the component within which it belongs.

Figure 5.2 and Figure 5.4 depicts the time taken for the specification set illustrated in Figure 5.1 and Figure 5.3 respectively. The time graph depicts the aggregate time required for global and selective search for a set of ten specification sets.

Observations

- It is evident from these figures that the time required for comprehensive search exceeds the selective search (method proposed in this thesis) by almost a factor of 5 and may be a dominant factor for large specification sets.

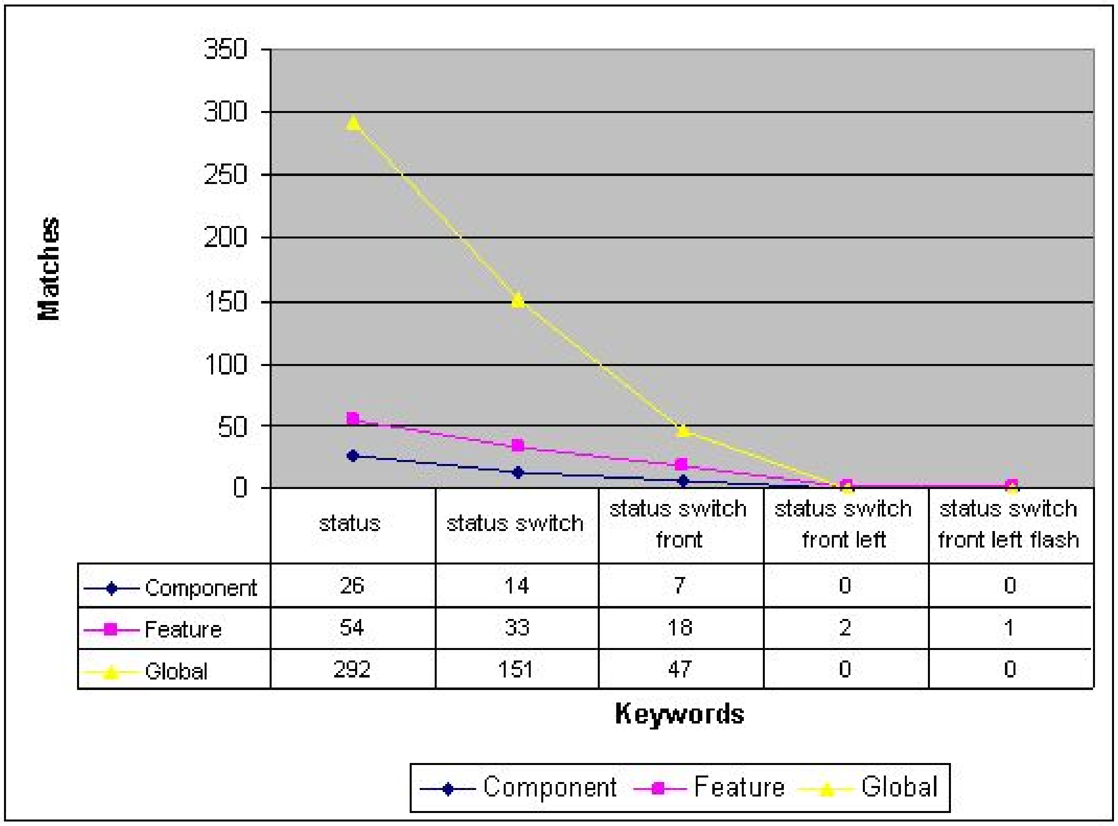
5.1.2 Evaluation using a multiple element specification set

Similar to the first the second experiment was conducted using incrementing from one to seven element specification sets as a group. Figure 5.5 and Figure 5.7 are two such sets illustrated here which depicts the result set in comprehensive search and selective search.

The "Global search" results match all occurrences of the set whether they are components or not. This search is only within the boundary of the element. Searches with "Component only" match if the elements are components. Here too the search is within the boundary of the element. Searches result with "Features" is across the boundary of features i.e. parts of the specification set occurs in different features elements within a component. The features behave collectively to match the specification set and identify a single component.

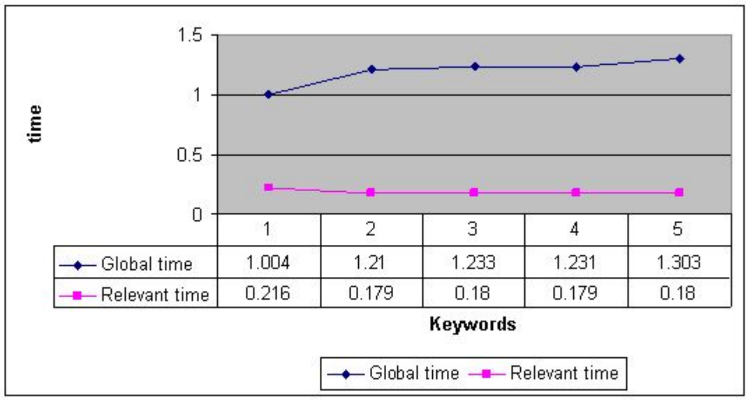
Observations

- The comprehensive search often yielded a large result set as it searches in individual node which it treats as atomic.
- The behavior exhibited is similar with varying sizes of the specification sets. As observed in Figure 5.5, the selective component-feature search result set demonstrates a value when the size of specification set exceeds 3, because in this case the matches take place across the boundary of the feature within the component. On the other hand the other methods return a null result set as the search is only within the boundary of the element.
- For any given size of specification sets, the selective component-feature search returns a much smaller result set and is more precise.



	status	status switch	status switch front	status switch front left	status switch front left flash
Component	26	14	7	0	0
Feature	54	33	18	2	1
Global	292	151	47	0	0

Figure 5.5: Occurrence graph for multiple element specification set (Sample set 1)

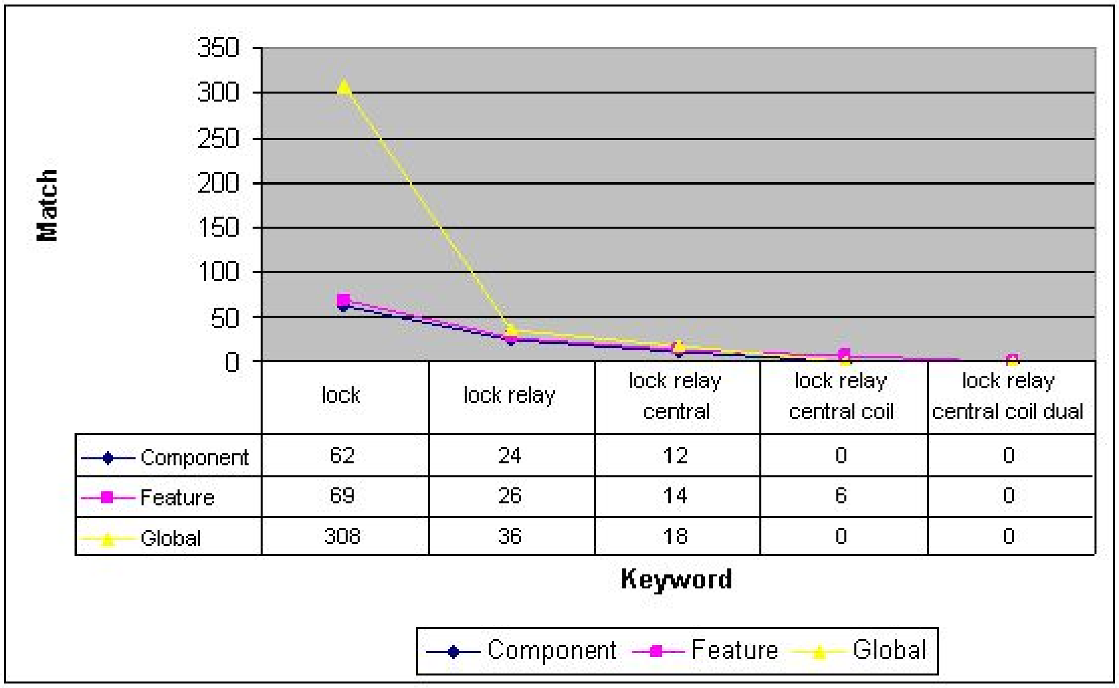


	1	2	3	4	5
Global time	1.004	1.21	1.233	1.231	1.303
Relevant time	0.216	0.179	0.18	0.179	0.18

Figure 5.6: Time graph for a multiple element element specification set (Sample set 1)

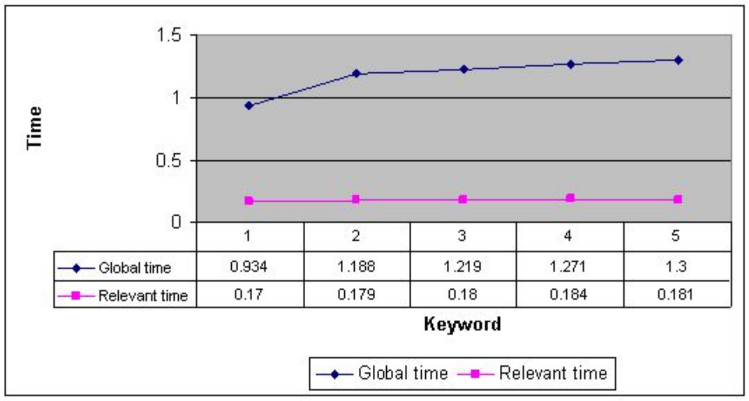
- Convergence is optimal with a specification set of size 3. A size too large of the specification set may result in a null set for both methods as shown in Figure 5.7.

Figure 5.6 and Figure 5.8 depicts the time taken for the specification set illustrated in



	lock	lock relay	lock relay central	lock relay central coil	lock relay central coil dual
Component	62	24	12	0	0
Feature	69	26	14	6	0
Global	308	36	18	0	0

Figure 5.7: Occurrence graph for a multiple element specification set (Sample set 2)



	1	2	3	4	5
Global time	0.934	1.188	1.219	1.271	1.3
Relevant time	0.17	0.179	0.18	0.184	0.181

Figure 5.8: Time graph for a multiple element specification set (Sample set 2)

Figure 5.5 and Figure 5.7 respectively. The time graph depicts the aggregate time required for global and selective search for a set of ten specification sets.

Observations

- Here too it is evident from these figures that the time required for comprehensive search exceeds the selective search (method proposed in this thesis) by almost a factor of 5 and may be a dominant factor for large specification sets.

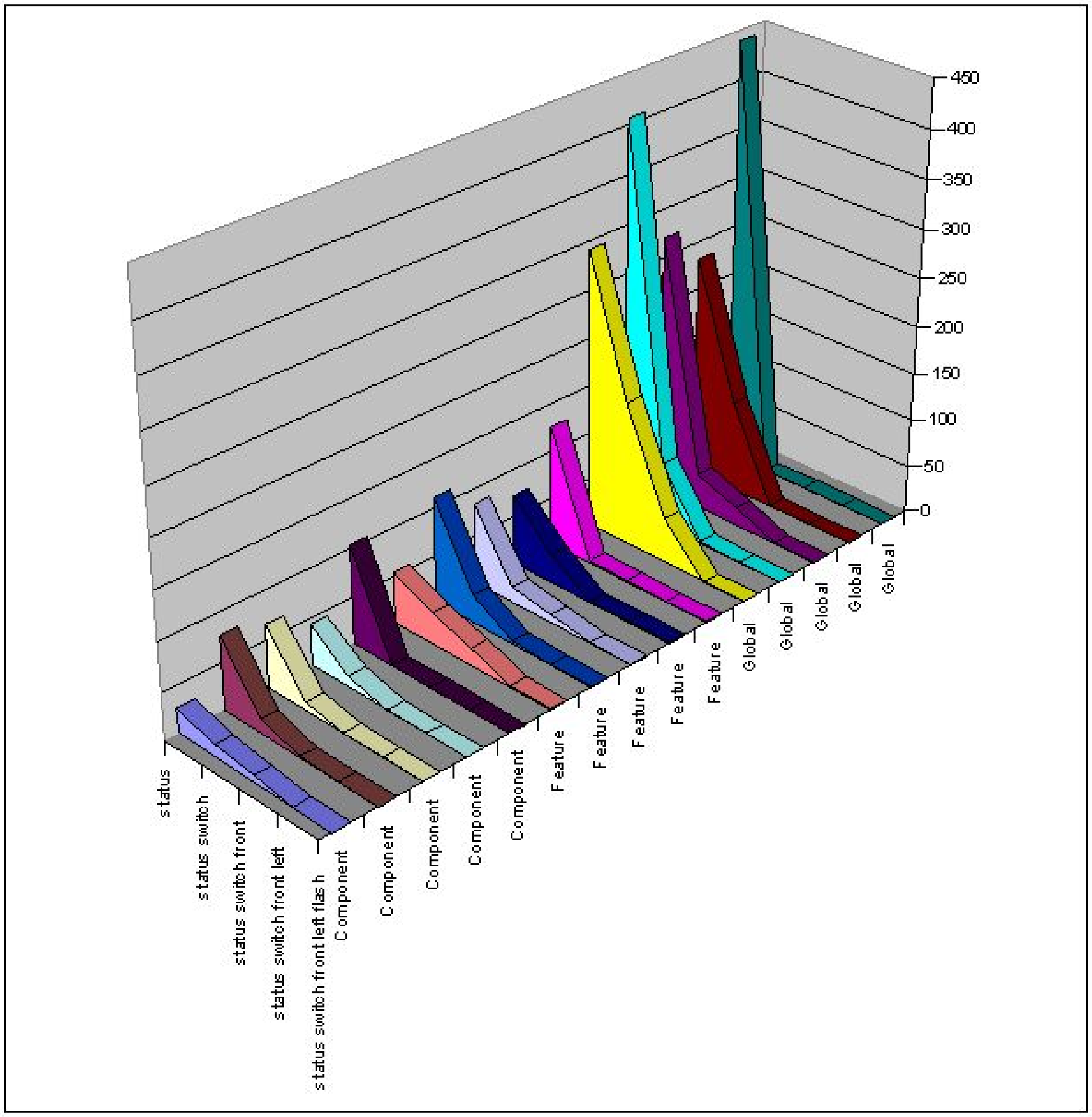


Figure 5.9: Occurrence graph for different starting point of elements within specification set

- Only a small increase in time is observed for single and multiple element selection sets. This does not have any major impact with the increasing size of the specification set.

5.1.3 Evaluation using different starting points for elements in specification sets

The third experiment was conducted using different starting points of elements within the specification sets. Figure 5.9 depicts the result set in comprehensive search and selective search.

To determine the effect of different starting points, the multiple element specification set was used and the orders of the elements within were changed to obtain five sets.

The result set for this had the same pattern as the two experiments above.

5.2 Identification through organized structuring

These names which adhere to the naming conventions usually have little relevance to describe the software components and do not contribute to the identification of their variant. The hierarchal ordering can facilitate easy identification as similar components or variants will position together.

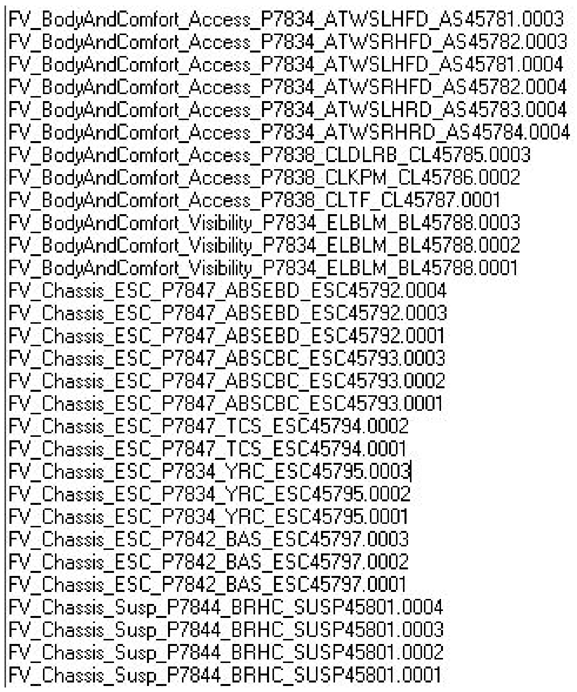
In this case study the behavioral description of the naming conventions using hierarchical state machine models was implemented. The client application was realized using the C# .NET framework and applied the behavioral model and the data model for generative purposes, while the user interface model was specified and was refined and implemented.

