

Architecting Product Diversification - Formalizing Variability Dependencies in Software Product Family Engineering

Michel Jaring and Jan Bosch
University of Groningen

Department of Mathematics and Computing Science
P.O. Box 800, 9700 AV Groningen, The Netherlands
{jaring, jan.bosch}@cs.rug.nl

Abstract

In a software product family context, software architects design architectures that support product diversification in both space (multiple contexts) and time (changing contexts). Product diversification is based on the concept of variability: a single architecture and a set of components support a family of products. Software product families need to support increasing amounts of variability, leading to a situation where variability dependencies become of primary concern. This paper presents (1) a formalization of variability dependencies and (2) a case study in designing a program monitor and exception handler. The case study uses the formalization to describe variability dependencies in constraint specification language style and shows that architectural robustness is related to the type of variability dependencies.

1. Introduction

Composing software products from software components is not a recent concept. Early work on software components already appeared three and a half decades ago [1], followed by the idea to develop so-called program families [2]. This has evolved into practical software engineering approaches that share the ability to promote software reuse across many products. An example of software reuse in practice is the successful adoption of software product families in industry. The goal of the software product family approach is the systematic reuse of core artifacts for building related software products. A software product family typically consists of a product family architecture, a set of components and a set of products. Each product derives its architecture from the product family architecture, instantiates and configures a subset of the product family components and usually contains some product specific code [3].

When developing a software product family, software architects try to prepare the family architecture for different product contexts, i.e., they prepare the architecture to sup-

port product diversification. Product diversification is based on the concept of variability and appears in all family artifacts where the behavior of an artifact can be changed, adapted or extended. Examples of variability are a configuration wizard to select the language or a license key entered by the customer to enable (or disable) particular functionality. Variability is implemented by delaying design decisions to a specific moment in the software development process, meaning that variant selection is delayed until a particular development or deployment phase such as implementation or run-time is reached. A typical example of a delayed design decision is a software system that dynamically adapts its run-time behavior by binding modules on-the-fly. The software architecture of such a system is dynamic - it is neither static during development nor at run-time. Design decisions can not be delayed beyond run-time, which implies that dynamic software architectures [4] are an extreme form of variability. Even a conditional expression is sometimes considered as a form of variability, meaning that virtually all software architectures exhibit a certain amount of dynamic behavior.

1.1. Variation points

So-called variation points have been introduced in [5] and are often used to express variability. A variation point refers to one or more delayed design decisions, has as an associated set of variants and indicates a specific moment in the development process. A typical example of a variation point is the preprocessor directive shown in Figure 1. The variation point in this example provides a choice in software operation mode, the available variants are embedded and simulation mode and the actual selection of a variant is delayed until the source code is compiled. In other words, the design decision 'operation mode' is not taken prior to compilation, meaning that the consequences of this decision should only affect the development phases that come after compilation.

As also discussed in [6], all design decisions are still open when developing a system from scratch, but they are

```

#define EMBEDDED /* SIMULATION */

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

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

```

Figure 1. Variation point for selecting either embedded or simulation mode during compile-time.

not left open intentionally. In other words, they do not refer to a particular variation point and are therefore implicit. Design decisions become explicit in the development process when the corresponding variation point (or points) is identified, i.e., when it is taken into account that specific variants may occur. A variation point is in an unbound state as soon as it is identified and as long as a particular variant has not been selected from the set of variants associated with this point. The variation point is bound when a variant is selected and incorporated into the system. A variation point is therefore in one of the following states at a given moment in the development process:

- Implicit: the design decision is not identified, but is accidentally left open;
- Explicit: the design decision is identified and intentionally left open:
 - Unbound: no variant has been selected from the set of variants associated with the variation point, i.e., the design decision is open until it binds a variant;
 - Bound: a particular variant is selected from the set of variants associated with the variation point and incorporated into the system, i.e., the design decision is final.

1.2. Domain and application engineering

There are two relatively independent development cycles in software product family engineering, namely domain and application engineering. Domain engineering is responsible for the design, development and evolution of the reusable artifacts such as the product family architecture and shared components. Application engineering, on the other hand, is about adapting the product family architecture to the architecture of the required product. In short, domain engineering *prepares* for product diversification and application engineering is the *act* of product diversification itself.

1.3. Variability dependencies

Product families need to support increasing amounts of variability. System functionality and system properties such as safety, security, reliability and usability move away from mechanics and electronics to software and become an integral part of the variability infrastructure. In addition, the number of products in a product family tends to grow, which makes the variability infrastructure more fine-grained and therefore more complex. The increasing amount of variability leads to a situation where variability engineering becomes of primary concern in software development. The number of variation points for industrial product families may range in the thousands, which is already a challenging number to manage, but the number of mutual variation point dependencies typically has an exponential relationship to the number of variation points. Particularly this exponential relationship illustrates that systematic approaches are necessary for properly managing variability. Systematic approaches require an easy to use and easy to understand formalization of the variability concept to describe dependencies in an unambiguous manner.

