

Representing Variability in Software Product Lines: A Case Study

Michel Jaring and Jan Bosch

University of Groningen
Department of Mathematics and Computing Science
PO Box 800, 9700 AV Groningen
The Netherlands
{m.jaring, bosch}@cs.rug.nl
<http://www.cs.rug.nl/Research/SE>

Abstract. Variability is the ability to change or customize a software system, i.e., software architects anticipate change and design architectures that support those changes. If the architecture is used for different product versions, e.g., in a software product line context, it becomes important to understand where change has to be planned and the options possible in particular situations. Three variability issues have been identified in a case study involving a software company specialized in product and system development for professional mobile communication infrastructure. These issues are discussed and analyzed and illustrate the need for handling variability in a more explicit manner. To address this need, this paper suggests a method to represent and normalize variability in industrial software systems. The method is exemplified by applying it to the software product line of the aforementioned company.

1 Introduction

The notion of constructing software systems by composing software components and a family of software products is not recent [23][25]. Combining both concepts has resulted in approaches towards software design that share the ability to delay design decisions to a later phase in the development process [19][20]. The recent emerge of software product lines provides an example of delayed design decisions. The core idea of software product line engineering is to develop a reuse infrastructure that supports the software development for a family of products. The differences between the products express variation or variability, i.e., it is a form of diversification.

The relationship between a family of products, e.g., a software product line, and variability has been identified in both research [1][10] and industry by, among others, the co-author of this paper [7]. However, insight into this relationship is still limited and the approach towards variability tends to differ in both areas of expertise. In general, the focus in research is to establish a commonly accepted concept of variability, whereas industry focuses on how to handle variability. Since architects have been able to handle variability in large-scale software systems in the past, it is not unlikely that practice may lead theory at this point [15]. To establish a basis for

understanding variability, it is necessary to investigate the current state of practice and the questions that arise in industry.

Several authors have reported on experiences with evolutionary software systems in an industrial context, e.g., [12][22][9], but they do not explicitly address variability. In addition, most of this work primarily reports on large, American software companies that are often related to defense industry and is therefore not necessarily representative for small and medium sized enterprises.

This paper focuses on variability in industrial software product lines. The issues that were identified in a case study involving a Dutch-based company are discussed and analyzed and illustrate the need for handling variability in a more explicit manner. This company, Rohill Technologies BV, is a worldwide operating organization specialized in product and system development for professional mobile communication infrastructure such as radio networks for police and fire departments. In our opinion, the organization is representative for small and medium sized enterprises in software industry (SMEs), i.e., a development department of approximately fifteen developers and products that are sold to industry and public organizations. The contribution of this paper is, we believe, threefold, it:

- discusses an exemplar of an industrial software product line,
- presents variability related issues and
- suggests a method to represent and normalize variability.

The next section discusses variability with a focus on software product lines and provides terminology based on the work of others. Section 3 describes the case study and discusses the issues that have been identified. A method to represent and normalize variability is then suggested and exemplified by applying it to Rohill's software product line in section 4. Section 5 surveys related work and the paper is concluded in section 6.

2 Variability in Industrial Software Systems

A common goal in software engineering is to prepare software for change, especially when an architecture for a family of products is designed [10]. A software architect will try to predict forthcoming changes, i.e., variations, and prepare the architecture accordingly. A boxes-and-lines diagram is sometimes used to document the architecture, where the implicit assumption is that the boxes are the points of variation [1]. However, a single variation is most often spread across multiple components due to interconnections, but a relation between two or more components is usually not documented with regard to variation. In other words, the relation between components is known, but the implications of variation through this relationship are not. In addition, it is likely that changes occur in the lifecycle of an architecture that have not been anticipated in advance. If an architecture has to support unplanned changes, the dependencies among components will typically increase, which makes it harder to understand the impact of a change. Furthermore, the creators of an architecture are not necessarily the adaptors [11]. The architect generally has a concept for what to do when a change actually occurs and also understands the implications of the change.

When this architect is involved in the analysis of a required variation, the result is probably sufficient [1]. However, in industrial software engineering, the adaptors of the architecture are most often not the creators. Consequently, adapting software architectures demands for proper documentation, methods, techniques and guidelines to handle change, i.e., to handle variability. Following this line, it also requires that all parties involved (customer, development, management, etc.) observe and identify variability from the same frame of reference.

The intention of identifying variability is to delay design decisions to a later phase in the software development process, i.e., to promote reuse. Recently, delaying design decisions towards later phases in the development process has received attention in both research and industry [18][21]. A software system that dynamically adapts its run-time behavior relevant to its environment by either selecting embedded alternatives or by binding modules is a typical example of a delayed design decision. The software architecture of such a product is said to be dynamic, i.e., it is neither static during development nor at run-time. Dynamic software architectures [24] are an extreme form of variability, i.e., it is not possible to delay design decisions any further. Even a conditional expression is sometimes considered as a form of variability, i.e., virtually all software architectures exhibit a certain dynamic behavior or flexibility.

Variability is generally expressed in terms of variation points. A variation point refers to the aforementioned delayed design decision, i.e., it indicates a specific point in the development or deployment phase of a software system. A typical example of variation points is the pre-processor directive shown in Fig. 1.

```
#define EMBEDDED /* SIMULATION */

#ifdef SIMULATION
    #include <stdio.h>
#endif

#ifdef EMBEDDED
    #include <drivers.h>
#endif
```

Fig. 1. Selecting embedded or simulation mode indicates a variation point, e.g., simulation mode does not require hardware drivers, but standard I/O instead

A software system recognizes a number of development phases and each phase relates to its own representation, e.g., requirement collection relates to requirement specification, whereas implementation relates to source code. As explained in [18], software development consists of transformations of these representations and each phase can be regarded as a different abstraction level of the system. Design decisions are taken during each transformation, but some decisions are deliberately left open to support variability. These open design decisions are actually delayed design decisions, i.e., variation points. A variation point is introduced at a particular phase in the software development process and bound at this phase or at phase further on in the software lifecycle. See also Fig. 2.