Figure 5.10a depicts a sample set of scattered software components, the nomenclature based on the naming conventions of AUTOSAR® used by the organization for the specification of process related information, which is based on a compatible data model.

Figure 5.10b depicts a transformed set of the sample set in Figure 5.10a using hierarchical state machine models to position similar components or variants together. The red eclipses highlight variants and their versions.

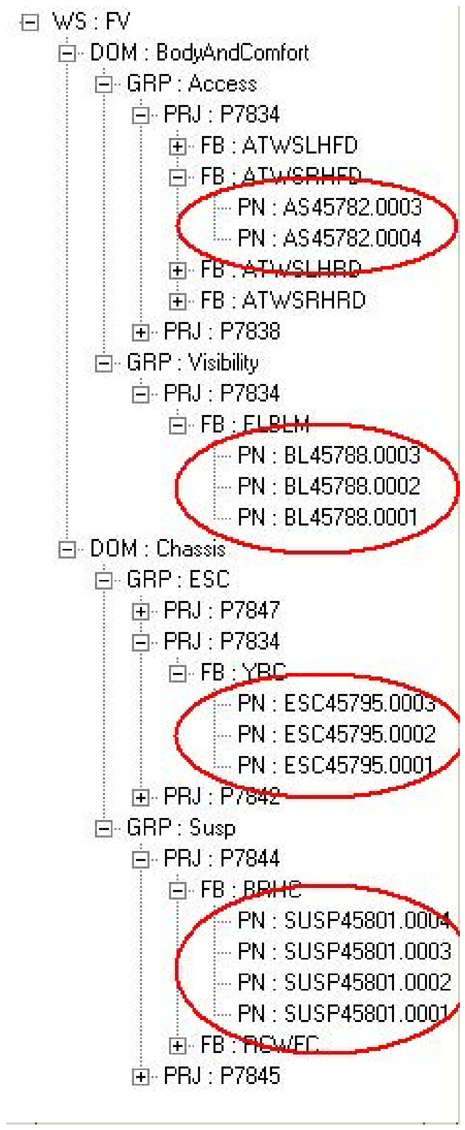
5.3 Summary

The case studies have demonstrated the feasibility for using models to define application parts in a structured and reusable way. The results demonstrate the performance improvements for the models compared to the traditional methods. On the other side the drawbacks of the multi-layered model hierarchy could be reduced by the application of programming languages with a lower abstraction for coding the framework (like C) and the usage of more static structures for managing the model elements.



```
FV_BodyAndComfort_Access_P7834_ATWSLHFD_AS45781.0003
FV_BodyAndComfort_Access_P7834_ATWSRHFD_AS45782.0003
FV_BodyAndComfort_Access_P7834_ATWSLHFD_AS45781.0004
FV_BodyAndComfort_Access_P7834_ATWSRHFD_AS45782.0004
FV_BodyAndComfort_Access_P7834_ATWSLHRD_AS45783.0004
FV_BodyAndComfort_Access_P7834_ATWSRHRD_AS45784.0004
FV_BodyAndComfort_Access_P7838_CLDLRB_CL45785.0003
FV_BodyAndComfort_Access_P7838_CLKPM_CL45786.0002
FV_BodyAndComfort_Access_P7838_CLTF_CL45787.0001
FV_BodyAndComfort_Visibility_P7834_ELBLM_BL45788.0003
FV_BodyAndComfort_Visibility_P7834_ELBLM_BL45788.0002
FV_BodyAndComfort_Visibility_P7834_ELBLM_BL45788.0001
FV_Chassis_ESC_P7847_ABSEBD_ESC45792.0004
FV_Chassis_ESC_P7847_ABSEBD_ESC45792.0003
FV_Chassis_ESC_P7847_ABSEBD_ESC45792.0001
FV_Chassis_ESC_P7847_ABSCBC_ESC45793.0003
FV_Chassis_ESC_P7847_ABSCBC_ESC45793.0002
FV_Chassis_ESC_P7847_ABSCBC_ESC45793.0001
FV_Chassis_ESC_P7847_TCS_ESC45794.0002
FV_Chassis_ESC_P7847_TCS_ESC45794.0001
FV_Chassis_ESC_P7834_YRC_ESC45795.0003
FV_Chassis_ESC_P7834_YRC_ESC45795.0002
FV_Chassis_ESC_P7834_YRC_ESC45795.0001
FV_Chassis_ESC_P7842_BAS_ESC45797.0003
FV_Chassis_ESC_P7842_BAS_ESC45797.0002
FV_Chassis_ESC_P7842_BAS_ESC45797.0001
FV_Chassis_Susp_P7844_BRHC_SUSP45801.0004
FV_Chassis_Susp_P7844_BRHC_SUSP45801.0003
FV_Chassis_Susp_P7844_BRHC_SUSP45801.0002
FV_Chassis_Susp_P7844_BRHC_SUSP45801.0001
```

(a) Sample set



```
⊟ WS : FV
    ⊟ DOM : BodyAndComfort
        ⊟ GRP : Access
            ⊟ PRJ : P7834
                ⊞ FB : ATWSLHFD
                ⊟ FB : ATWSRHFD
                    PN : AS45782.0003
                    PN : AS45782.0004
                ⊞ FB : ATWSLHRD
                ⊞ FB : ATWSRHRD
            ⊞ PRJ : P7838
        ⊟ GRP : Visibility
            ⊟ PRJ : P7834
                ⊟ FB : ELBLM
                    PN : BL45788.0003
                    PN : BL45788.0002
                    PN : BL45788.0001
    ⊟ DOM : Chassis
        ⊟ GRP : ESC
            ⊞ PRJ : P7847
            ⊟ PRJ : P7834
                ⊟ FB : YRC
                    PN : ESC45795.0003
                    PN : ESC45795.0002
                    PN : ESC45795.0001
            ⊞ PRJ : P7842
        ⊟ GRP : Susp
            ⊟ PRJ : P7844
                ⊟ FB : BRHC
                    PN : SUSP45801.0004
                    PN : SUSP45801.0003
                    PN : SUSP45801.0002
                    PN : SUSP45801.0001
                ⊞ FB : RCWFC
            ⊞ PRJ : P7845
```

(b) Transformed set

Figure 5.10: Software components with nomenclature based on the naming conventions

Chapter 6

Conclusion

A system for executing identification of model-based software components has been presented in this work, that targets to provide improved support for distributed application development of embedded software. A three-layered architecture has been introduced consisting of application layer, identification layer that forms the middle layer, and resource layer. This architecture has been used for illustrating the heterogeneity of the artifacts contained in each layer and their different roles. The proposed system is targeted for distributed systems of embedded software in automotive application.

6.1 Overview of the proposed framework

Managing variants is of utmost importance in today's large software bases as they reflect legal constraints, marketing decisions, and development cycles. As these software bases often grew from different sources and were developed by different teams using different tools it is in many cases very complicated if not nearly impossible to find artifacts that might be variants, both for historical reasons as for development purposes.

Related works presents various aspects of model-based techniques and form a basis, governing the software component models of embedded devices. The related approaches put on view a need for a generic methodology in the identification of software components developed using several design tools.

The deployment of a variability management process is considered, as it increases the possibility to derive more products. However, the management process must be formed by activities well suited to the production of the assets.

Searching algorithms have to reflect both the capability to match keywords and to reflect the structure that characterizes a component. Our proposed method is capable of both aspects and therefore helps the developer to find matches even in large and heterogeneous databases. In addition to that not only the required time for the search is a lot shorter, but also the accuracy of the retrieved set of candidates is highly improved.

The proposed approach has been evaluated in several case studies for enabling application specific functionality, while respecting the constraints of the distributed system and its members. As a consequence, while the application specific solution is specified in different models, each supporting the best suited modeling technique, this architecture can be used to manage the heterogeneity with the usage of general purpose programming languages and corresponding component.

The proposed objective does not ensure a perfect solution, but offers a variability mechanism alternative.

6.2 Future work

The proposed framework is targeting the characteristic challenges for identification of platform specific software components in distributed pervasive systems.

While the issue of heterogeneity of software models is targeted, the issue of determining the accuracy of the approach in platform specific implementation needs further investigation.

Another issue requires further research in understanding the integration of model-based techniques for analyzing the behavior of the application.

Another field of future work is the specification and realization of additional models to foster the invisibility of an application as well as to consider the further abstraction of the user interface.

Chapter 7

Publications

This chapter provides the publications written during this thesis ordered after the significance to the contributions as discussed in Section 3.3 and depicted in the layered architecture in Figure 3.9.

The concept of Model-based Variability Management is described in Section 7.1, which has been presented at the Fifth International Multi-conference on Computing in the Global Information Technology Valencia, Spain in September 2010. It intends to facilitate reusable software solution. Analyzing the user requirements by specifying the cases. Based on these cases the spatial, functional and name features are extracted to facilitate the identification and implementation of variability. This paper also presents the usage of the proposed strategy in the design tool ESCAPE.

The issues of interactive restructuring strategies to identify commonality based on architecture, functionality and naming conventions in identification, specification and realization of variants within a product development is discussed in Section 7.2. This paper has been presented in October 2010 at the 10th IFAC Workshop on Programmable Devices and Embedded Systems in Poland and is based on a scenario developed in the ESCAPE design tool.

An adaptation of a formal mathematical model to envisage the problem and prove theoretically the proposed strategy is described in Section 7.3. The paper has been presented at the Seventh International Conference on Evaluation of Novel Approaches to Software Engineering, Wrocaw, Poland in June 2012.

The overview of the process to the identification of variability, its implementation and evaluation results is described in Section 7.4. The paper has been presented at the Seventh International Multi-conference on Computing in the Global Information Technology Venice, Italy in June 2012.

The paper “Variability Identification by Selective Targeting of Significant Nodes” has been selected by the editors of IARIA Board and invited to submit an extended version of ICCGI 2012 paper to the International Journal On Advances in Networks and Services, Vol. 6 No. 1&2 2013.

2010 Fifth International Multi-conference on Computing in the Global Information Technology

Model-based Variability Management for Complex Embedded Networks

Anilloy Frank
Institute of Technical Informatics,
Graz University of Technology
Inffeldgasse 16, 8010 Graz, Austria
Email: anilloy.frank@student.tugraz.at

Eugen Brenner
Institute of Technical Informatics,
Graz University of Technology
Inffeldgasse 16, 8010 Graz, Austria
Email: brenner@tugraz.at

***Abstract*—An environment involving large scale development has a tendency to repeatedly designing functional blocks amongst different groups from several domains. To avoid at least a part of these problems a general architecture can be planned within an organization to facilitate reuse. This offers benefits such as cost savings, lower redundancy, improved productivity and quality. Complexity increases significantly with use or reuse of external blocks, as they may not adhere to the internal architecture. Major challenges are in identifying the commonality of functionality, where the designs involve variability (ability to customize) within these blocks. The concept of a Software Product Line (SPL) largely addresses this issue. But in addition to variants, their versions/releases also play an important role for effective management over the entire product cycle from concept, design, development, test up to after sales maintenance. Moreover, to reduce the development time, processes to assist team members when choosing a component from the core assets is essential. Our proposal is based on the work with the industrial design tool named ESCAPE. It addresses the issues to identify commonality based on architecture, functionality and naming conventions in identification, specification and realization of variants within a product development as well as to assist the development team. As this tool in the discussed aspect follows the practices commonly used in industry, it can be seen as an example without limiting the general validity of the proposed process for variability management.**

***Keywords*-Embedded Systems; Software Configuration; Variability Management**

I. Introduction

Reuse of automotive embedded software is difficult as it is developed for a small ECU (Electronic control unit) that lacks both processing speed and memory of general purpose machines. Moreover, the complexity of the algorithms is dramatically increasing. Also the trend is emerging to integrate different ECUs and map the software functions onto these. In view of this complexity achieving the required reliability and performance is one of the most challenging problem [8].

Variants of embedded software functions are vital in customizing the software for different regions (Europe, Asia, etc.), in particular to meet regulations of the respective regions. Also different sensors/actuators, different device drivers and the distribution of functionality to different ECUs necessitate variants. Managing variability involves extremely complex and challenging tasks, which must be supported by effective methods, techniques, and tools [4].

Most major companies have focused on trying to develop both tools and specific ECUs to enable the reuse of the software. When developing automotive software legacy code would normally be a starting point. A type of standardized software architecture may be a solution to handle such complexity as reusability may be improved and time and costs can be saved.

The proposed strategy is to introduce a variability management layer. It intends to facilitate reusable software solution. We start by analyzing the user requirements by specifying the cases. Based on these cases the spatial, functional and name features are extracted to facilitate the identification and implementation of variability. Usage of the proposed strategy is shown applying it to design tool ESCAPE as an example.

II. State-of-the-art

A. Variability

The term variability generally refers to the ability to change. Variability does not occur by chance, but is brought about on purpose alternatively ways to represent choices. Pohl suggests three questions to define variability [10].

What does vary? Identifying precisely the variable item or property of the real world. This leads us to the definition of the term variability subject. A variability subject is a variable item of the real world or a variable property of such an item.

Why does it vary? There are different reasons for an item or property to vary: different stakeholder needs, different laws, technical reasons, etc. Moreover, in the case of interdependent items, the reason for an item to vary can be the variation of another item.

How does it vary? This deals with the different shapes a variability subject can take. To identify the different shapes of a variability subject we define the term variability object (a particular instance of a variability subject).

Decision points, choices and constraint dependency rules describe variability. In definition of variability, we have variability subjects and variability objects. Variability subject may be suggested as a decision points, and the choices as variability objects. Several thousand decision points and choices are required. Constraint dependency rules are: requires or excludes decision points, requires or excludes choices, and choice requires or excludes decision points [9].

B. Variability management

Variability management (VM) is a fundamental Software Product Line Engineering (SPLE) activity that explicitly represents software artifact variations for managing dependencies among variants and supporting their instantiations throughout the SPL life cycle [4].

An SPL is a set of software-intensive systems that share a common set of features for satisfying the needs of a particular market. SPLs can reduce development costs, shorten time-to-market, and improve the product quality by reusing core assets for project-specific customizations [4].

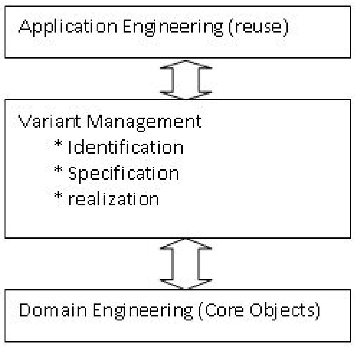

Application Engineering (reuse)
Variant Management
* Identification
* Specification
* realization
Domain Engineering (Core Objects)

Figure 1. Variability management in product lines

To enable reuse on a large scale, SPLE identifies and manages commonalities and variations across a set of system artifacts such as requirements, architectures, code components, and test cases.

SPLE can be categorized into domain engineering and application engineering (see Fig. 1). Domain engineering involves design, analysis and implementation of core objects, whereas application engineering is reusing these objects for product development [1], [2].

Activities on the variant management process involves Variability identification, Variability specification and Variability realization [3].

- A Variability Identification process will incorporate feature extraction and feature modeling.
- A Variability Specification process is to derive a pattern.
- A Variability realization process is a mechanism to allow variability.

III. Escape

ESCAPE is an authoring tool for graphical definition, manipulation and analysis of functional networks with ability to manage a wide range of configurations. The modelling can be performed without regard to their subsequent implementation in software (SW) and hardware (HW). In addition it provides diagnostics tools like dependency analysis, fault back tracking, support for mathlab/simulink simulation models and various programming language implementations [13].

There are 3 views,

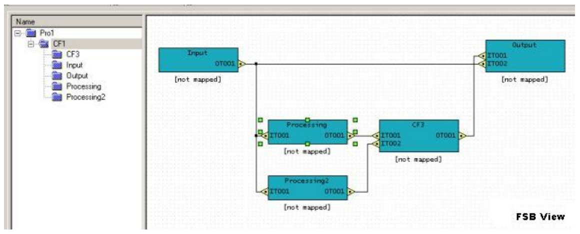


Figure 2. Escape: FSB View (screenshot)

- **FSB (Functional structure builder)** (see Fig. 2) facilitates to build the structure of the model. In the FSB view a project is displayed in the form of a tree that represents the product model structure. The left pane displays the hierarchy of compound functions within a project. The right pane displays the schematic of the functional structure.

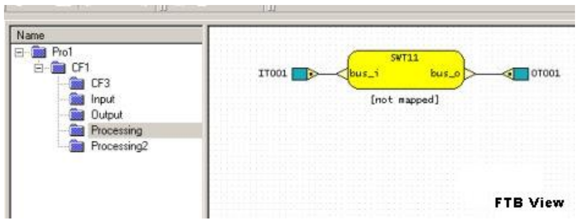


Figure 3. Escape: FTB View (screenshot)

- **FTB (Function type builder)** (see Fig. 3) provides defining hardware and software types. The left pane of the FTB view displays the hierarchy of hardware types and software types. The right pane of the FTB view shows the definitions of these types.These types can have basic- and user-defined subtypes.
- **HSB (Hardware Structure Builder)** (see Fig. 4) which allows networking ECUs and mapping the software functions. The HSB view shows the hierarchy of networked ECUs interconnected by bus systems. Various Buses and ECUs can be instantiated. The compound functions designed in the FSB view can be

mapped on these ECUs in this view.

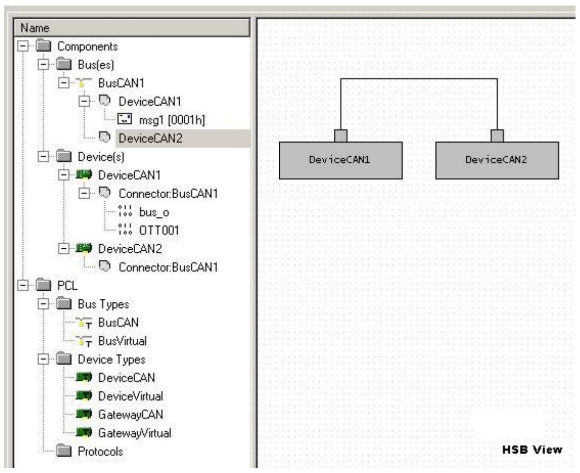


Figure 4. Escape: HSB View (screenshot)

As the depth of hierarchy in FSB and FTB grows, possibilities of redesigning similar functions and user-defined types which may be variants of existing types increases, simply because the mechanism to manage variants is lacking.

This tool displays inadequacy in handling variants. Therefore a need arises to develop patterns (parameters and procedures) for extracting object features, and to develop a mechanism to manage and support variability.

Example: The problem may be best explained by citing an example.

Broadly the domains in an automotive industry are Body and Comforts, Powertrains, Chassis, Safety and Multimedia and Telematics. Numerous technical and administrative groups are involved in the development of each domain.

These domains further have subsets

Body and comforts

- Central Locking
- Wiper/Washer
- Anti Theft Warning System
- Window Control

:

Powertrains

- Transmission System
- Combustion Engine
- Engine torque and mode management

:

Chassis

- Vehicle Longitudinal Control
- Electronic Parking Brake
- Adaptive Cruise Control
- Steering System

:

Current trends in automotive software development is mapping of software components on networked ECUs, a shift from an ECU based approach to a function based approach. System configuration is possible on the basis of description of software component, ECU resources and constraints in system description.

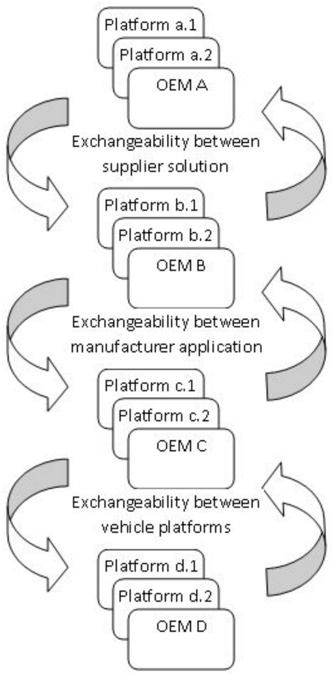

Platform a.1
Platform a.2
OEM A
Exchangeability between supplier solution
Platform b.1
Platform b.2
OEM B
Exchangeability between manufacturer application
Platform c.1
Platform c.2
OEM C
Exchangeability between vehicle platforms
Platform d.1
Platform d.2
OEM D

Figure 5. Exchangeability of software components

Software components are either developed internally using a standardized naming convention, or obtained from various OEMs. The use of standardized application interfaces increases the quality on exchange with suppliers and improves the software integration from the system standpoint. OEMs are applying AUTOSAR naming conventions; and there are more than 10,000 interfaces and calibrations data for industrial purposes based on these specifications of naming convention [14].

This tends to severely affect the development cycle. Variability management becomes crucial in such scenarios. Managing complexity by exchangeability (see Fig. 5) and reuse of software components, integration of functional modules from multiple suppliers, scalability to different vehicle and platform variants, standardization of basic system functions, software updates over vehicle lifetime, maintainability throughout the whole product life cycle, meet the non functional legal requirements, resources efficiency, redundancy and many others factors play an important role in the process and need to be handled properly.

IV. Proposed Pattern for Variability Management

The proposed method is to introduce a layer that provides the capability for variability management (see Fig. 6), which enhances both readability and clarity in representation of variability. It offers the user an option for the configuration of all information related to variation points, insertion of variants, the definition of their types and their storage in a data base. All that information is made available so that the user can select and create new variants. Thus, depending on the user's selection, objects are created with appropriate type, properties, and method information previously registered during the configuration within the database.

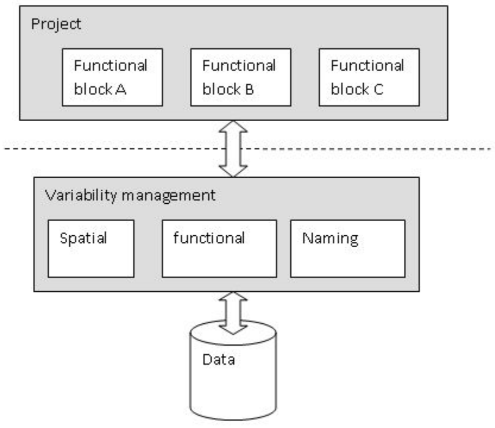


Figure 6. Variant management layer

A. Specification of the Cases

The example (see Fig. 7), displays the development phase for two products A and B. Both products reuse functions from a data backbone. The development phase may also include the development of software functions in the repository themselves, which may have numerous versions. The software development for product A is tested with the ver-3 of functional block B and a variant of ver-4 of functional block A. Similarly product B also uses different version and variants of functional blocks from the data backbone.

A certain version/release of a variant of a functional block suitable in a product may exhibit an improper behavior, when a different version or variant is selected. Even a more improved version of the same variant may not yield a reliable results.

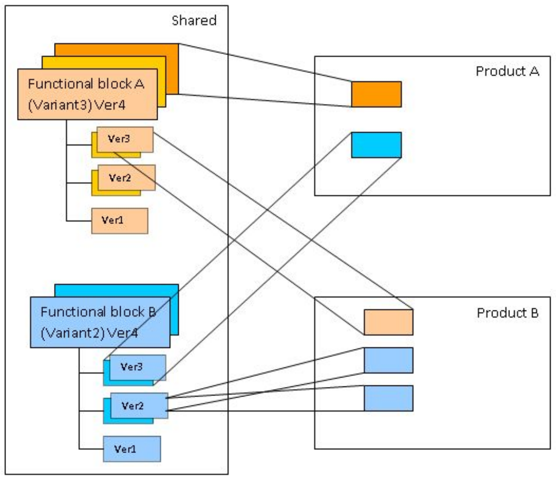

Shared
Functional block A (Variant3) Ver4
Ver3
Ver2
Ver1
Functional block B (Variant2) Ver4
Ver3
Ver2
Ver1
Product A
Product B

Figure 7. Figure describing the variants, versions and products

Case 1: The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks whilst using later version of the function and may fail to perform as expected.

Case 2: To enable parallel development, extracting features to identify and specify the functional blocks based on architecture, functionality in the repository.

Case 3: A process that tracks usability and prevent inconsistencies due to deprecating variants and version from repository.

Case 4: Establishing a testing mechanism for validations to maintain high quality for components and it variants.

Case 5: To assist the developer, a process to intelligently determine whether a functional block or its variant exist in the data backbone, to avoid redesigning of existing functions, thereby improving productivity.

V. Feature extraction

Based on the specification of the cases broadly the decision points, choices and constraint dependency rules pattern for feature extraction to extract spatial, functional, and name are described (see Fig. 8).

A. Architectural features

- Structural design rules, design convention, consistent ways: these could include the number of inputs and outputs, their data types, and their default values,
- Parts in different sub tree which could be similar: a group of blocks having same functionality,
- Brief description about the object, comments, description convention: meta data, keywords and other textual description about the object.

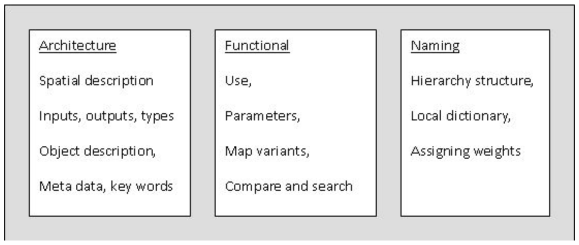


Figure 8. Feature extraction: Spatial, Functional and Name

B. Functionality of the objects

- Methods to identify identical blocks: based on functional use,
- Representing hierarchy in structure of variants: based on parameters,
- Possibility to map variants,
- Comparing and searching; rules for comparing functionality and search for the functional block.

C. Naming conventions

In order to maintain uniformity in names of functional blocks, parameters, etc., usually long and cumbersome naming conventions are used, which are not just difficult to remember but also tedious to construct. Eg. a typical convention for naming defined as project_group_year_functionalBlock_type_number could look like

BRGD_CODCA_2010_FNRS_USINT_14357
BRGD_CODCA_2010_FNRS_USINT_14486
BRGD_CODCA_2010_FNRS_USINT_14332

These names are neither user friendly nor meaningful. But to assist the user to identify, search and construct these names comfortably displaying them as hierarchy and a procedure to navigate and to simplify the construction of such names will enable the user to quickly build long names uniformly over the entire project.

- Building a local dictionary of all names used in the project if they are not the standard words,
- Using some characters as delimiters and displaying suggestions to the next level in the hierarchy from the existing names, as the user types new names or edits existing ones. In addition suggestions of thesaurus, synonyms, antonyms, etc.,
- Automatically navigate through the levels of hierarchy as the corresponding suggestions are selected or build the names as one traverses through the hierarchy,
- Assigning weights depending on related names, most often used, frequently used, etc.

VI. Conclusion

The deployment of a variability management process is considered as it increases the possibility to derive more products. However, the management process must be formed by activities well suited to the production of the assets.

The proposed objective does not ensure a perfect solution, but offers a variability mechanism alternative. Feedback from developers on parts of features included in the prototype indicate positive on preliminary results without stressable data as they can include variability management in their development cycle. Although no research results exist about the necessary overhead, here we actually see benefits because it increases significantly the reusability, reliability and hence the productivity. It also cuts cost by avoiding repeatability of redesigning existing functions and time to deliver.

References

[1] F. Bachmann and P.C. Clements, ”Variability in software product lines,” Technical Report -CMU/SEI-2005-TR-012, 2005.
[2] J. Bosch, ”Design and use of software architectures: Adopting and evolving a product-line approach,” Addison-Wesley, 2000
[3] L.A. Burgareli, Selma, S.S. Melnikoff, and G.V. Mauricio Ferreira, ”A variation mechanism based on adaptive object model for software product line of brazilian satellite launcher,” First IEEE Eastern European Conference on the Engineering of Computer Based Systems, 2009, pp. 24-31
[4] P. Clements and L. Northrop, ”Software product lines: Practices and patterns,” Addison-Wesley, 2007
[5] H. Gomaa, ”Designing software product lines with UML 2.0: from use cases to pattern-based software architectures,” Addison-Wesley, 2005
[6] H. Gomaa and D.L. Webber, ”Modeling adaptive and evolvable software product lines using the variation point model,” Proceedings of the 37th Hawaii international Conference on System Sciences, Washington, Jan 2004, 10 pp.
[7] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, ”FORM: A feature-oriented reuse method with domain specific reference architectures,” Annual Software Engineering-5, 1998, pp. 143-168
[8] D. Kum, G. Park, S. Lee, and W. Jung, ”AUTOSAR migration from existing automotive software,” International Conference on Control, Automation and Systems , Oct 2008, pp. 558-562
[9] A.O. Elfaki, S. Muthaiyah, H.M. Ibrahim, S. Amnuaisuk, and Chin Kuan Ho, ”Defining variability in DSS: An intelligent method for knowledge representation and validation,” Proceedings of the 43rd Hawaii International Conference on System Sciences, Jan 2010, pp. 1-9
[10] K. Pohl, G. Bäckle, and F.J. Linden, ”Software product line engineering: Foundations, principles and techniques,” Springer-Verlag, New York, 2005
[11] J. Weiland and D. Chrysler, ”Configuring variant-rich automotive software architecture models,” published by the IEE, Michael Faraday House, Mar 2007, pp. 73-80
[12] J.W. Yoder, F. Balaguer, and R. Johnson, ”Architecture and design of adaptive object-models,” ACM Sigplan Notices. Vol. 36, Fasc. 12, 2001, pp. 50-60
[13] ESCAPE ”http://www.gigatronik2.de/index.php?seite=escape_produktinfos_de&navigation=3019&root=192&kanal=html”
[14] AUTOSAR ”http://www.autosar.org/download/conferencedocs/03_AUTOSAR_Tutorial.pdf”

Strategy for Modeling Variability in Configurable Software

Anilloy A. Frank [*] **Eugen Brenner** [**]

[*] *Institute of Technical Informatics, Graz University of Technology, Inffeldgasse 16, 8010 Graz, Austria (e-mail: anilloy.frank@student.tugraz.at).*
[**] *Institute of Technical Informatics, Graz University of Technology, Inffeldgasse 16, 8010 Graz, Austria (e-mail: brenner@tugraz.at)*

Abstract: Large development environments with numerous groups from several domains tend to repeatability in designing functional blocks. With constantly changing requirements within the set of products derived from these functional blocks, the variability needs to evolve. Many embedded systems are implemented with a set of alternative function variants to adapt to the changing requirements. The ever growing number of variable features and variants makes the problem more and more unmanageable. Typically, obsolete features are not removed and add to the chaos. Planning a standardized architecture within an organization may address a part of these problems and facilitate reuse. Major challenges are in identifying the commonality of functionality, where the designs involve variability (ability to customize) within these blocks. In addition to variants, versions/releases of functional blocks also play an important role for the effective management over the entire product cycle. Moreover, to reduce the development time, interactive processes to assist team members when choosing a component from the core assets are essential.
Our proposal addresses the issues of interactive restructuring strategies to identify commonality based on architecture, functionality and naming conventions in identification, specification and realization of variants within a product development as well as to assist the development team. Though based on use of the industrial design tool named ESCAPE, the discussed aspects follow the practices commonly used in industry. It therefore can be seen as examples without limiting the general validity of the proposed process for variability management.

Keywords: Design Tool, Embedded Systems, Software configuration,Variability management.

1. INTRODUCTION

The current development trend in automotive software is mapping of software components on networked ECU's (Electronic control unit), which includes shift from an ECU based approach to a function based approach. Reuse of automotive embedded software is difficult, as it is developed for a small ECU that lacks both processing speed and memory of general purpose machines. Moreover, the complexity of the algorithms is dramatically increasing. In view of this complexity, achieving the required reliability and performance is one of the most challenging problems Kum et al. (2008).