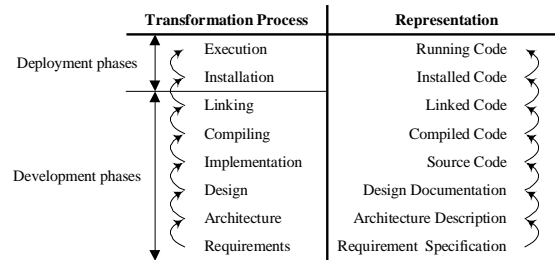


Fig. 2. Transformation and representation processes; the software lifecycle is a combination of the development and deployment phases. The arrows are unidirectional, but traversing back and forth through the phases is possible (e.g., in case of iterative software development)

In this paper, the lifecycle of a software system consist of all transformation processes. The intention of designing a variation point into a system is to insert a variant (alternative) at a later phase in this lifecycle. Each design decision constrains the number of possible systems that can be built [20], i.e., during early development there are many open design decisions. However, these decisions have not been deliberately left open and they therefore do not refer to a particular point in the lifecycle of a software system. From this perspective, a variation point is in one of the following three states [18]:

- Implicit: the design decision is not deliberately left open.
- Designed: the design decision is deliberately left open.
- Bound: a variant (drawn from a set of variants) related to the variation point is inserted into the system.

Each variation point relates to a set of variants that can be bound to it. With respect to this set, the following distinction between variation points is made [18]:

- Open variation points: the set of variants can be extended.
- Closed variation points: the set of variants can not be extended.

For example, a range of wireless modems may support two profiles that aim for two groups of users. Technical users such as wireless service provider personnel who are interested in debugging problems using their wireless communications service and non-technical users who do not require detailed information (they are only interested in general information about the current connection). The components that implement these profiles are the available variants to bind this designed variation point. The variation is, e.g., introduced at the design phase and bound at run-time. If the two profiles are the only variants possible, the variation point is closed.

2.1 Variability in Software Product Lines

A typical example of delayed design decisions is found in software product lines. The goal of the software product line approach is the systematic reuse of core assets for building related products, i.e., by developing a product line instead of single products, a more efficient development and maintainability process is anticipated [2][27].

Systematic methods with a focus on software product line engineering such as the Family-Oriented Abstraction, Specification and Translation (FAST) process described in [31] begin to emerge. In [6], the co-author of this paper describes a software product line as follows. A software product line typically consists of a product line architecture, a set of components and a set of products. Each product derives its architecture from the product line architecture, instantiates and configures a subset of the product line components and usually contains some product specific code. The IEEE Standard 1471 defines architecture as [13]: “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.” In [3], the software architecture of a program or computing system is defined as: “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

In general, there are two relatively independent development cycles in organizations that use software product lines, i.e., product line development (domain engineering) and product instantiation (application engineering). Product instantiation is the process of creating a specific product that is based on a software product line. Domain engineering is responsible for the design, development and evolution of the reusable assets, i.e., the product line architecture and shared components. Application engineering, on the other hand, relates to adapting the product line architecture to fit the system architecture of the product in question. In other words, the product line domain is divided in reusable and non-reusable assets, i.e., domain and application engineering, respectively. This approach is also taken in, e.g., [19].

Variability in software product lines can be observed in two dimensions [8], viz. space (e.g., software artifacts appear in several products) and time (i.e., software artifacts change over time). Features are suggested as a useful abstraction to express variability [18] in both dimensions, i.e., to describe the differences between products. In [6], a feature is defined as a logical unit of behavior that is specified by a set of functional and quality requirements. The underlying idea is that a feature groups related requirements, i.e., they abstract from requirements. Features can be seen as incremental units of development [16] and dependencies make it possible to link them to a component or class. Features that are associated with (depend on) other features are so-called crosscutting features. Crosscutting features make it difficult to properly implement variability and generally demand a systematic and reproducible approach towards software system design.

3 Case Study: Rohill’s TetraNode Software Product Line

This section details a case study conducted at Rohill Technologies BV and provides an example of an industrial software product line. The variability issues that have been identified in the study are discussed and analyzed. In the next section, the study is used to exemplify a theory concerning variability representation.

3.1 Case Study Methodology

The case study methodology is, among others, based on [29] and [30]. The goal of this study is to exemplify relevant variability issues that may arise in small and medium sized enterprises in software industry (SMEs), i.e., development departments of five to fifty developers and products that are sold to industry and public organizations. The objectives deriving from this goal are:

1. An analysis of variability in an industrial software product line.
2. Provide a context to verify and validate theories on variability.

Relevant to the organization in question, the research questions arising from the above objectives were as follows:

- Where and in what form occurs variability in the software lifecycle?
- What variability issues are applicable?

Several variability concerns are discussed in [8] by, among others, the co-author of this paper. Based on previous studies (not focused on variability), e.g., [4][7][9], it was expected that a number of them would apply to the organization under consideration. However, suitable literature related to variability in SMEs is limited, if not unavailable. In our opinion, the most appropriate method to answer the research questions was through interviews with key persons in the organization. The interviewed party, Rohill Technologies BV [26], develops hardware-dedicated software (object oriented, C++) for a range of products and employs approximately fifteen developers with an electrical engineering background. This organization is therefore, we believe, representative for a larger category of SMEs.