Variants of embedded software functions are vital in customizing the software for different regions (Europe, Asia, etc.), in particular it is necessary to meet legal regulations of the respective regions. Also different sensors/actuators, different device drivers and the distribution of functionality to different ECUs necessitate variants. Managing variability involves extremely complex and challenging tasks, which must be supported by effective methods, techniques, and tools Clements and Northrop (2007).

Most major companies have focused on trying to develop both tools and specific ECUs to enable the reuse of the software. A type of standardized software architecture may be a solution to handle such complexity, as reusability may be improved, and time and costs can be saved.

The proposed strategy intends to facilitate automated and interactive strategies for reusable software solutions. We start by analyzing the user requirements by specifying the cases. Based on these cases the spatial, functional, and name features are extracted to facilitate the identification and implementation of variability. The usage of the proposed strategy is shown applying it to design tool ESCAPE as an example.

2. VARIABILITY MANAGEMENT

Variability management (VM) is a fundamental activity in Software Product Line Engineering (SPLE), that explicitly represents software artifact variations for managing dependencies among variants and supporting their instantiations throughout the Software Product Line (SPL) life cycle Clements and Northrop (2007).

A SPL is a set of software-intensive systems that share a common set of features for satisfying the needs of a particular market. SPLs can reduce development costs, shorten time-to-market, and improve the product quality by reusing core assets for project-specific customizations Clements and Northrop (2007); Gomaa (2005).

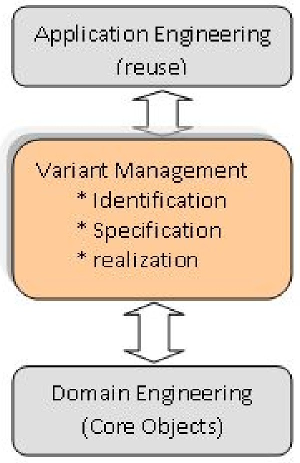


Fig. 1. Variability management in product lines

To enable reuse on a large scale, SPLE identifies and manages commonalities and variations across a set of system artifacts such as requirements, architectures, code components, and test cases.

SPLE can be categorized into domain engineering and application engineering (see Fig. 1). Domain engineering involves design, analysis, and implementation of core objects, whereas application engineering is reusing these objects for product development Bachmann and Clements (2005); Bosch (2000).

Activities on the variant management process involves variability identification, variability specification, and variability realization Burgareli et al. (2009).

- A variability identification process will incorporate feature extraction and feature modeling.
- A variability specification process is to derive a pattern.
- A variability realization process is a mechanism to allow variability.

3. ESCAPE

ESCAPE is an authoring tool for graphical definition, manipulation, and analysis of functional networks with ability to manage a wide range of configurations. The modeling can be performed without regard to their subsequent implementation in software (SW) and hardware (HW). In addition it provides diagnostics tools like dependency analysis, fault back tracking, support for mathlab/simulink simulation models, and various programming language implementations GIGATRONIK (2009).

ESCAPE supports 3 different views (refer Fig. 2):

- FSB (Functional structure builder) facilitates to build the structure of the model,
- FTB (Function type builder) provides defining hardware and software types, and
- HSB (Hardware Structure Builder) which allows networking ECUs and mapping the software functions.

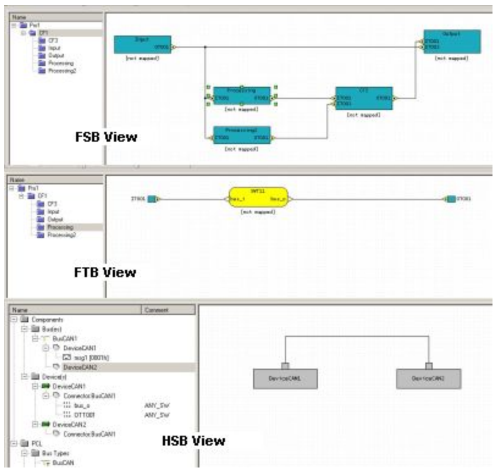


Fig. 2. Escape: FSB, FTB, and HSB View

In the FSB view a project is displayed in the form of a tree that represents the product model structure. The left pane displays the hierarchy of compound functions within a project. Groups or teams can work independently on a sub-tree for the development of functional parts of the product model. The right pane displays the schematic of the functional structure, and leafs of the hierarchy are the instances of the hardware- and software-types defined in the FTB. These sub-trees can then be integrated into a single large model in FSB. It also provides tools to trace the forward and backward impact on the model.

The left pane of the FTB view displays the hierarchy of hardware types and software types. These types can have basic- and user-defined subtypes. The user-defined hardware types can have subtypes like input hardware (e.g. sensors, switches), output hardware (e.g. actuators), and control hardware which can be further sub-typed to any depth grouped by similar hardware domain, hardware logic, and actual hardware type. The right pane of the FTB view shows the definitions of these types. Similarly the basic software types include integer, boolean, float, double, etc., and the user defined software types can have subtypes to any depth grouped by domain related user-defined software types based/derived from basic types, software parameters, etc.

The HSB view shows the hierarchy of networked ECUs interconnected by bus systems. Various buses and ECUs can be instantiated. The compound functions designed in the FSB view can be mapped on these ECUs in this view.

As the depth of hierarchy in FSB and FTB grows, the probability for redesigning similar functions and user-defined types, which may be variants of existing types increases, simply because the mechanism to manage variants is lacking.

This tool displays inadequacy in handling variants. Therefore the need arises to develop patterns (parameters and procedures) for extracting object features, and to develop a mechanism to manage and support variability.

4. SPECIFICATION OF THE CASES

Our example (see Fig. 3) displays the development phase for two products A and B. Both products reuse functions from a data backbone. The development phase may also include the development of software functions in the repository itself, which may have numerous versions. The software development for product A is tested with the ver-3 of functional block B and a variant of ver-4 of functional block A. Similarly product B also uses different version and variants of functional blocks from the data backbone.

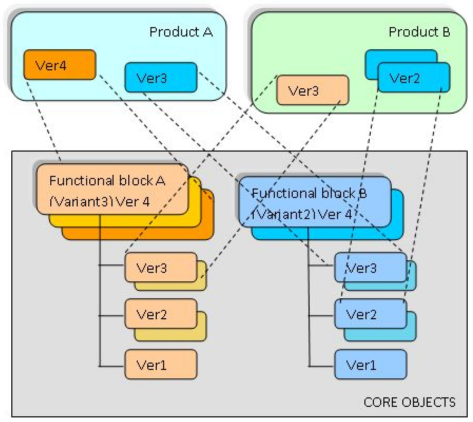


Fig. 3. Example describing variants, versions, and products

A certain version/release of a variant of a functional block suitable in a product may exhibit an improper behavior, when a different version or variant is selected. Even a more improved version of the same variant may not yield a reliable results.

When handling variants and versions we can state the following problems:

Case 1: The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks, whilst using later version of the function may fail to perform as expected.

Case 2: To enable parallel development, it is necessary to be able to extract features and to identify and specify the functional blocks in the repository based on architecture and functionality.

Case 3: A process that tracks usability and prevents inconsistencies due to deprecate variants and version from repository is required.

Case 4: A testing mechanism for validations in order to maintain high quality for components and its variants has to be established.

Case 5: The developer has to be assisted by a process to intelligently determine whether a functional block or its variant should exist in the data backbone, to avoid redesigning of existing functions, thereby improving productivity.

5. PROPOSED PATTERN FOR VARIABILITY MANAGEMENT

The term variability generally refers to the ability to change. Variability does not occur by chance, but is brought about on purpose, showing alternative ways which represent choices. Pohl suggests three questions to define variability Pohl et al. (2005); Gomaa and Webber (2004).

What does vary? Identifying precisely the variable item or property of the real world. This leads us to the definition of the term variability subject. A variability subject is a variable item of the real world or a variable property of such an item.

Why does it vary? There are different reasons for an item or property to vary: different stakeholder needs, different laws, technical reasons, etc. Moreover, in the case of interdependent items, the reason for an item to vary can be the variation of another item.

How does it vary? This deals with the different shapes a variability subject can take. To identify the different shapes of a variability subject we define the term variability object (a particular instance of a variability subject).

Decision points, choices and constraint dependency rules describe variability. In the definition of variability, we have both variability subjects and variability objects. Variability subject may be understood as decision points, and variability objects as choices. Typically several thousand decision points and choices are required. Constraint dependency rules are: requires or excludes decision points, requires or excludes choices, and choice requires or excludes decision points Osman et al. (2010).

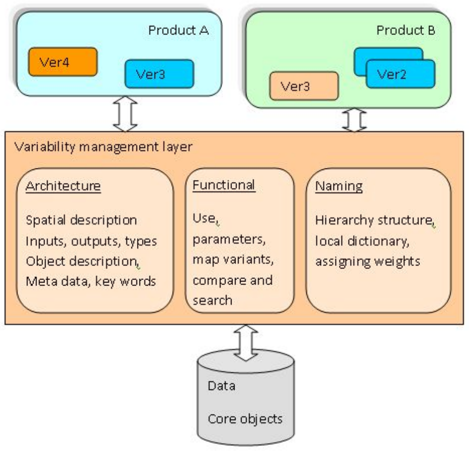


Fig. 4. Variant management layer

The proposed method is to introduce a layer that provides the capability for variability management (see Fig. 4), which enhances both readability and clarity in representation of variability. It offers the user an option for the configuration of all information related to variation points, insertion of variants, the definition of their types and their storage in a data base. All that information is made available so that the user can select and create new variants. Thus, depending on the user's selection, objects are created with appropriate type, properties, and method information previously registered within the database during the configuration Weiland and D. Chrysler (2007); Yoder et al. (2001).

6. FEATURE EXTRACTION

Broadly the decision points, choices and constraint dependency rules pattern for feature extraction to extract spatial, functional, and name are described.

The feature extraction of Software components may be summed up as

$$E = \{A, F, N, R\} \tag{1}$$

where E denotes the set of features extracted, A denotes the set of architectural features, F denotes the set of functional features, N denotes the set of naming conventions, and R denotes the set of relations.

Extracting features is a structuring concept and does not define functionality. Naming convention are solely for interactive solutions. Both non-variant and variant features are included in the set. This enables for a closer approximation to determine the Software component.

In addition some parameters of Software components also contribute to variants. These parameters are represented as

$$P = \{N, V, T\} \tag{2}$$

where P denotes the set of parameters, N denotes the set of naming conventions, V denotes the set of values, and T denotes the set of data types of these parameters.

Parameters which are constant are non-variants.

6.1 Architectural features

With standardized naming convention as a rule, software functions tend to have names which are a combination of project, group, year, numbers, etc. These name convention described later (see subsection 6.3) are not helpful to the developer in identifying functional blocks from the core assets. Variability may only be in some spatial features of software functions like a different parameter or a different data type for inputs or outputs, etc. Thus in an interactive tool one of the techniques to identify software functions can be based on the architectural features.
To enhance accessibility the architectural features supported are:

- Structural design rules, design convention, consistent ways: these could include the number of inputs and outputs, their data types, and their default values,
- Parts in different sub tree which could be similar: a group of blocks having same functionality,
- Brief description about the object, comments, description convention: meta data, keywords and other textual description about the object.

The architectural features are represented as

$$A = \{I, O, D, S\} \tag{3}$$

where A denotes the set of architectural features, I denotes the set of inputs, O denotes the set of outputs, D denotes the set of description, keyword or meta-data of these components, and S denotes the subtree.

Inputs and outputs can be represented as

$$I = \{V, T\} \tag{4}$$

$$O = \{V, T\} \tag{5}$$

where I denotes the set of inputs, O denotes the set of outputs, V denotes the set of values, and T denotes the set of data types.

Inputs and outputs must match exactly to reasonably support exchangeable components.

For any set of objects $E \subseteq \mathcal{E}$, the set of common attributes is defined as

$$\sigma(E) = \{a \in A \mid \forall e \in E : (e, a) \in R\} \tag{6}$$

Similarly for any set of attributes $A \subseteq \mathcal{A}$, the set of common objects is defined as

$$\tau(A) = \{e \in E \mid \forall a \in A : (e, a) \in R\} \tag{7}$$

The maximum collection of objects with common set of attributes, if $A = \sigma(E)$ and $E = \tau(A)$

6.2 Functionality of the objects

Though the spatial features may provide a certain amount of accessibility, there can be numerous functions with different functionality having the same or similar spatial features. Thus identifying software functions based on spatial features alone is not sufficient and therefore identifying functionality becomes essential. Also it can further narrow down the search. Defining architectural features is a relatively simpler process than defining functionality. Describing functionality is an extremely complex process with interrelation between numerous parameters, logic tables, data types, states, etc.

To further narrow down the accessibility, functionality features supported are:

- Methods to identify identical blocks: based on functional use,
- Representing hierarchy in structure of variants: based on parameters,
- Possibility to map variants,
- Comparing and searching: rules for comparing functionality and search for the functional block.

The functionality features are represented as

$$F = \{D, \Gamma, M\} \tag{8}$$

where F denotes the set of functionality, D denotes the set of descriptions, keywords or meta-data, Γ denotes the set of tables describing functionality, and M denotes the set of map variants.

For any set of objects $E \subseteq \mathcal{E}$, the set of common functionality is defined as

$$\rho(E) = \{f \in F \mid \forall e \in E : (e, f) \in R\} \tag{9}$$

Similarly for any set of functionality $F \subseteq \mathcal{F}$, the set of common objects is defined as

$$\delta(F) = \{e \in E \mid \forall f \in F : (e, f) \in R\} \tag{10}$$

The maximum collection of objects with a common set of functionality is obtained when $F = \rho(E)$ and $E = \delta(F)$

6.3 Naming conventions

Naming conventions of functional blocks, parameters, etc., are usually long and cumbersome, which are not just difficult to remember but also tedious to construct. Eg. a typical convention for naming defined as: project_group_year_functionalBlock_type_number looks like

BRGD_CODCA_2010_FNRS_USINT_14357
BRGD_CODCA_2010_FNRS_USINT_14486
BRGD_CODCA_2010_FNRS_USINT_14527

These names are neither user friendly nor meaningful. But to assist the user to identify, search, and construct these names, comfortably displaying them as hierarchy, as well as having a procedure to navigate and simplify the construction of such names will enable the user to quickly build long names uniformly over the entire project. Often used activities that are supported by our solution are:

- Building a local dictionary of all names used in the project, if they are not the standard words,
- Using some characters as delimiters and displaying suggestions to the next level in the hierarchy from the existing names, as the user types new names or edits existing ones. In addition suggestions of thesaurus, synonyms, antonyms, etc. are given,
- Automatically navigate through hierarchy levels as the corresponding suggestions are selected, or build the names as one traverses through the hierarchy,
- Assigning weights depending on related names, most often used names, frequently used names, etc.

The naming convention are represented as

$$N = \{L, \Psi, \mu, W\} \tag{11}$$

where N denotes the set of naming convention, L denotes the set of custom local dictionary with non-standard words, Ψ denotes the delimiter, μ denotes the functionality to traverse hierarchy and W denotes the set of weights.

7. CONCLUSION

The deployment of a variability management process is considered, as it increases the possibility to derive more products. However, the management process must be formed by activities well suited to the production of the assets.

The proposed objective does not ensure a perfect solution, but offers a variability mechanism alternative, helps to reduce redundancy, enhances the speed of development, and restricts the user's handling errors, thereby facilitating a better management of variants. Including variability management in the development process therefore not only improves the quality of the delivered software under reduced costs, but in the long run it makes larger solutions with individualization of software for specific targets possible.

REFERENCES

Bachmann, F. and Clements, P.C. (2005). Variability in software product lines. *Technical Report -CMU/SEI-2005-TR-012.*

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

Burgareli, L., Selma, S.S., Melnikoff, and Ferreira, G.V.M. (2009). A variation mechanism based on adaptive object model for software product line of brazilian satellite launcher. *First IEEE Eastern European Conference on the Engineering of Computer Based Systems.*

Clements, P. and Northrop, L. (2007). Software product lines: Practices and patterns.

GIGATRONIK (2009). Escape. `http://www.gigatronik2.de/index.php?seite=escape_produktinfos_de&navigation=3019&root=192&kanal=html`.

Gomaa, H. (2005). *Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley.

Gomaa, H. and Webber, D. (2004). Modeling adaptive and evolvable software product lines using the variation point model. *Proceedings of the 37th Hawaii international Conference on System Sciences, Washington.*

Kum, D., Park, G., Lee, S., and Jung, W. (2008). Autosar migration from existing automotive software. *International Conference on Control, Automation and Systems*, 558–562.

Osman, A., Muthaiyah, S., Ibrahim, H.M., Amnuaisuk, S., and Ho, C.K. (2010). Defining variability in dss: An intelligent method for knowledge representation and validation. *Proceedings of the 43rd Hawaii International Conference on System Sciences.*

Pohl, K., Backle, G., and Linden, F.J. (2005). *Software Product LineEngineering: Foundations, Principles and Techniques.* Springer-Verlag, New York.

Weiland, J. and D. Chrysler, A. (2007). Configuring variant-rich automotive software architecture models. *published by the IEE, Michael Faraday House.*

Yoder, J.W., Balaguer, F., and Johnson, R. (2001). Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36, 50–60.

A GENERIC APPROACH FOR THE IDENTIFICATION OF VARIABILITY

Anilloy Frank[1] and Eugen Brenner[1]
[1]*Institute of Technical Informatics, Technische Universitt, Inffeldgasse 16, 8010 Graz, Austria*
anilloy.frank@student.tugraz.at, brenner@tugraz.at

Keywords: Design Tools; Embedded Systems; Feature Extraction; Software Reusability; Variability Management;

Abstract: The automotive electrical/electronics (E/E) embedded software development largely uses Model Based Software Engineering (MBSE), an industrially accepted approach. With an ever increasing complexity of embedded software, the E/E models in automotive applications are getting enormously unmanageable. The heterogeneous nature of projects developed using several modeling and simulation tools, and the hierarchical structure with numerous composite components deeply embedded within, tends to repeatability. Hence it is often necessary to define a mechanism to identify reusable components from these that are embedded deep within. The proposed approach addresses the identification process in the development and deployment of software components used in the realization of such distributed processes, by selectively targeting the component-feature model (CF) instead of a comprehensive search to improve the identification. It addresses the issues to identify commonality of variants within a product development. The results obtained are faster and are more accurate compared to other methods.

1 INTRODUCTION

The current development trend in automotive software is to map embedded software components on networked Electronic Control Units (ECU) (Kum et al., 2008).

Variants of embedded software functions are inevitable in customizing for different regions (Europe, Asia, etc.), to meet regulations of the respective regions. Also different sensors / actuators, different device drivers, and distribution of functionality on different ECUs necessitate variants (Frank and Brenner, 2010a); (Frank and Brenner, 2010b).

Often it is apparent to procure well established software components tested for performance, safety and reliability from external sources or Original Equipment Manufacturers (OEM), illustrated in Figure 1. The black box characteristics of such software components, when integrated in models, further add to the complexity, and work as hindrance in managing variability.

Managing variability involves extremely complex and challenging tasks, which must be supported by effective methods, techniques, and tools (Clements and Northrop, 2007). In view of this complexity, achieving the required reliability and performance is one of the most challenging problems (Bosch, 2000).

The proposed strategy is a model-based approach for the distributed business process. The approach intends to facilitate automated and interactive strategies to addresses the identification process in the development and deployment of software components. We start by analyzing the textual representation of the model structure and form a concept to extract an element list to facilitate the identification of variability. Based on the adaptation of a formal mathematical model presented in this paper is the implementation and evaluation of the proposed strategy.

2 RELATED WORK

For achieving large-scale software reuse, reliability, performance and rapid development of new products, Software Product-Line Engineering(SPLE) is an effective strategy. SPLE can be categorized into domain engineering and application engineering (Bachmann and Clements, 2005); (Bosch, 2000). Domain engineering involves design, analysis and implementation of core objects, whereas application engineering is reusing these objects for product development.

Model Driven Software Development (MDSD) is typically realized in a distributed system environment for the development of automotive applications and products (Kulesza et al., 2007). Model-based techniques are used to support the usage of platform independent code. The abstract specification of the components is done by domain experts, and the task for deploying these components on different platforms is handled separately by specific platform developers. As a consequence the effort required for porting elements is reduced (Gomaa and Webber, 2004).

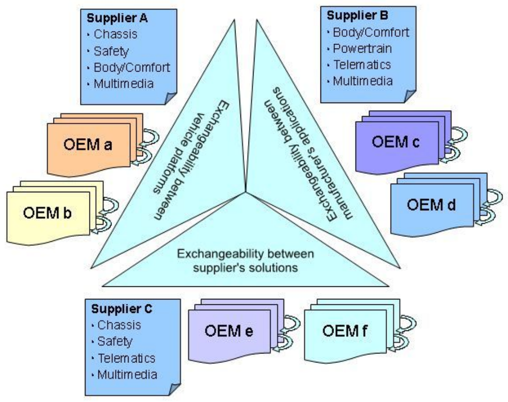


Figure 1: External components as a hindrance to variability management.

The Software Product-Line (SPL) approach promotes the generation of specific products from a set of core assets, domains in which products have well defined commonalities and variation points (Oliveira et al., 2005).

One of the fundamental activity in SPLE is Variability management (VM). Throughout the SPL life cycle VM explicitly represents variations of software artifacts, managing dependencies among variants and supporting their instantiations (Clements and Northrop, 2007).

Activities on the variant management process involves variability identification, variability specification and variability realization.

- The Variability Identification Process will incorporate feature extraction and feature modeling.
- The Variability Specification Process is to derive a pattern.
- The Variability Realization Process is a mechanism to allow variability.

One of the basic element in these approaches is a software component, which is an execution unit with well defined interfaces (Szyperski, 2002). The usage of software components is driven by the requirements of improving the reusability of developed software artifacts. Mapping of software components on networked ECU is a distinct shift from Component Based Software Engineering (CBSE). Software components are combined with the help of assembly descriptions. They are specified in the development phase and are resolved in the deployment phase of a CBSE process (Crnkovic, 2005).

Despite of all the hype there is a lack of an overall reasoning about variability management.

Although variability management is recognized as an important issue for the success of SPLs, there are not many solutions available (Heymans and Trigaux, 2003). However, there are currently no commonly accepted approaches that deal with variability holistically at architectural level (Galster and Avgeriou, 2011).

3 PROPOSED APPROACH

Models confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications, and AUTOSAR® were considered, although this concept is not limited to the automotive domain alone.

3.1 Problem Analysis

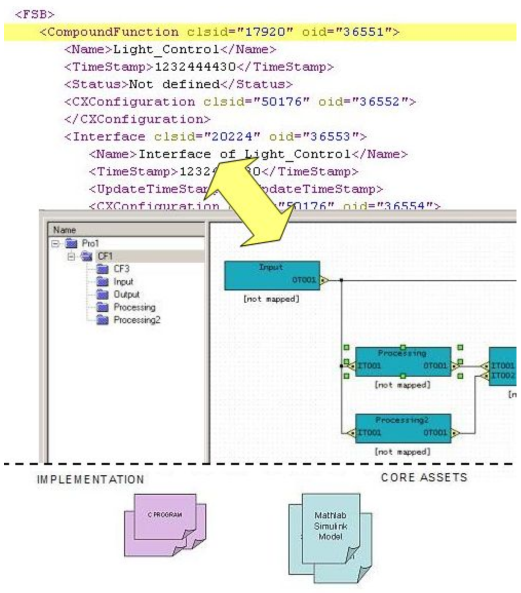
- *Textual representation:* An analysis of the models exhibits a common architecture. Figure 2 depicts the textual representation that underlies the graphical model. The textual representation usually is given in XML, which strictly validates to a schema.

  The schema defines elements transformed into an explicit mapping that specify integrity constraints modeled as real world entities in the project.

- *Significant nodes:* Examination of the nodes in the textual representation of models depicted in Figure 3 reveals some interesting information. The nodes outlined in rectangles provide important information regarding the identity, specification, physical attributes, etc. of a component, but are insignificant from the perspective of variant.

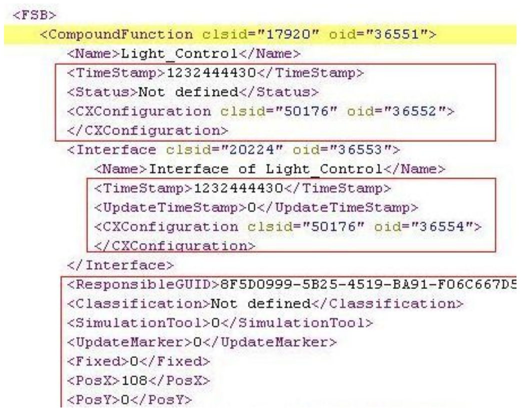
  The CF model is derived manually from the set of elements in the schema that signify components are clustered to obtain a component list; and elements within these which characterize features as a feature vector.

- *Heterogeneous modeling environment:* A heterogeneous modeling environment may consist of numerous design tools, each with its own unique schemata, to offer integrity and avoid inconsistencies. Developed projects have to be strictly validated to the schemata of these tools.



```
<FSB>
   <CompoundFunction clsid="17920" oid="36551">
      <Name>Light_Control</Name>
      <TimeStamp>1232444430</TimeStamp>
      <Status>Not defined</Status>
      <CXConfiguration clsid="50176" oid="36552">
      </CXConfiguration>
      <Interface clsid="20224" oid="36553">
         <Name>Interface of Light_Control</Name>
         <TimeStamp>1232[illegible]0</TimeStamp>
         <UpdateTimeSta[illegible]pdateTimeStamp>
         <CXConfiguration [illegible]"50176" oid="36554">
```

Figure 2: Mapping textual and graphical representations.



```
<FSB>
   <CompoundFunction clsid="17920" oid="36551">
      <Name>Light_Control</Name>
      <TimeStamp>1232444430</TimeStamp>
      <Status>Not defined</Status>
      <CXConfiguration clsid="50176" oid="36552">
      </CXConfiguration>
      <Interface clsid="20224" oid="36553">
         <Name>Interface of Light_Control</Name>
         <TimeStamp>1232444430</TimeStamp>
         <UpdateTimeStamp>0</UpdateTimeStamp>
         <CXConfiguration clsid="50176" oid="36554">
         </CXConfiguration>
      </Interface>
      <ResponsibleGUID>8F5D0999-5B25-4519-BA91-F06C667D5
      <Classification>Not defined</Classification>
      <SimulationTool>0</SimulationTool>
      <UpdateMarker>0</UpdateMarker>
      <Fixed>0</Fixed>
      <PosX>108</PosX>
      <PosY>0</PosY>
```

Figure 3: XML Nodes that are not significant for variability.

3.2 Concept and Approach

The work flow of the concept is depicted in Figure 4.

The top layer here represents the domain or core assets. Sets of projects confirming to respective schemata of several modeling tools are depicted. Models are hugely hierarchical in nature with numerous composite components deeply embedded within projects.

The middle layer is a semi-automatic variability identification layer, subdivided into two parts. The left part depicts sets of distinct component lists and corresponding feature vectors derived manually from the schemata for each modeling tool; a collection of elements that represent components and their descriptive features that significantly contribute to the identification of the component's variant. To assist the selection the right part is a customized parser that generates a relevant lexicon from the set of software components within a project and set of rules (viz., mandatory, optional, exclude) to govern the identification of variability.

The lower layer is an application layer where the application developer provides the specification set and based on the rules the result set is returned.

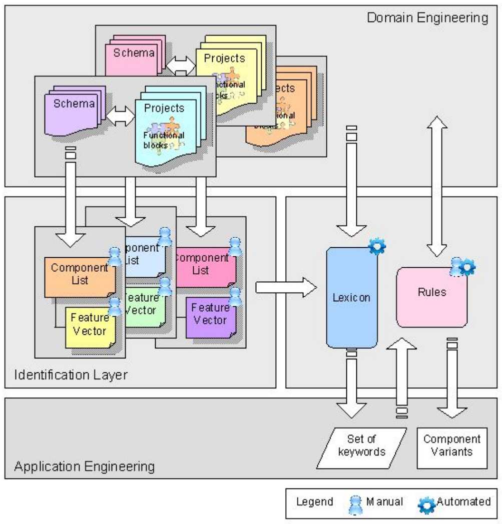


Figure 4: Work flow for semi-automated identification of variants.

Algorithm

1. Obtain a subset of nodes from within the schema that signifies importance and description of the whole, or part components to a component list.
2. Components themselves may further be comprised of sub nodes (components and features). Not all sub nodes of the components in the component list may be essential to describe variability.
3. Therefore for each element within the component list further obtain a subset of the sub nodes from the schema, which describes features of the components to a feature vector.
4. Using the component list and the feature vector generate a dictionary of keywords from within the

project, along with the frequency to determine the weight or significance of the keywords.

5. Apply rules (like contains all, one or more, and does not contain) to search the specification set to obtain an intersection set, union set, and difference set to identify the components.

3.3 Mathematical Model

The formal representation of such a model is complex. The software model is composed of a set of functions, which further contain sub-functions and so exhibiting a hierarchical structure. The software models can be defined as

$$P = \{E, \Gamma\} \tag{1}$$

$P = \{p_1, p_2, ...p_n\}$ is a finite set of models consisting of elements that forms the functional modeling (the abstract specification of the components), solution modeling (the implementation of the components), and architecture design (deploying and mapping these components on different platforms). In addition it also contains elements that are general rationale and do not signify any of these functionality.

$E = \{e_1, e_2, ...e_m\}$ is a finite set of elements that constitutes elements providing general information (viz., id, time stamp, date, owner, etc.), elements that form components, elements within the components that represent features. Some of these elements may be categorized as elements that describe variability or that contribute to signify variants.

$\Gamma = \{\gamma_1, \gamma_2, ...\gamma_o\}$ is a finite set of elements which describes complex relationships that reflect information relationships, inheritance flow, and message exchanges.

Each of these models validate to a schema; and there is an isomorphic mapping relationship between the elements of the schema and the models.

We define a schema S as a set of formulas that specify integrity and constraints

$$S = \{N, C\} \tag{2}$$

The schema defines the structure, entities, attributes, relationships, views, indexes, packages, procedures, triggers, types, sequences, synonyms and other elements.

$N = \{n_1, n_2, ...n_k\}$ denotes a finite set of nodes or elements in a schema that describes integrity, whereas $C = \{c_1, c_2, ...c_j\}$ denotes a finite set of elements in a schema that describes constraints, and further to adapt a heterogeneous environment which consists of projects developed using several modeling and simulation tools.

$S = \{s_1, s_2, ...s_i\}$ is a finite set of schemata each representing a modeling or simulation tools.

At user reconfiguration level, the software model is represented in an abstract form, consisting of modules, functions, relationship, information, inherited flow, and message flow. Subdividing the set of nodes N and the set of constraints C into general elements and elements that signify

$$\begin{aligned} N &= \{n, \eta\} \\ C &= \{c, \upsilon\} \end{aligned} \tag{3}$$

$\eta = \{\eta_1, \eta_2, ...\eta_p\}$ and $\upsilon = \{\upsilon_1, \upsilon_2, ...\upsilon_q\}$ are a finite set of nodes and constraints respectively that signify components, features, functions, relations, whereas, $n = \{n_1, n_2, ...n_r\}$ and $c = \{c_1, c_2, ...c_s\}$ are a finite set of nodes and constraints respectively that signify all other nodes.

Targeting all nodes in the model that are isomorphically mapped to η and υ leads to a set of nodes that can be viewed as a Significant Nodes (SN). As the functions are hierarchical the software model may be viewed as a Significant Node Mesh (SNM).

SN can be defined as

$$SN = \{C_m, F_c, N_c, R\} \tag{4}$$

where $C_m = \{C_{m1}, C_{m2}, ...C_{mn}\}$ is a finite set of all components defined on the set P, $\forall\, C_{mi} \subset C_m$ and $i = 1, ...m$, C_{mi} is a finite set including all components of p_i, and is a subset of C_m. $F_c = \{F_{c1}, F_{c2}, ...F_{co}\}$ is a finite set of all features defined on the set P, $\forall\, F_{cj} \subset F_c$ and $j = 1, ...o$, F_{cj} is a finite set including all features of p_i, and is a subset of F_c. N_c and R denotes the set of naming conventions and the set of relations respectively.