We conducted interviews with the chief architect and the main developer. The interviews were preceded by an introductory talk (at least a week prior to the interviews) to explain the reason of the interviews and to itemize the topics to be discussed. This allowed the interviewees to prepare themselves, e.g., to collect the architectural overviews to be shown during the interviews. A questionnaire was used during the interviews to guide the process and the interviewee could elaborate if necessary. Only individuals were interviewed to increase the possibility for having in-depth discussions and to ensure that the interviewee could speak freely. The results of the interviews were discussed with the interviewees who then provided feedback to prevent misunderstandings. Additional information has been collected from company documents and by having short talks with the developers in the company. The latter was particularly useful to verify the statements of the interviewees.

3.2 Rohill Technologies BV

3.2.1 Company Background

Rohill started in 1976 with the sales of conventional private mobile radio equipment to the domestic market. Nowadays, Rohill is a worldwide operating organization specialized in product and system development for professional mobile communication infrastructure such as radio networks for police and fire departments (export accounts for 90 per cent of their sales). Their main activities include development and integration of trunked mobile radio systems according to mandatory

standards (trunking is the automatic and dynamic allocation of a small number of radio repeaters among many users). In the early '90s the past had proven that professional communication users require specialized, tailor-made systems to maximize the result of their organization. The company therefore focused on research and development and contracted out most of the actual hardware production to so-called original equipment manufacturers. Their products are now tailored according to customer requirements by modifying a reusable soft- and hardware infrastructure.

3.2.2 TETRA versus GSM

Standard wireless communication solutions are mainly based on GSM (Global Standard for Mobile telecommunication) technology. However, the market not covered by GSM solutions requires customer specific solutions, i.e., application specific development is necessary. Most of the specialized products are related to trunked mobile radio systems that are compliant to several standards such as TETRA (Trans-European Trunked Radio). The TETRA standard is defined by ETSI (European Telecommunications Standards Institute) that joins the interests of network operators, national administrations, equipment manufacturers and users. ETSI publishes telecommunication standards that are mandatory for use in Europe, but also widely applicable outside Europe (this is the case with, among others, GSM). GSM is primarily a telephony standard for public cellular telephony, whereas TETRA is a professional mobile radio standard for demanding markets such as railway and airline companies, fire and police departments, etc.

The TETRA and GSM standard overlap, but there are fundamental differences in the functional and quality requirements. The transmission architecture for GSM is highly hierarchical and optimized for central delivery and call validation is performed on each transaction. The consequences are long call set-up times, high transmission costs and limited inherent resilience, i.e., a single system failure can cause major disruption in services. TETRA, on the other hand, relies on an intelligent, distributed network architecture optimized for local delivery and call validation performed per session. Such an architecture results in low transmission costs, fast call set-up times, inherent system resilience and is scalable (with regard to the GSM transmission architecture).

3.2.3 TetraNode Software Design Model

The trunked mobile radio product range developed by Rohill is referred to as TetraNode. It is an implementation of the TETRA standard and the focus of the case study. The general architecture of TetraNode does not use the hierarchical model of the traditional (mobile) telephone network. Instead, it uses a model similar to the Internet. A central switch is therefore not necessary and reliability is increased due to redundant pathways and distributed functionality. The protocol for data transport is based on the TCP/IP standard, which allows adding new nodes to the network to extend its coverage. Fig. 3 depicts the overall decomposition of the software design model of TetraNode in terms of components and subsystems.

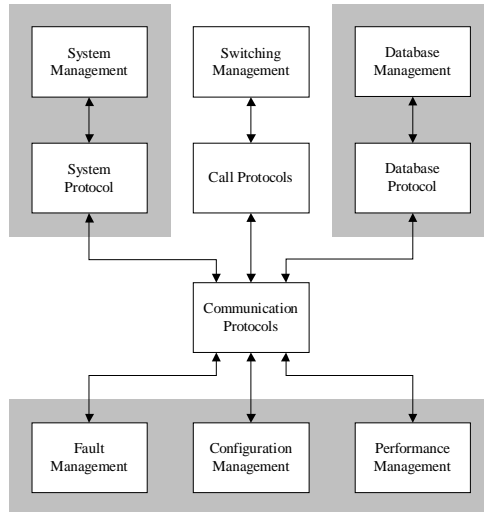


Fig. 3. Software design model of the trunked mobile radio system (TetraNode). The shaded boxes indicate related subsystems of functional modules

3.2.4 Quality Issues

The high level of quality demanded by the professional telecommunications market and regulatory bodies requires a quality system that relates to the organization, products and services. For the development and implementation of the adopted quality system the company refers to the ISO 9004-1 guidelines. The actual production is contracted out to ISO 9002 certified sub-contractors. The specifications and procedures for quality assurance in final inspection and testing have been documented, i.e., each product has to meet the relevant requirements with respect to safety, durability, liability, type approvals and ETSI regulations. Over the last few years, several large companies such as Dutch PTT, AEG Germany and GEC Marconi England have successfully audited Rohill according to the ISO 9000 model.

3.3 Rohill's Software Product Line

3.3.1 Software Architecture

As illustrated in Fig. 4, Rohill applies a layered software architecture that allows building applications according to the software product line approach. Three TetraNode products are aimed for, viz. the professional, medium and minimal solution. This range of products is characterized by differences in both soft- and hardware that are anticipated prior to implementation. Specific customer requirements are met by adapting one of these three products after product instantiation. For example, a Dutch electricity supply company requested a trunked radio network that would allow radio communication in case of power supply, radio channel, site or switch failure. An appropriate product was selected from the product line and adapted by adding additional components that implement the required functionality.

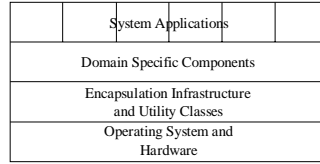


Fig. 4. Layered software architecture of TetraNode

The general architecture of TetraNode is designed and maintained by the architect. In addition, the architect also controls the header files and reviews the source code directly related to the quality requirements of the system. Documentation such as Architecture Design Document, Quality Assurance Documents, etc. is available.