Let S_N denote the nodes in model P and M denotes the nodes in schema S. Then there is a map (function) τ from S_N into M, defined such that $\tau(n)$ is the definition (or rule) of $n \in S_N$ in M.

$$\tau : S_N \to M \tag{5}$$

Let S_c be an element of S representing a component c. Let E_C be the subset of the schema S which is extracted manually such that each element represents a variant component.

$$E_C = \{S_c \in S : c \text{ represent a component}\} \tag{6}$$

Let E_F be the subset of a S which is extracted manually such that each element represents a feature of the component c.

$$\begin{aligned} E_F = \{E_{F_c} \in S : E_{F_c} &\text{ represents a feature} \\ &\text{of the component } c\} \end{aligned} \tag{7}$$

$E_F(i, c)$ denotes the i^{th} element of E_F of a component c.

Let C_1 be the subset of C such that all elements of C_1 are represented in E_C.

$$C_1 = \{c \in C : \tau(c) \in E_C\} \tag{8}$$

Let F'_c be the subset of F_c such that element of F'_c are represented in E_F.

$$F'_c = \{f \in F_c : \tau(f) \in E_F\} \tag{9}$$

Let $F'(i,c)$ be the i^{th} element of F'_c, where i is an integer.

Let V be the specification set. Then

$$R = \left[\bigcup_{c \in C_1} \left(c\left(\bigcup_i F'(i,c)\right)\right)\right] \bigcap V \tag{10}$$

In this method the number of elements in the resultant set R is

$$|R| = \left|\left[\bigcup_{c \in C_1} \left(c\left(\bigcup_i F'(i,c)\right)\right)\right] \bigcap V\right| \tag{11}$$

On the other hand, in global search we get

$$|R| = |V \cap N| \tag{12}$$

where N is the set of nodes in the project.

Clearly

$$|V \cap N| \geq \left|\left[\bigcup_{c \in C_1} \left(c\left(\bigcup_i F'(i,c)\right)\right)\right] \bigcap V\right| \tag{13}$$

Hence we conclude an improved result set using this approach.

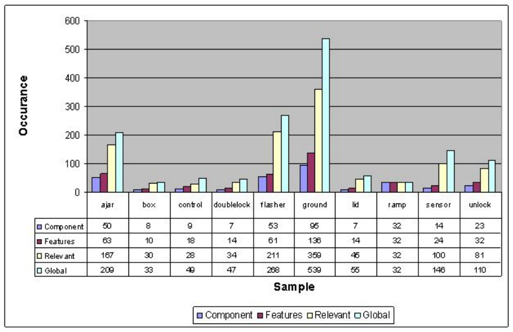
3.4 Evaluation

The case studies targeted the design of model-based software components firstly in an industrial use case where the project model was developed using the design tool ESCAPE® (Gigatronik, 2009), and secondly in a case study targeting the execution of specific paradigms based on the naming convention of AUTOSAR®.

The specific project data set, which was used to verify the implementation, consisted of a total of 32909 elements. A total of 1583 of these elements signify components; these were categorized into 23 categories when enlisted in the component list. A total of 13353 elements signified features that were assigned into 12 categories.

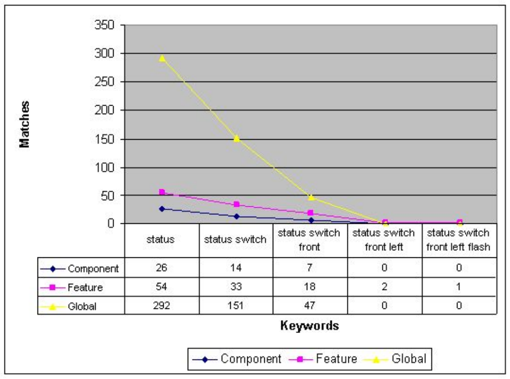
Three different approaches were adopted to evaluate and determine the performance with respect to comprehensive search. The notion of comprehensive search is used, when scanning all occurrences of the specification set within projects, irrespective of whether they are components or features of those components. This may return a result set that contains false matches.

- The evaluation using a single element specification set is illustrated in Figure 5.
- The evaluation using multiple element specification set, up to seven elements as a group is illustrated in Figure 6.
- The evaluation using different starting points for elements in specification sets is shown in Figure 7.



	ajar	box	control	doublelock	flasher	ground	lid	ramp	sensor	unlock
Component	50	8	9	7	53	95	7	32	14	23
Features	63	10	18	14	61	136	14	32	24	32
Relevant	167	30	28	34	211	359	45	32	100	81
Global	209	33	49	47	268	539	55	32	146	110

Figure 5: Occurrence graph for a single element specification set.



	status	status switch	status switch front	status switch front left	status switch front left flash
Component	26	14	7	0	0
Feature	54	33	18	2	1
Global	292	151	47	0	0

Figure 6: Occurrence graph for multiple element specification sets.

Observations

- The comprehensive search often yielded large result sets, as it searches in individual nodes that are treated as atomic.The result set contains every occurrence of the specification set, even if these nodes do not characterize a component.
- The exhibited behavior is similar to the varying size of the specification set. As observed in Figure 6, the selective component-feature search result set delivers a value when the size of the specification set exceeds 3, because in this case the matches take place across the boundary of the feature within the component. On the other hand

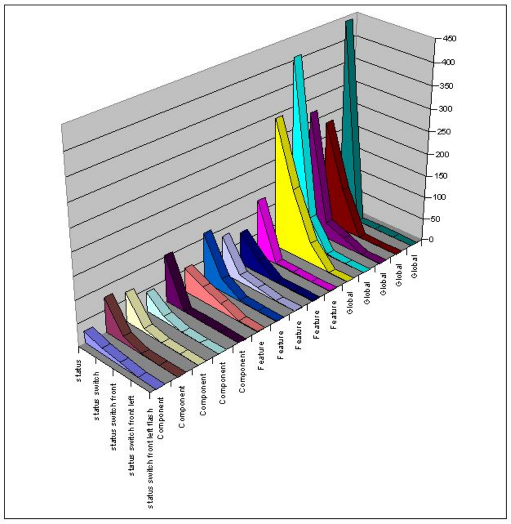


Figure 7: Occurrence graph for different starting points.

the other methods returns a null result set as the search is only within the boundary of the element.

- The nodes representing components yield a result set which is somewhat realistic, though these do not epitomize the complete set desired.
- These nodes along with the feature set yield a more elaborate result set. A match contained by any node in a set of features would result in representing the component to which it belongs.
- For any given size of the specification set, the selective component-feature search returns a much smaller result set and is more precise.
- Convergence is optimal with a specification set of size 3. If the size of the specification is too large the result may be null for both methods as shown in Figure 6.
- To determine the effect of different starting points, a multiple-element specification set was used, where the orders of the elements were changed to obtain five sets. The result set for this exhibits the same pattern as the two experiments above.

4 CONCLUSION

An approach that can significantly improve the identification of variant is proposed by targeting significant nodes instead of comprehensive search. The approach reflect both the capability to match keywords and to reflect the structure that characterizes a component enabling the identification in large distributed and heterogeneous development environment. The developed prototype is itself independent of a specific tool as it works on textual descriptions that typically are available in XML. Although the accuracy of the retrieved set of candidates is highly improved. The future work may comprise to extend the concept to specify and verify reusable components.

REFERENCES

Bachmann, F. and Clements, P. C. (2005). Variability in software product lines. *Technical Report -CMU/SEI-2005-TR-012*.

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

Clements, P. and Northrop, L. (2007). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Crnkovic, I. (2005). Component-based software engineering for embedded systems. *Software Engineering, ICSE 2005. Proceedings. 27th International Conference*, pages 712–713.

Frank, A. and Brenner, E. (2010a). Model-based variability management for complex embedded networks. *2010 Fifth International Multi-conference on Computing in the Global Information Technology*, pages 305–309.

Frank, A. and Brenner, E. (2010b). Strategy for modeling variability in configurable software. *Programmable Devices and Embedded Systems PDES 2010*.

Galster, M. and Avgeriou, P. (2011). Handling variability in software architecture: Problem and implications. *2011 Ninth Working IEEE/IFIP Confernce on Software Architecture*, pages 171–180.

Gigatronik (2009). Escape. http://www.gigatronik_2.de/index.php?seite=escape_produktinfos_de &navigation=3019&root=192&kanal.html.

Gomaa, H. and Webber, D. (2004). Modeling adaptive and evolvable software product lines using the variation point model. *Proceedings of the 37th Hawaii international Conference on System Sciences, Washington*.

Heymans, P. and Trigaux, J. (2003). Software product line: state of the art. *Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur*.

Kulesza, U., Alves, V., Garcia, A., Neto, A. C., Cirilo1, E., de Lucena, C. J. P., and Borba, P. (2007). Mapping features to aspects: A model-based generative approach. *Current Challenges and Future Directions, Lecture Notes in Computer Science*, pages 155–174.

Kum, D., Park, G., Lee, S., and Jung, W. (2008). Autosar migration from existing automotive software. *International Conference on Control, Automation and Systems*, pages 558–562.

Oliveira, E., Gimenes, I., Huzita, E., and Maldonado, J. (2005). A variability management process for software product lines. *CASCON 05*, pages 225 – 241.

Szyperski, C. (2002). Component software: Beyond object-oriented programming. *2nd Edition, Addison-Wesley, USA*.

Variability Identification by Selective Targeting of Significant Nodes

Anilloy Frank
Institute of Technical Informatics,
Technische Universitt
Inffeldgasse 16, 8010 Graz, Austria
Email: anilloy.frank@student.tugraz.at

Eugen Brenner
Institute of Technical Informatics,
Technische Universitt
Inffeldgasse 16, 8010 Graz, Austria
Email: brenner@tugraz.at

***Abstract*—The automotive industry is characterized by numerous product variants, often driven by embedded software. With ever increasing complexity of embedded software, the electrical/electronic models in automotive applications are getting enormously unmanageable. Significant concepts for modeling and management of variability in the software architecture are under development. Models are hugely hierarchical in nature with numerous composite components deeply embedded within projects comprising of Simulink models, implementations in legacy C, and other formats. Hence, it is often necessary to define a mechanism to identify reusable components from these that are embedded deep within. The proposed approach is selectively targeting the component-feature model (CF) instead of an inclusive search to improve the identification. We explore the components and their features from a predefined component node list and the features node vector respectively. It addresses the issues to identify commonality in identification, specification and realization of variants within a product development. Since the approach does not depend on the depth of the components or on its order, it serves well with all the scenarios, thereby exhibiting a generic nature. The results obtained are faster and more accurate compared to other methods.**

***Keywords*-Design Tools; Embedded Systems; Feature Extraction; Software Reusability; Variability Management.**

I. Introduction

Embedded systems are microcontroller-based systems built into technical equipment mainly designed for a dedicated purpose, where communication with the outside world occurs via sensors and actuators [1]. Although this definition implies that embedded systems are used as isolated units, there is also a trend to construct distributed pervasive systems by connecting several embedded devices, as noted by Tanenbaum and van Steen [2].

The current development trend in automotive software is to map software components on networked Electronic Control Units (ECU), which includes the shift from an ECU based approach to a function based approach. Also, according to data presented by Ebert and Jones, up to 70 electronic units are used in a car containing embedded software consisting of more than 100 million lines of object code, which is mainly responsible for the value creation of the car.

Variants of embedded software functions are vital in customizing for different regions (Europe, Asia, etc.), to meet regulations of the respective regions. Also different sensors / actuators, different device drivers, and distribution of functionality on different ECUs necessitate variants. Managing variability involves extremely complex and challenging tasks, which must be supported by effective methods, techniques, and tools [3].

Ebert and Jones present recent data about embedded software in [4], stating that the volume of embedded software is increasing between 10 and 20 percent per year as a consequence of the increasing automation of devices and their application in real world scenarios.

The proposed strategy is to introduce a variability identification layer. It intends to facilitate a reusable software solution. We start by analyzing the textual representation of the model structure. Based on this we form a concept to extract an element list to facilitate the identification of variability. Both implementation and evaluation of the proposed strategy is based on a technically advanced adaptation of a formal mathematical model, which is beyond the scope of this paper.

II. Software reuse

In the 1960s, reuse of software started with subroutines, followed by modules in the 1970s and objects in the 1980s. About 1990 components appeared, followed by services at about 2000. Currently, Software Product Lines (SPL) are state of the art in the reuse of software.

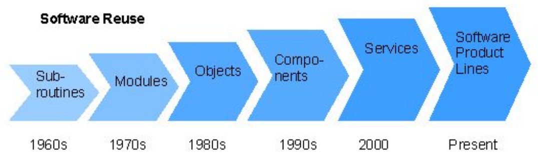


Figure 1. Software reuse history.

Figure 1 shows a short history of the usage of reuse in software development. The key idea of Product Lines is very old; it is based on Henry Ford's mass customization to provide a effective way for cheap individual cars. Today, many different approaches exist to the implementation of Software Product Lines.

A SPL is a set of software-intensive systems that share a common set of features for satisfying a particular market segment's needs. SPL can reduce development costs, shorten time-to-market, and improve product quality by reusing core assets for project-specific customizations [3][5].

Despite of all the hype, there is a lack of an overall reasoning about variability management.

The SPL approach promotes the generation of specific products from a set of core assets, domains in which products have well defined communalities and variation points[6].

Although variability management is recognized as an important issue for the success of SPLs, there are not many solutions available [7]. However, there are currently no commonly accepted approaches that deal with variability holistically at architectural level [8].

III. Variability management

One of the fundamental activity in Software Product Line Engineering (SPLE) is Variability management (VM). Throughout the SPL life cycle, VM explicitly represents variations of software artifacts, managing dependencies among variants and supporting their instantiations [3].

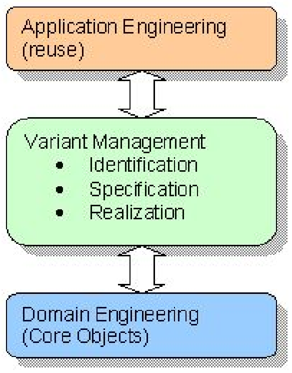

Application Engineering (reuse)
Variant Management
Identification
Specification
Realization
Domain Engineering (Core Objects)

Figure 2. Variability management in product lines.

To enable reuse on a large scale, SPLE identifies and manages commonalities and variations across a set of system artifacts such as requirements, architectures, code components, and test cases. As seen in the Product Line Hall of Fame [9], many companies have adopted this development approach.

SPLE as depicted in Figure 2 can be categorized into domain engineering and application engineering [10][11]. Domain engineering involves design, analysis and implementation of core objects, whereas application engineering is reusing these objects for product development.

Activities on the variant management process involves variability identification, variability specification and variability realization [12].

- The Variability Identification Process will incorporate feature extraction and feature modeling.
- The Variability Specification Process is to derive a pattern.
- The Variability Realization Process is a mechanism to allow variability.

IV. Software Architecture

Figure 3 depicts a layered software architecture that is considered in the proposed architecture. It shows a comparison of distributed systems and platform with the proposed layered architecture and the feasibility of mapping the corresponding artifacts and responsibilities for each layer.

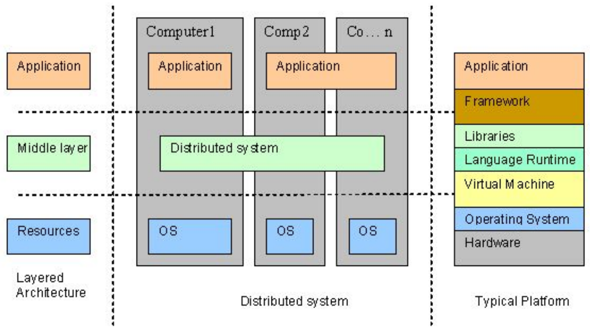


Figure 3. Comparison of architecture, system, and platform.

The definition of software architecture given in the ISO/IEC 42010 IEEE Std 1471-2000: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [13]."

In the middle illustrates a distributed system. Tanenbaum and van Steen define distributed systems as "A distributed system is a collection of independent computers that appears to its users as a single coherent system [2]."

Similarly to the right side is depicted a typical platform as specified by Atkinson and Kühner in Model Driven Architectures (MDA) [14].

V. Specification of the Cases

To enable identification of variability for software components in a distributed system within the automotive domain [15][16], we enlist the specifications below:

- *Specification of components by compatibility*
  The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks, whilst

using later version of the function may fail to perform as expected.

- *Extract, identify, and specify features*
  To enable parallel development, it is necessary to be able to extract features, and to identify and specify the functional blocks in the repository based on architecture and functionality.
- *Usability and prevention of inconsistencies*
  A process that tracks usability and prevents inconsistencies due to deprecate variants and versions in the repository is required.
- *Testing mechanism for validations*
  A testing mechanism for validations in order to maintain high quality for components and its variants has to be established.
- *Mechanism for simplified assistance*
  The developer has to be assisted by a process to intelligently determine whether a functional block or its variant should exist in the data backbone to avoid redesign of existing functions, thereby improving productivity.

VI. Proposed approach for Variability Identification

Models confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications and AUTOSAR® were considered. Although this concept is not limited to automotive domain alone.

A. Project analysis

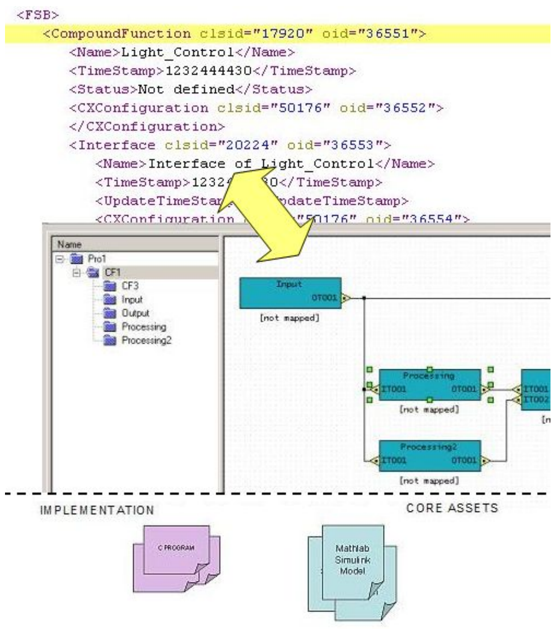
An analysis of the models exhibits a common architecture. Figure 4 depicts the textual representation that underlies several graphical model. The textual representation usually is given in XML, which strictly validates to a schema. A heterogeneous modeling environment may consist of numerous design tools, each with its own unique schema, to offer integrity and avoid inconsistencies. Developed projects have to be strictly validated to the schemas of these tools.

A closure examination of the nodes in the textual representation of models depicted in Figure 5 reveals some interesting information. The nodes outlined in rectangles provide important information regarding the identity, specification, physical attributes, etc. of a component, but are insignificant from the perspective of variant.

B. Concept and approach

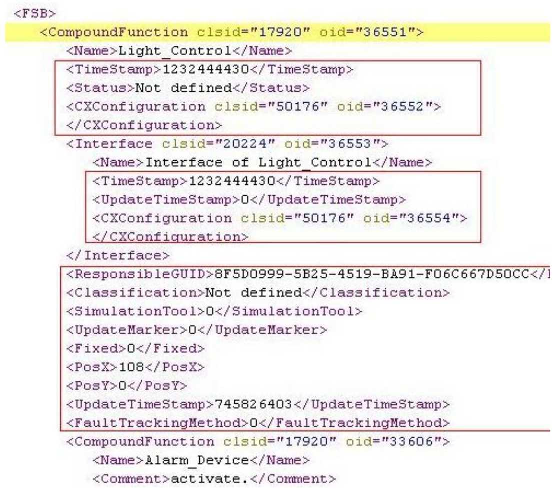
The basic concept to identify variability is depicted in Figure 6.

The left side is a set of projects that have software components hierarchically embedded. These projects validate to the corresponding schemas. The middle layer is an identification layer with three functional blocks. A set of component lists is derived from the node list in the schema. Similarly a feature vector is derived from it that corresponds to components.



```
<FSB>
    <CompoundFunction clsid="17920" oid="36551">
        <Name>Light_Control</Name>
        <TimeStamp>1232444430</TimeStamp>
        <Status>Not defined</Status>
        <CXConfiguration clsid="50176" oid="36552">
        </CXConfiguration>
        <Interface clsid="20224" oid="36553">
            <Name>Interface of Light_Control</Name>
            <TimeStamp>1232    0</TimeStamp>
            <UpdateTimeSta    pdateTimeStamp>
            <CXConfiguration    "50176" oid="36554">
```

Figure 4. Mapping textual and graphical representations.



```
<FSB>
    <CompoundFunction clsid="17920" oid="36551">
        <Name>Light_Control</Name>
        <TimeStamp>1232444430</TimeStamp>
        <Status>Not defined</Status>
        <CXConfiguration clsid="50176" oid="36552">
        </CXConfiguration>
        <Interface clsid="20224" oid="36553">
            <Name>Interface of Light_Control</Name>
            <TimeStamp>1232444430</TimeStamp>
            <UpdateTimeStamp>0</UpdateTimeStamp>
            <CXConfiguration clsid="50176" oid="36554">
            </CXConfiguration>
        </Interface>
        <ResponsibleGUID>8F5D0999-5B25-4519-BA91-F06C667D50CC</
        <Classification>Not defined</Classification>
        <SimulationTool>0</SimulationTool>
        <UpdateMarker>0</UpdateMarker>
        <Fixed>0</Fixed>
        <PosX>108</PosX>
        <PosY>0</PosY>
        <UpdateTimeStamp>745826403</UpdateTimeStamp>
        <FaultTrackingMethod>0</FaultTrackingMethod>
        <CompoundFunction clsid="17920" oid="33606">
            <Name>Alarm_Device</Name>
            <Comment>activate.</Comment>
```

Figure 5. XML Nodes that are not significant for variability.

The second block is a customized parser that generates a relevant lexicon from the set of software components within a project. The third block is a set of rules (viz., mandatory, optional, exclude) to govern the identification of variability.

The basic concept can be extended to obtain a working model for the identification of variants. The work flow is depicted in Figure 7. The top layer here represents the domain or core assets. The middle layer is a semi-automatic identification layer for variants. A component list and a

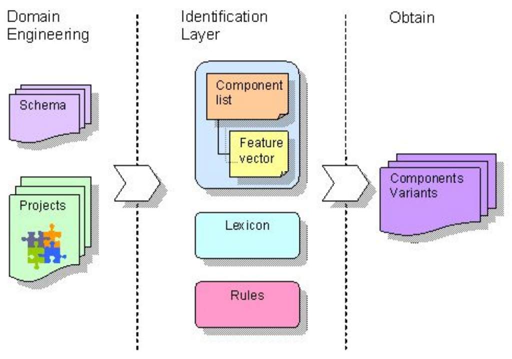


Figure 6. Basic Concept.

feature vector is derived manually from the schema of the project; a collection of elements that represent components and their descriptive features that significantly contribute to the identification of the component's variant.

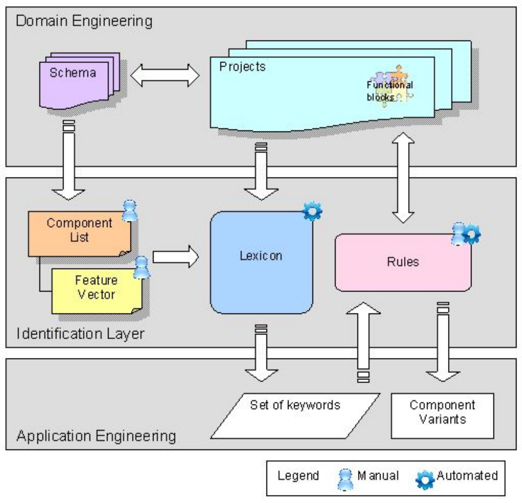


Figure 7. Work flow of the identification process.

The workflow can be further extended to adapt a heterogeneous environment which consist of projects developed using several modeling and simulation tools. The identification layer is separated into two parts. Numerous component lists and feature vectors can be derived for each distinct schema as depicted in Figure 8, whereas a common lexicon and common rules govern the identification process.

C. Evaluation

A prototype of the architecture presented here has been implemented. These case studies targeted the design of model-based software components firstly in an industrial use

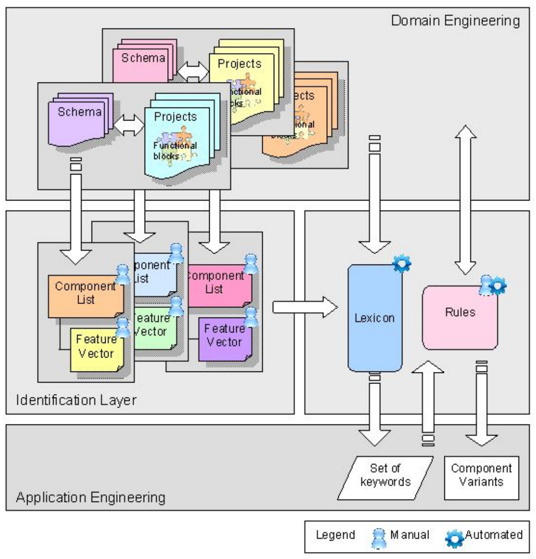
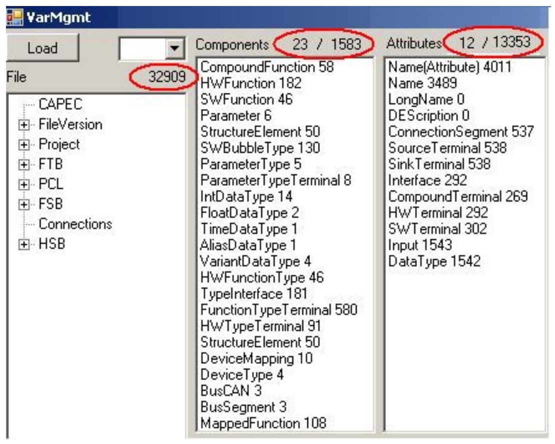


Figure 8. Work flow of the identification process for heterogeneous systems.

case where the project model was developed using the design tool ESCAPE® [17], and secondly in a case study targeting the execution of specific paradigms based on the naming convention of AUTOSAR® [18].

The specific project data set depicted in Figure 9, which was used to verify the implementation, consisted of a total of 32909 elements. Of these elements a total of 1583 elements signify components, these were categorized into 23 categories when enlisted in the component list. A total of 13353 elements signified features that were assigned into 12 categories.



32909

Components 23 / 1583

CompoundFunction 58
HWFunction 182
SWFunction 46
Parameter 6
StructureElement 50
SWBubbleType 130
ParameterType 5
ParameterTypeTerminal 8
IntDataType 14
FloatDataType 2
TimeDataType 1
AliasDataType 1
VariantDataType 4
HWFunctionType 46
TypeInterface 181
FunctionTypeTerminal 580
HWTypeTerminal 91
StructureElement 50
DeviceMapping 10
DeviceType 4
BusCAN 3
BusSegment 3
MappedFunction 108

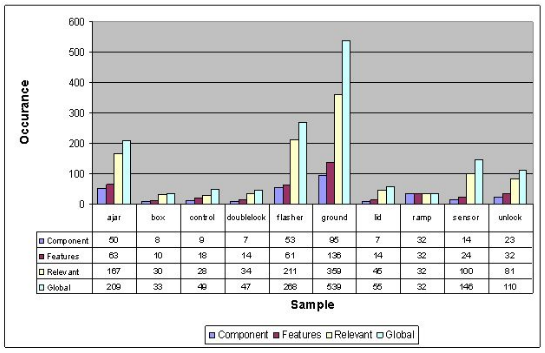
Attributes 12 / 13353

Name(Attribute) 4011
Name 3489
LongName 0
DEScription 0
ConnectionSegment 537
SourceTerminal 538
SinkTerminal 538
Interface 292
CompoundTerminal 269
HWTerminal 292
SWTerminal 302
Input 1543
DataType 1542

Figure 9. Dataset summary of project using ESCAPE design tool.

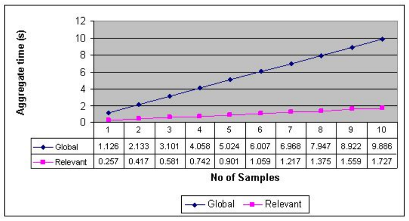
Three different approaches were adopted to evaluate and determine the performance with respect to matches and time.

- **Evaluation using a single element specification set**
  The first experiment was conducted on a single element specification set. A group of ten sets formed the input to determine the result set in both comprehensive (global) search and selective search as illustrated in Figure 10. The notion of comprehensive search is used, when scanning all occurrences of the specification set within projects, irrespective of whether they are components or features of those components. This can return a result set that contains false matches.



	ajar	box	control	doublelock	flasher	ground	lid	ramp	sensor	unlock
Component	50	8	9	7	53	95	7	32	14	23
Features	63	10	18	14	61	136	14	32	24	32
Relevant	167	30	28	34	211	359	45	32	100	81
Global	209	33	49	47	268	539	55	32	146	110

Figure 10. Occurrence graph for a single element specification set.



	1	2	3	4	5	6	7	8	9	10
Global	1.126	2.133	3.101	4.058	5.024	6.007	6.968	7.947	8.922	9.886
Relevant	0.257	0.417	0.581	0.742	0.901	1.059	1.217	1.375	1.559	1.727

Figure 11. Time graph for a single element specification set.

The pattern of the results displayed similar behavior.

Observations

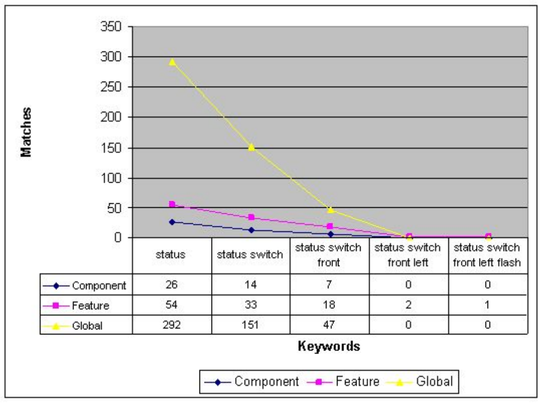
- The comprehensive search yields a result set that contains every occurrence of the specification set, even if these nodes do not characterize a component.
- The nodes representing components yield a result set which is somewhat realistic, though these do not epitomize the complete set desired. This is often observed when the component nodes do not match, but their features collectively match the specification set.
- These nodes along with the feature set yield a more elaborate result set. A match contained by any node in a set of features would result in representing the component to which it belongs.

Figure 11 depicts the time taken to obtain the specification set illustrated in Figure 10. The time graph depicts the aggregate time required for global and selective search for a set of ten specification sets.

Observations

- It is evident from these figures that the time required for comprehensive search exceeds the selective search - which is the method proposed in this article - by almost a factor of 5; this may be a dominant factor for large specification sets.

- **Evaluation using multiple element specification set**
  The second experiment was conducted using one up to seven element specification sets as a group illustrated in Figure 12.



	status	status switch	status switch front	status switch front left	status switch front left flash
Component	26	14	7	0	0
Feature	54	33	18	2	1
Global	292	151	47	0	0

Figure 12. Occurrence graph for multiple element specification sets.

Observations

- The comprehensive search often yielded large result sets, as it searches in individual nodes that are treated as atomic.
- The exhibited behavior is similar to the varying size of the specification set. As observed in Figure 12, the selective component-feature search result set demonstrates a value when the size of specification set exceeds 3, because in this case the matches take place across the boundary of the feature within the component. On the other hand the other methods return null result set as the search is only within the boundary of the element.
- For any given size of specification set, the selective component-feature search returns a much smaller result set and is more precise.
- Convergence is optimal with a specification set of size 3. If the size of the specification is too large the result may be null for both methods as shown in Figure 12.