3.3.2 TetraNode Software Product Line

The TetraNode product line is based on single software architecture that is reduced to yield one out of the three products, i.e., it is a software product line according to the so-called maximalist approach. In other words, the product line architecture supports all (hardware related) features and disabling specific features composes the professional, medium or minimal solution. As illustrated in Fig. 5, a product is selected from the product line and, if necessary, adapted to meet additional requirements. The product line and product specific lifecycle refer to domain and application engineering, respectively. In other words, change occurs before and after product instantiation. Change before product instantiation relies on the reuse infrastructure, whereas change after instantiation relies on non-reusable code.

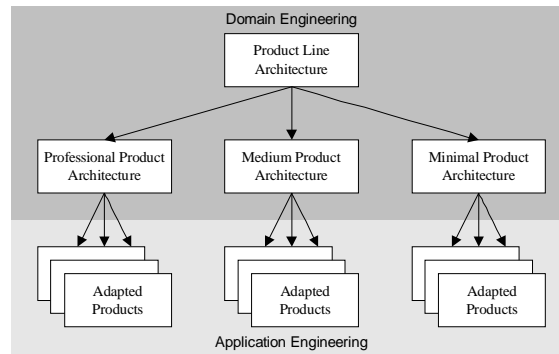


Fig. 5. TetraNode product line architecture, product architectures and derived products

An object-oriented approach is used, mainly to facilitate program modifiability (C++). The subsystems and functional modules in Fig. 3 are ultimately decomposed into classes and class utilities. A developer is assigned to one or more classes and is responsible for the functionality of these classes. Classes that are (partially) implemented are extensively tested on the target hardware and, if necessary, adapted in an iterative manner. Tests are performed by Rohill, but also by regulatory bodies to meet specific demands concerning, e.g., reliability and radio transmitter power. In general, an internal software release is available for testing every few months.

3.3.3 System Size

As opposed to their hardware platform, Rohill's software is proprietary and based on the TetraNode Foundation Classes (TFC). These classes are developed internally and provide a library for protocols, routing, security, etc. The TetraNode system has been under development for four years and the TFC has recently been finished. The remainder of the system is still under development and the total size of the system is expected to be 100 KLOC (kilo lines of code). Interestingly, the TFC implements almost all functionality, but its size is about 30 KLOC. This implies that the code currently being developed is focused on selecting the appropriate features from the TFC to create the product line. In other words, the dependencies among features require more programming than programming the features themselves. This is probably due to the dependencies of crosscutting features.

3.4 Variability in the TetraNode Software Product Line

As illustrated in Fig. 6, variability is identified before and after product instantiation and, as opposed to variability before instantiation, variability after instantiation is most often not anticipated. It is expected that 80 per cent of the customers require a product with standard functionality, i.e., a product directly selected from the product line will generally suffice.

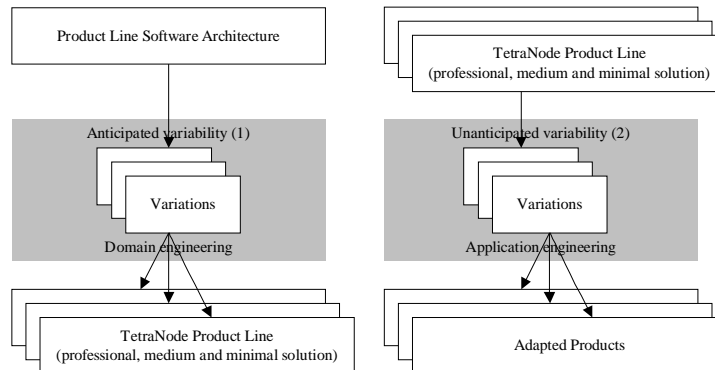


Fig. 6. Variability is identified before (1) and after product instantiation (2). The variability in the first layer is about targeting the appropriate hardware and the associated functionality. Unanticipated variability in the second layer addresses the need for customer specific solutions

Variation points in the first layer of variability are introduced at the design phase and bound at run-time by means of a license key (the software architecture supports all features and the license key disables features that are not needed). The license key is primarily based on dynamic linked libraries (DLLs). DLLs provide a way to separate the functionality of an application, i.e., it is a collection of small programs, which can be called when needed by the executable program that is running.

The second layer of variability introduces variation points at the requirement specification phase and binds them at the implementation or link phase. This layer refers to customer specific requirements and is primarily based on makefiles and product specific code. A makefile defines those files that are needed to compile a program and how those programs are to be compiled. Product specific code is source code that only appears in a single product, i.e., it is not to be reused.

3.5 Variability Concerns in Rohill

The case study underpins several variability concerns as discussed in [8]. The following concerns were identified as relevant with respect to the organizational and technical aspects of Rohill:

- Mechanism: a means to describe differences between products, e.g., pre-processor directives and makefiles.
- Phase: particular phases in the lifecycle introduce and bind variation points, e.g., a variation point is introduced during design and bound during implementation.
- Representation: a notation format to describe variability.
- Dependencies: the relationships between variability and features.
- Tool support: the means to maintain variability information, i.e., the number of variation points, associated variants and dependencies.

Table 1 lists how these concerns are handled in Rohill.

Table 1. Variability concerns in Rohill from the perspective of domain engineering (DE) and application engineering (AE)

Mechanism	DE	DLLs
	AE	Makefiles and product specific code
Phase (with respect to the majority of variation points)	DE	Introduced at design phase Bound at run-time
	AE	Introduced at the requirement specification phase Bound at the implementation or link phase
Representation	DE	No explicit representation available, the architect implicitly discerns variability
	AE	No explicit representation available, the architect and the developers implicitly discern variability
Dependencies	DE	The architect implicitly discerns dependencies
	AE	The architect and the developers implicitly discern dependencies
Tool support	DE	No tool support available
	AE	No tool support available

3.6 Variability Issues in Rohill

Three issues regarding variability have been pinpointed in Rohill, viz. variability identification, variability dependencies and tool support. Each issue is a combination of variability concerns and discussed together with an example taken from Rohill.

3.6.1 Variability Identification

- **Mechanism.** Typically, variability is mapped to a single mechanism, which particularly simplifies the design phase with a risk of obstructing future maintenance. The process of selecting a variability mechanism is often based on earlier experience, but the associated costs and impact are not always understood. If a variation point has to be bound at another moment in the lifecycle, a redesign to replace the current variability mechanism is necessary. The consequences of a particular variability mechanism on resource usage are generally not clear until run-time and changing the mechanism at this phase, if possible, is probably costly.
Example. Three variability mechanisms are used in Rohill, i.e., DLLs, makefiles and product specific code. The first applies to instantiating the product line, whereas the latter two apply to adapting the products that are selected from the product line to meet customer specific requirements. The mechanisms selected by Rohill are in accordance with the skill level of the organization. For example, pre-processor directives were also used in the past, but after slightly reorganizing the development process (i.e., increasing the number of code reviews), this particular mechanism was replaced by makefiles. In other words, the software system has been made more flexible as soon as the organization was able to do so. The architect considers the consequences of selecting a particular mechanism with respect to resource usage and underlying dependencies in an implicit manner. Based on his experience and insight into the system, he determines how variability is handled to support the required functionality and future maintenance.
- **Phase.** Variation points are introduced and bound at particular phases in the lifecycle and selecting the optimal phase for each of these activities has a considerable impact on the flexibility of the system as well as on the development and maintenance costs. If a variation point is bound too early, flexibility of the product line artifacts is less than required. Binding it later than necessary typically results in a more costly solution. Identifying variability is often based on analyzing commonalities and differences between products, but actually specifying the optimal phases is difficult due to lacking methods, techniques and guidelines.
Example. Methods, techniques and guidelines for optimal phase selection are not used in Rohill. Phase selection is primarily based on the experience of the architect who takes product and organizational characteristics such as technical aspects, competence and business considerations into account. This may explain why variability tends to accumulate at certain phases in the lifecycle, i.e., phase selection is probably related to the cultural facets of the organization (political coalition). To meet customer specific requirements after product instantiation, different phases are selected at the beginning of the lifecycle to introduce and bind variation points. In other words, customer specific design decisions appear earlier in the lifecycle than design decisions related to the reuse infrastructure.

3.6.2 Variability Dependencies

- **Representation.** A notation format to describe variability is not available and insight into variability is therefore difficult to obtain. In addition, the perception of variability often depends on the organization in question and the area of expertise of the people involved. There is a need for a common frame of reference, i.e., a system independent notation to describe variability.

Example. In Rohill, variability is observed as the ability to deliberately form different products in a repeatable manner. As a result of developing products according to standards like TETRA, the differences between products are well documented, especially from a hardware perspective. Since most variability is related to the target hardware, variability and dependencies are mainly observed from a hardware perspective. The architect manages and implicitly represents variability from this perspective, which implies that (in-depth) knowledge on the product line technology is necessary to discuss variability in Rohill. In other words, Rohill's frame of reference concerning variability is technology-based.

- **Dependencies.** In general, the dependencies between variability and features are not made explicit, i.e., they are (indirectly) implied, not documented and sometimes even unintended. As a result, it is not always clear what assets should be selected to compose a product with the required functionality. This problem is strongly related to the problem of variability representation, i.e., to describe the dependencies it is, in the first instance, necessary to describe variability.

Example. Rohill develops a software architecture that supports all features of the product line. This architecture is then reduced to yield one out of three product architectures (the maximalist approach). Removing functionality from an architecture results in fewer dependencies, i.e., the complexity decreases. Despite the fact that the architect manages all dependencies, this approach works well in Rohill. However, managing all dependencies by just one person is more than likely to become difficult, if not impossible, if the size of the system increases.

3.6.3 Tool Support

- The number of variation points, the associated variants as well as the dependencies tend to increase rapidly. Consequently, tool support for maintaining variability information is necessary, especially in case of large-scale software systems. However, current software configuration and management tools have not been developed from a lifecycle perspective and lack many required characteristics, i.e., most tools refer to a particular development phase and not to interacting transformation processes.

Example. The architect manages most of the variability in Rohill and, strictly speaking, tool support to maintain variability information is not mandatory at the moment. However, when the size of the system increases, proper tool support is probably beneficial, if not necessary (especially in case of the maximalist approach). In addition, it may make Rohill less dependent on key persons, since it would add explicit knowledge on variability to the organization.

3.7 A Common Understanding of Variability

The three issues that have been identified in the study are (indirectly) related to each other, i.e., tool support implies insight into variability dependencies, which in its turn relies on variability identification. In addition, these issues are characterized by a lacking frame of reference; how variability is observed often depends on the organization in question and the issue at hand. To handle variability, a common understanding of variability is necessary, which may support methods, techniques, tools and guidelines that are needed by software architects and developers. Several research communities study this problem from their own perspective, e.g., object oriented frameworks, maintainability and configuration management, etc. However, the common denominator is, in our opinion, variability representation, which implies an integrated approach towards finding a common frame of reference.

It would be beneficial if different areas of expertise such as engineering and management could refer to the same representation of variability. However, this requires a frame of reference that does not assume in-depth knowledge of the software system in question, but, at the same time, it should provide a proper level of information. A common understanding of variability may pave the way for identifying the rate of variability of (parts of) a software system, which would make it possible to decide on variability in a more explicit manner. The next section suggests a representation and normalization of variability, i.e., a frame of reference.