- **Evaluation using different starting points for elements in specification sets**

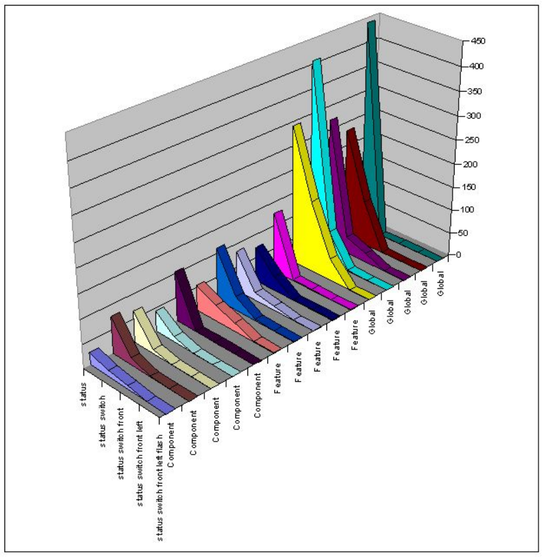


Figure 13. Occurrence graph for different starting points.

The third experiment was conducted searching for elements within specification sets using different starting points. Figure 13 depicts the result sets in comprehensive search and selective search.
To determine the effect of different starting points, a multiple-element specification set was used, where the orders of the elements were changed to obtain five sets. The result set for this exhibits the same pattern as the two experiments above.

VII. Conclusion

Managing variants is of utmost importance in today's large software bases as they reflect legal constraints, marketing decisions, and development cycles. As these software bases often grew from different sources and were developed by different teams using different tools it is in many cases very complicated if not nearly impossible to find artefacts that might be variants, both for historical reasons as for development purposes.

Searching algorithms have to reflect both the capability to match keywords and to reflect the structure that characterizes a component. Our proposed method is capable of both aspects and therefore helps the developer to find matches even in large and heterogeneous databases. In addition to that not only the required time for the search is a lot shorter, but also accuracy of the retrieved set of candidates is highly improved.

The developed prototype is itself independent of a specific tool as it works on textual descriptions that typically are available in XML.

References

[1] Ebert, C. and Salecker, J.; *Guest editors' introduction: Embedded software technologies and trends*. Software, IEEE, Vol 26(3): pp. 14-18, 2009

[2] Tanenbaum, A.S. and van Steen, M.; *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006

[3] Clements, P. and Northrop, L.; *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2007

[4] Ebert, C. and Jones, C.; *Embedded software: Facts, figures, and future*. Computer, IEEE Vol 42(4): pp. 42-52, 2009

[5] Gomaa, H. and Webber, D.L.; *Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model*. The Proceedings of the 37th Hawaii international Conference on System Sciences, 2004

[6] Oliveira, E., Gimenes, I., and Maldonado, J.; *A variability management process for software product lines*. CASCON 2005, The conference of the Centre for Advanced Studies on Collaborative research: pp. 225 - 241

[7] Heymans, P. and Trigaux, J.; *Software product line: state of the art*. Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur, 2003

[8] Galster, M. and Avgeriou, P.; *Handling variability in software architecture: Problem and implications*. WICSA 2011, Ninth Working IEEE/IFIP Confernce on Software Architecture: pp. 171-180

[9] PRODUCT LINE HALL OF FAME; *"http://splc.net/fame.html"*. retrieved: 04,2012

[10] Bachmann, F. and Clements, P. C.; *Variability in Software Product Lines*, Technical Report -CMU/SEI-2005-TR-012, 2005.

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

[12] Burgareli, L.A., Selma, Melnikoff, S.S., and Mauricio Ferreira, G. V.; *A Variation Mechanism Based on Adaptive Object Model for Software Product Line of Brazilian Satellite Launcher*, ECBS-EERC 2009, First IEEE Eastern European Conference on the Engineering of Computer Based Systems: pp. 24-31

[13] IEEE; *Iso/iec standard for systems and software engineering - Rrecommended practice for architectural description of software-intensive systems*. Technical report, IEEE, 2000

[14] Atkinson, C. and Kühne, T.; *A generalized notion of platforms for model-driven development*. Model-Driven Software Development, Springer-Verlag, Berlin: pp. 119–136, 2005

[15] Frank, A.A. and Brenner, E.; *Model-based Variability Management for Complex Embedded Networks*. ICCGI 2010, The Fifth International Multi-conference on Computing in the Global Information Technology: pp. 305-309

[16] Frank, A.A. and Brenner, E.; *Strategy for modeling variability in configurable software*. PDES 2010, The 10th IFAC workshop on Programmable Devices and Embedded Systems

[17] ESCAPE; *"http://www.gigatronik2.de/index.php?seite=escape_produktinfos_de&navigation=3019&root=192&kanal=html"*. retrieved: 04,2012

[18] AUTOSAR; *"http://www.autosar.org/download/conferencedocs/03_AUTOSAR_Tutorial.pdf"*. retrieved: 04,2012

Appendix A

Glossary

Distributed system: "A distributed system is a collection of independent computers that appears to its users as a single coherent system." [Tanenbaum and van Steen, 2006, p.2]

Embedded system: "Embedded systems are microcontroller-based systems built into technical equipment. They're designed for a dedicated purpose and usually don't allow different applications to be loaded and new peripherals to be connected. Communication with the outside world occurs via sensors and actuators; if applicable, embedded systems provide a human interface for dedicated actions.".[Ebert and Salecker, 2009, p.14]

Entity: "An object fundamentally defined not by its attributes, but by a thread of continuity and identity" [Evans, 2003, p.512]

Environment: "The environment, or context, determines the setting and circumstances of developmental, operational, political, and other influences upon that system. The environment can include other systems that interact with the system of interest, either directly via interfaces or indirectly in other ways. The environment determines the boundaries that define the scope of the system of interest relative to other systems." [IEEE, 2007, p.4]

Metamodeling: Kleppe [Kleppe, 2008] provides a definition of a model, which is rested upon a combination of a type graph and a set of constraints at various types of this graph. A type graph is defined as a combination of

- a set of nodes which may include data types
- a set of edges
- a source function from edges to nodes, which gives the source node of an edge
- a target function from edges to nodes, which gives the target node of an edge
- An inheritance relationship between nodes (a reflexive partial ordering)

The concept of a labeled graph is also applied in the representation of a class diagram, which is usually made use of for the presentation of elements in the M3 and M2 layer of the OMG four-level meta-model hierarchy. Having motivated the usage of graphs for the definition of a model Kleppe also defines an instance of a model M as a labeled graph that can be typed over the type graph of M and satisfies all the constraints in M's constraint set.

Platform: "A platform is the combination of a language specification, predefined types, predefined instances, and patterns, which are the additional concepts and rules needed to use the capabilities of the other three elements." [Kleppe, 2008, p.69]

Software Architecture: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEEE, 2007]

Software Component: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party." [Szyperski, 2002, p.41]

Software-intensive system: "A software-intensive system is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole." [IEEE, 2007, p.1]

Software product line: "Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisations." [Pohl et al., 2005, p.14]

System: A collection of components organized to accomplish a specific function or a set of functions. [IEEE, 2007, p.3]

Value objects: "An object that describes some characteristic or attribute but carries no concept of identity." [Evans, 2003]

Virtual Organization: "A collaboration whose participants are both geographically and organizationally distributed." [Foster and Kesselman, 2004, p.672]

Bibliography

[Atkinson et al., 2001] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., and Zettel, J. (2001). *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, 1st edition.

[Atkinson and Kühne, 2005] Atkinson, C. and Kühne, T. (2005). A generalized notion of platforms for model-driven development. In Beydeda, S., Book, M., and Gruhn, V., editors, *Model-Driven Software Development*, chapter 6, pages 119–136. Springer-Verlag, Berlin/Heidelberg.

[Autosar, 2011] Autosar (2011). Autosar. `http://www.autosar.org/download/conferencedocs/03_AUTOSAR_Tutorial.pdf`.

[B. Graaf, 2003] B. Graaf, M. Lormans, H. T. (2003). Embedded software engineering: The state of the practice. *IEEE Software, v. 20*, pages 61–69.

[Bachmann and Clements, 2005] Bachmann, F. and Clements, P. C. (2005). Variability in software product lines. *Technical Report -CMU/SEI-2005-TR-012.*

[Bachmann, 2005] Bachmann, P. C. C. F. (2005). Variability in software product lines. *Technical report, Software Engineering Institute,*.

[Bayus, 1994] Bayus, B. (1994). Are product life cycles really getting shorter? *Journal of Product Innovation Management, Vol. 11 (4)*, pages 300–308.

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

[Burgareli et al., 2009] Burgareli, L., Selma, S. S., Melnikoff, and Ferreira, G. V. M. (2009). A variation mechanism based on adaptive object model for software product line of brazilian satellite launcher. *First IEEE Eastern European Conference on the Engineering of Computer Based Systems.*

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition.

[Clements and Northrop, 2007] Clements, P. and Northrop, L. (2007). Software product lines: Practices and patterns.

[Crnkovic, 2005] Crnkovic, I. (2005). Component-based software engineering for embedded systems. *Proceedings of the 27th International Conference on Software engineering*, pages 712–713.

[E. Oliveira and Maldonado, 2005] E. Oliveira, I. Gimenes, E. H. and Maldonado, J. (2005). A variability management process for software product lines.

[E. Ostertag and Braun, 1992] E. Ostertag, J. Hendler, R. P.-D. and Braun, C. (1992). Computing similarity in a reuse library system: An aibased approach. *ACM Trans. on Software Engineering and Methodology 1(3)*, pages 205–228.

[Ebert and Jones, 2009] Ebert, C. and Jones, C. (2009). Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52.

[Ebert and Salecker, 2009] Ebert, C. and Salecker, J. (2009). Guest editors' introduction: Embedded software technologies and trends. *Software, IEEE*, 26(3):14–18.

[Ericsson and Erixon, 1999] Ericsson, A. and Erixon, G. (1999). Controlling design variants modular product platforms. *Society of Manufacturing Engineers, Dearborn, MI.*

[Evans, 2003] Evans, E. J. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, Amsterdam, 1. a. edition.

[Foster and Kesselman, 2004] Foster, I. and Kesselman, C., editors (2004). *The Grid: Blueprint for a New Computing Infrastructure (Elsevier Series in Grid Computing)*. Morgan Kaufmann, second edition.

[Frank van der Linden Klaus Pohl, 2005] Frank van der Linden Klaus Pohl, G. B. (2005). Software product line engineering. *Springer*, pages 5–20.

[Galster and Avgeriou, 2011] Galster, M. and Avgeriou, P. (2011). Handling variability in software architecture: Problem and implications. *2011 Ninth Working IEEE/IFIP Confernce on Software Architecture*, pages 171–180.

[GIGATRONIK, 2009] GIGATRONIK (2009). Escape. `http://www.gigatronik_2.de/index.php?seite=escape_produktinfos_de\&navigation=3019\&root=192\&kanal=html`.

[H. Mili and Mili, 1995] H. Mili, F. M. and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE Transaction on Software Engineering 21(6)*, pages 528–562.

[Heymans and Trigaux, 2003] Heymans, P. and Trigaux, J. (2003). Software product line: state of the art. *Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur.*

[I. Bird and Kee., 2009] I. Bird, B. J. and Kee., K. F. (2009). The organization and management of grid infrastructures. *Computer, 42(1)*, pages 36–46.

[IEEE, 2007] IEEE (2007). Iso/iec standard for systems and software engineering - recommended practice for architectural description of software-intensive systems. Technical report, IEEE.

[J. Bergey and Smith, 2000] J. Bergey, L. O. and Smith, D. (2000). Mining existing assets for software product lines. *Technical Note, CMU/SEI-2000-TN-008, SEI, USA.*

[Jan Bosch and Pohl, 2001] Jan Bosch, Gert Florijn, D. G. J. K. J. H. O. and Pohl, K. (2001). Variability issues in software product lines. *In van der Linden*, page 1321.

[Jonathan and Collard, 2005] Jonathan, I. M. and Collard, M. (2005). Adding structure to unstructured text.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (foda): Feasibility study. *CMU/SEI-90-TR-21, SEI, USA.*

[Kleppe, 2008] Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition.

[Kum et al., 2008] Kum, D., Park, G., Lee, S., and Jung, W. (2008). Autosar migration from existing automotive software. *International Conference on Control, Automation and Systems*, pages 558–562.

[Liggesmeyer and Trapp, 2009] Liggesmeyer, P. and Trapp, M. (2009). Trends in embedded software engineering. *Software, IEEE*, 26(3):19–25.

[M. Jazayeri and van der Linden, 2000] M. Jazayeri, A. R. and van der Linden, F. (2000). Software architecture for product families: principles and practice. *Addison-Wesley Longman Publishing Co., Inc.*

[Muller et al., 2009] Muller, P.-A., Fondement, F., and Baudry, B. (2009). Modeling modeling. In *Model Driven Engineering Languages and Systems*, chapter 2, pages 2–16. Springer Berlin / Heidelberg.

[N. Leveson, 2004] N. Leveson, K. W. (2004). Making embedded software reuse practical and safe. *Proceedings of the 12th International Symposium on Foundations of Software Engineering, ACM SIGSOFT '04/FSE-12, v. 29.*

[Northrop, 2002] Northrop, L. (2002). Sei's software product line tenets.

[Northrop, 2007] Northrop, L. (2007). Fourth product line practice workshop report. *Technical report, Software Engineering Institute. Carnegie Mellon University*, pages 3–20.

[OBrien, 2005] OBrien, L. (2005). Reengineering. *Carnegie Mellon University Pittsburgh, USA. ¡http://www.cs.cmu.edu/ aldrich/courses/654-sp05/handouts/MSE-Reeng-05.pdf¿.*

[Osman et al., 2010] Osman, A., Muthaiyah, S., Ibrahim, H. M., Amnuaisuk, S., and Ho, C. K. (2010). Defining variability in dss: An intelligent method for knowledge representation and validation. *Proceedings of the 43rd Hawaii International Conference on System Sciences.*

[P. Child and Wisniowski, 1991] P. Child, R. Diederichs, F. S. and Wisniowski, S. (1991). The management of complexity. *Sloan Management Review, Vol. 33 (1)*, pages 73–80.

[Paul C. Clements and McGregor, 2005] Paul C. Clements, Lawrence G. Jones, L. M. N. and McGregor, J. D. (2005). Project management in a software product line organization. *IEEE Software, 22(5)*, page 5462.

[Pohl et al., 2005] Pohl, K., Backle, G., and Linden, F. J. (2005). *Software Product LineEngineering: Foundations, Principles and Techniques.* Springer-Verlag, New York.

[Poole and Simon, 1997] Poole, S. and Simon, M. (1997). Technological trends, product design and the environment. *Design Studies, Vol. 18 (3)*, pages 237–248.

[Ramos and Penteado, 2008] Ramos, M. and Penteado, R. (2008). Embedded software revitalization through component mining and software product line techniques. *Journal of Universal Computer Science, vol. 14, no. 8*, pages 1207–1227.

[Simpson, 2004] Simpson, T. (2004). Product platform design and customization: Status and promise. *Artificial Intelligencefor Engineering Design, Analysis and Manufacturing, Vol.18 (1)*, pages 3–20.

[Stahl and Völter, 2006] Stahl, T. and Völter, M. (2006). *Model-driven Software Development: Technology, Engineering, Management.* Wiley.

[Szyperski, 2002] Szyperski, C. (2002). Component software: Beyond object-oriented programming. *2nd Edition, Addison-Wesley, USA.*

[Tanenbaum and van Steen, 2006] Tanenbaum, A. S. and van Steen, M. (2006). *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice Hall.

[Uira Kulesza and Borba, 2007] Uira Kulesza, Vander Alves, A. G. A. C. N. E. C. C. J. P. d. L. and Borba, P. (2007). Mapping features to aspects: A model-based generative approach. *Current Challenges and Future Directions, Lecture Notes in Computer Science*, pages 155–174.

[William C. Chu and He, 2005] William C. Chu, C. P. Hsu, C. L. and He, X. (2005). A semi-formal approach to assist software design with reuse.

[Ziadi, 2003] Ziadi, T.; Jzquel, J.-M. F. F. (2003). Product line derivation with uml. *In Proceedings Software Variability Management Workshop, University of Groningen.*