4 Variability: A Frame of Reference

This section suggests a representation and normalization of variability, i.e., a first step to address the need for a system independent frame of reference. The theory is exemplified by applying it to Rohill's software product line.

4.1 Representing Variability

There are many ways to support variability, ranging from conditional statements, pre-processor directives and inheritance to run-time selection of the appropriate components. Such variability realization techniques are diverse, but characterized by their introduction and binding phase in the software lifecycle. To represent variability, we classify variability realization techniques according to these phases by introducing variability sets. A variability set relates to a particular combination of an introduction and binding phase and each variability realization technique with an introduction and binding phase that concurs with this combination is a member of this set. In other words, a variability realization technique is a member of exactly one variability set and a variability set may consist of zero or more variability realization techniques. As shown in Fig. 7, grouping variability realization techniques according to their introduction and binding phase forms a two-dimensional, triangular space.

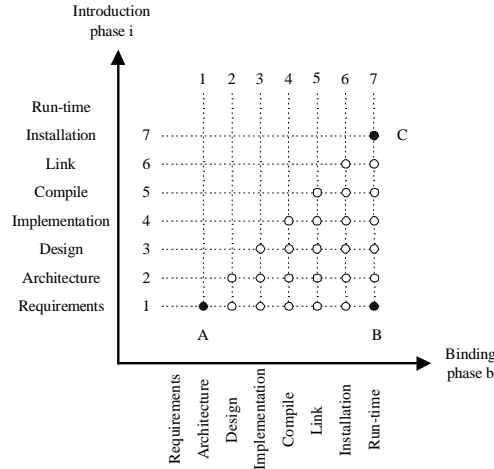


Fig. 7. A variability set is represented by a single, discrete point in a fixed-size space. The combination of the introduction and binding phase of the variability realization techniques included in a variability set specifies the coordinates of this set

In the context of this paper, the variability space comprises 28 variability sets. Set A indicates minimum variability, whereas set C indicates maximum variability. Set B and C share the same binding phase, but they have different introduction phases. We have taken the line that set C indicates maximum variability, e.g., set C is more variable than set B. As said, a design decision, whether it is delayed or not, constrains the number of possible systems that can be built. When a variation point is introduced, the options possible are limited for all subsequent phases, regardless of whether the variation point is bound. For example, in Rohill, the differences between the professional, medium and minimal solution are introduced at the design phase, but product instantiation is at run-time. The phases in between have to take these differences into account, which limits the number of options, i.e., it is less variable.

4.2 Normalizing Variability

To suggest a normalization of variability independent from the system under consideration, we assume the following with respect to Fig. 7:

- A variability set relates to a certain rate of variability.
- The variability realization techniques associated with a particular variability set relate to the same rate of variability.
- Set A indicates minimum variability, whereas set C indicates maximum variability.

The area covered by a two-dimensional space is commonly used in various fields to express a normalized measure. For example, the instantaneous power (work per unit time) consumed by a resistor is simply $P=V \cdot I$. For V in volts and I in amperes, P comes out in watts. Both V and I can be mapped on a relative scale ranging from, e.g., 0 through 1 to yield a normalized value for the consumed power.

Considering that the axes in Fig. 7 provide a normalized scale from 0 through 7, the area of variability space covered by a particular combination of an introduction and binding phase provides a relative measure for variability. The area of variability space covered by a variability set is

$$A(i, b) = i \cdot b - \frac{i^2}{2}, \quad (1)$$

where i is the number of the introduction phase and b the number of the binding phase (the second part of the formula is due to the triangular shape of the variability space). The relation between variability sets and variability space is depicted in Fig. 8. Note that the y-axis ranges from 0 through 100 per cent (24.5 absolute).

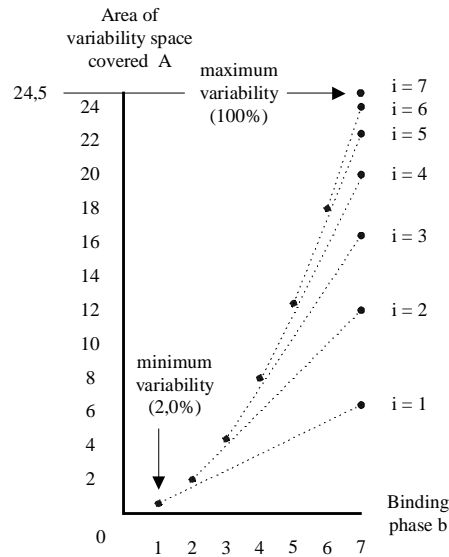


Fig. 8. Accumulation of variability space covered per introduction phase i . For clarity, only the extreme values for each introduction phase are depicted

4.3 Case Study: Representing Variability in Rohill's Software Product Line

In Rohill, products that are directly instantiated from the product line account for approximately 80 per cent of the products shipped. Instantiation is based on a license key (DLLs). The remaining products are customer specific, i.e., these products are adapted after instantiation. In general, makefiles account for approximately 5 and product specific code for 15 per cent of the variability supported in customer specific products. See also Table 2.

Table 2. Variability realization techniques in Rohill's TetraNode product range, the associated introduction and binding phases and the ratio of products that is based on these techniques

Variability Realization Technique	Introduction Phase	Binding Phase
A: DLLs (80%)	Design	Run-time
B: Makefiles (5%)	Requirement specification	Link
C: Product specific code (15%)	Requirement specification	Implementation

Fig. 9 exemplifies the theory of representing variability graphically. The ratio between the different variability realization techniques corresponds with the diameter of the solid points in the figure.

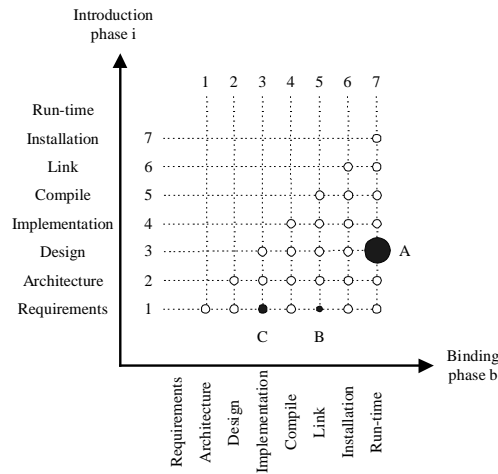


Fig. 9. Graphical representation of variability in Rohill's TetraNode product range. A: DLLs (product instantiation, 80%). B: makefiles (customer specific, 5%). C: product specific code (customer specific, 15%)

As shown in Table 3, calculating the area covered according to equation 1 for A, B and C, respectively, yields normalized variability values. The maximum, normalized value for variability is 24,5 when $i = 7$ and $b = 7$ (the total area of variability space).

Table 3. Normalized values for the variability supported in Rohill's TetraNode product range. These values indicate for each realization technique separately how variable they are, i.e., 0 per cent implies no variability, whereas 100 per cent implies maximum variability

Variability Realization Technique	Normalized: absolute	Normalized: relative (%)
A: DLLs ($i = 3, b = 7$)	16,5	67
B: makefiles ($i = 1, b = 5$)	4,5	18
C: product specific code ($i = 1, b = 3$)	2,5	10

The normalized values are plotted in Fig. 10. Moving from one line to another (in this case from the line associated with $i=1$ to the line associated with $i=3$) implies a nonlinear increase of variability.

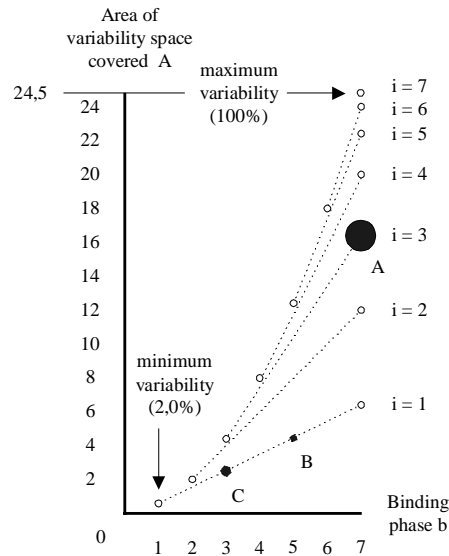


Fig. 10. The variability realization techniques in Rohill's TetraNode product range. The position of a solid points indicates the rate of variability, whereas its size indicates the rate of use. A: DLLs (product instantiation, 80%). B: makefiles (customer specific, 5%). C: product specific code (customer specific, 15%)

Taking into account that the entire product range (including customer specific products) is based on point A, the total rate of variability for Rohill's software system is $((100 \cdot 16,5) + (5 \cdot 4,5) + (15 \cdot 2,5)) / 120 = 14,3$.¹ This yields a relative, normalized variability of $(14,3 / 24,5) \cdot 100\% \approx 58\%$. This measure provides an insight into the variability supported by Rohill's software. For example, makefiles have replaced the pre-processor directives in the past, i.e., the relative, normalized variability used to be $((100 \cdot 16,5) + (5 \cdot 3,5) + (15 \cdot 2,5)) / 120 = 14,2 \approx 58\%$. In other words, replacing preprocessor directives by makefiles has hardly affected the variability in Rohill's product range. Suppose that the variation points related to product instantiation are introduced at the implementation instead of the design phase (point A is then positioned at the end of the line $i=4$ in Fig. 10). The absolute area of variability space covered for point A then equals 20 and the normalized value becomes $((100 \cdot 20) + (5 \cdot 4,5) + (15 \cdot 2,5)) / 120 = 17,2 \approx 70\%$. In other words, delaying the introduction of the professional, medium and minimal solution characteristics would substantially increase the variability supported by Rohill's product line architecture.

¹ All products (100 per cent) use the license key for initial instantiation.

5 Related Work

The notion of variation points was introduced in [19]. This book discusses topics related to software reuse among which variability. In [20], several variability mechanisms are discussed and it sets forth how design decisions remove variability from an architecture, i.e., how variability is constrained. Using patterns to model variability in software product families is explained in [21] with a focus on the design phase. Design patterns are discussed in detail in [14] (a pattern is an element of reusable software). A domain-independent model of software system design and construction is presented in [5]. The model is based upon interchangeable software components (variants) and large-scale reuse. A categorization of features (variants) is suggested in [17], which is slightly related to the types of variation listed in [1]. The issues that have been identified in the case study relate to earlier work presented in [8] by, among others, the co-author of this paper. A framework of terminology and concepts regarding variability is discussed in [18]. It presents three recurring patterns of variability and suggests a method for managing variability in software product lines. Case studies and experience reports from our research group, e.g., [7][28], learn that evolution in a family of software products is more effortful than in stand-alone products due to dependencies between products and conflicting features.

6 Conclusions

Software architects and developers have been able to handle variability in software systems in the past in an implicit manner. However, present-day software systems, e.g., software product lines, require a more explicit approach. It becomes important to represent variability in advance to understand where change has to be planned and the options possible in particular situations.

This paper focuses on variability in industrial software product lines. The variability issues listed in Table 4 have been identified in a case study involving a Dutch software company. In our opinion, this company, Rohill Technologies BV, is representative for small and medium sized enterprises in software industry and provides a suitable context to verify and validate theories on variability. The issues identified in the study have been discussed and analyzed and illustrate the need for handling variability in a more explicit manner.

Table 4. Variability issues that have been identified in Rohill

Issue	Analysis
Variability identification	<ul style="list-style-type: none">• Limited insight into the consequences of selecting a particular variability mechanism• Methods, techniques and guidelines to select the optimal mechanism are not available
Variability dependencies	<ul style="list-style-type: none">• A notation format to describe variability is not available• Dependencies between variability and features are not made explicit
Tool support	<ul style="list-style-type: none">• Tools to maintain variability information are not available

These issues are (indirectly) related to each other and characterized by a common denominator, viz. variability representation. This implies an integrated approach towards finding a common frame of reference for variability, which may contribute to making variability more explicit.

A system independent method to represent and normalize variability is suggested in this paper. It consists of a graphical representation and normalization of the variability supported in a software system and is based on classifying variability realization techniques according to their introduction and binding times in the software lifecycle. The method is exemplified by applying it to Rohill's software product line. For example, the case study shows that replacing preprocessor directives by makefiles has hardly affected the variability supported in Rohill's software. On the other hand, the variability related to product instantiation has a substantial impact on the software system as a whole. In general, the method can be used to illustrate the variability of a software system without focusing on specific engineering aspects.

7 Acknowledgements

We would like to thank Rohill Technologies BV for participating in the case study.

References

- [1] F. Bachmann, L. Bass. Managing Variability in Software Architectures. Proceedings of the 2001 Symposium on Software Reusability, May 2001, 126-132.
- [2] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. Proceedings of the Fifth Symposium on Software Reusability, May 1999, 122-131.
- [3] L. Bass, P. Clements, R. Kazman. Software Architecture in Practice. Addison-Wesley, 1998.
- [4] L. Bass, P. Clements, S. Cohen, L. Northrop, J. Withey. Product Line Practice Workshop Report. Technical Report CMU/SEI-97-TR-003, Software Engineering Institute, 1997. <http://www.sei.cmu.edu/publications/publications.html>
- [5] D. Batory, S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 4, October 1992, 355-398.
- [6] J. Bosch. Design & Use of Software Architectures - Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000.
- [7] J. Bosch. Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. Proceedings of the First Working IFIP Conference on Software Architecture, October 1999, 92-97.
- [8] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl. Variability Issues in Software Product Lines. Proceedings of the Fourth Workshop on Product Family Engineering, October 2001, to be published at Springer LNCS.
- [9] L. Bronsword, P. Clements. A Case Study in Successful Product Line Development. Software Engineering Institute, CMU/SEI-96-TR-016, 1996. <http://www.sei.cmu.edu/publications/publications.html>

- [10] J. Coplien, D. Hoffman, D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, Vol. 15, No. 6, November/December 1998, 37-45.
- [11] P. Clements, L. Northrop. A Framework for Software Product Line Practice - Version 3. Software Engineering Institute, 2001. <http://www.sei.cmu.edu/plp/framework.html>
- [12] D. Dikel, D. Kane, S.Ornburn, W.Loftus, J.Wilson. Applying Software Product Line Architecture. *IEEE Computer*, August 1997, 49-55.
- [13] D. Emery, R. Hilliard, M. W. Maier. Software Architecture: Introducing IEEE Standard 1471, *IEEE*, Vol. 34, No. 4, April 2001, 107-109.
- [14] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [15] R. L. Glass. The Relationship Between Theory and Practice in Software Engineering. *Communications of the ACM*, Vol. 39, No. 11, November 1996, 11-13.
- [16] J. P. Gibson. Feature Requirements Model: Understanding Interactions. *Feature Interactions in Telecommunications IV*, IOS Press, June 1997.
- [17] M. L. Griss, J. Favaro, M. D'Alessandro. Integrating Feature Modelling with the RSEB. *Proceedings of ICSR98, IEEE*, June 1998, 36-44.
- [18] J. van Gurp, J. Bosch, M. Svahnberg. On the Notion of Variability in Software Product Lines. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, August 2001.
- [19] I. Jacobson, M. Griss, P. Jonsson. *Software Reuse - Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
- [20] M. Jazayeri, A. Ran, P. van der Linden. *Software Architecture For Product Families: Putting Research into Practice*. Addison-Wesley, May 2000.
- [21] B. Keepence, M. Mannion. Using Patterns to Model Variability in Product Families. *IEEE Software*, July/August 1999, 102-108.
- [22] R. R. Macala, L. D. Stucky, D.C. Gross. Managing Domain-Specific Product-Line Development. *IEEE Software*, 1996, 57-67.
- [23] M. D. McIlroy. Mass Produced Software Components. Report on a conference of the NATO Science Committee, 1968, 138-150.
- [24] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A.L. Wolf. *Self-Adaptive Software: An Architecture Based Approach*. *IEEE Intelligent Systems*, 1999.
- [25] D. Parnas. On the Design and Development of Product Families. *IEEE Transactions on Software Engineering*, Vol. 2, No. 1, 1976.
- [26] Rohill Technologies BV. <http://www.rohill.com>
- [27] K. Schmid. Scoping Software Product Lines - An Analysis of an Emerging Technology, in *Software Product Lines: Experience and Research Directions*. *Proceedings of the First Software Product Line Conference*, Kluwer Academic Publishers, August 2000, 513-532.
- [28] M. Svahnberg, J. Bosch. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance - Research and Practice*, Vol. 11, No. 6, 1999, 391-422.
- [29] W. Tellis. Introduction to Case Study. *The Qualitative Report*, Vol. 3, No. 2, July 1997. <http://www.nova.edu/ssss/QR/QR3-2/tellis1.html>
- [30] W. Tellis. Application of a Case Study Methodology. *The Qualitative Report*, Vol. 3, No. 3, September 1997. <http://www.nova.edu/ssss/QR/QR3-3/tellis2.html>
- [31] D. M. Weiss, C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.