We detail a case study in representing variability in a family of Magnetic Resonance Imaging (MRI) scanners developed by Philips Medical Systems in [7]. The MRI case illustrates how variability can be made an integral part of system development at different levels of abstraction and identifies several research questions. This paper attempts to answer one of these questions: *what types of variability dependencies exist in software product family engineering?*

1.4. Outline

The next section suggests a formalization of variability dependencies. The formalization is used in a case study in designing a program monitor and exception handler in section 3. Future and related work is surveyed in section 4 and 5, respectively. The paper is concluded in section 6.

2. Formalizing variability dependencies

This section suggests a variability nomenclature and a taxonomy of variability dependencies.

2.1. System variability

Software product family engineering is characterized by two main types of development, namely development with an embedded perspective and development with a non-embedded perspective. Unlike non-embedded software, embedded software is typically (very) hardware dependent. For example, the display size of mobile phones in pixels depends on the manufacturer and sometimes even varies within a particular product family. The different display sizes are hardware variants and

each variant requires a specific display driver. These hardware variants have to be supported by a variation point in the mobile phone software, i.e., the software depends on the choices in hardware and the hardware can only function properly when the right software is selected.

Product families are often embedded systems that strongly depend on hardware, but this is hardly addressed in presentations of ideal applications. Due to dependencies between soft- and hardware, the concept of software reuse in an embedded system typically overestimates the flexibility of software. Supporting the differences in hardware is called the hardware challenge in [8].

In academia, system development is often considered as being independent from the hardware - it is assumed that the software does not have to take the hardware into account. However, organizations that develop complex systems tend to use a more generalized approach where system development comprises both soft- and hardware aspects. The reuse infrastructure in a product family may consist of only software or a combination of soft- and hardware. This is also the approach taken by, e.g., the CAFÉ project (from Concept to Application in system-Family Engineering). The focus is on software, but in a soft- and hardware context. See [9] for more information on CAFÉ and similar projects.

As detailed in [7], we have identified (at least) two types of *software* variability in relation to the *hardware* configuration of a system:

- Hardware neutral variability: software variability independent from the hardware configuration, e.g., multiple language support in a mobile phone;
- Hardware enforced variability:
 - Software variability that depends on the hardware configuration, e.g., text output formatting depending on the display size;
 - Software variability that is required to enable the hardware configuration, e.g., display drivers.

Literature on variability most often refers to hardware neutral variability - it generally assumes that variability originates from software and not from the combination of soft- and hardware. We refer to software variability as the *combination of hardware neutral and hardware enforced variability*, i.e., system variability. Please note that embedded software implements both hardware neutral and hardware enforced variability.

2.2. Describing variability

Binding a variation point involves establishing a *relationship* between the variation point and the selected variant. This relationship may imply certain *dependencies* (constraints), e.g., a system generally requires that specific variation points are bound to have a working, minimal system.

There can be many different types of dependencies and pinpointing them requires a more formal way to describe variability. We use a notation that has many characteristics of a constraint specification language. Constraint specification languages have been developed outside the immediate software engineering research community such as the configuration management community and have been used in practice for several decades now. See, e.g., [10]. Please note that we are trying to pinpoint the different types of variability dependencies and that we use a constraint specification language as a tool to prevent ambiguities - a constraint specification language is a means and not a goal in itself. The following nomenclature aims for describing variability in system-independent terms, i.e., independent from a particular system, method or organization:

- The set of all variation points:
 $VP = \{vp_a, vp_b, vp_c, \dots\}$
- The set of variants for vp_x :
 $vp_x = \{v_{x1}, v_{x2}, v_{x3}, \dots\}$
- The power set (the set of subsets) of all variants:
 $V = \{\{v_{a1}, v_{a2}, v_{a3}, \dots\}, \{v_{b1}, v_{b2}, v_{b3}, \dots\}, \{v_{c1}, v_{c2}, v_{c3}, \dots\}, \dots\}$
- A relationship between vp_x and v_{xn} (vp_x binds v_{xn}):
 (vp_x, v_{xn})

Dependencies between variation points and variants can be expressed in the form of conditional expressions:

- if vp_x is bound then vp_y should be bound:
if vp_x **then** vp_y
- if vp_x is bound then vp_y should bind v_{ym} :
if vp_x **then** (vp_y, v_{ym})
- if vp_x binds v_{xn} then vp_y should be bound:
if (vp_x, v_{xn}) **then** vp_y
- if vp_x binds v_{xn} then vp_y should bind v_{ym} :
if (vp_x, v_{xn}) **then** (vp_y, v_{ym})

Dependencies may involve negation, meaning that the condition and expression of the **if-then** statement can exclude variation points or variants from binding. For example, if vp_x binds v_{xn} then vp_y should **not** bind v_{ym} is expressed as **if** (vp_x, v_{xn}) **then** $\neg(vp_y, v_{ym})$. A relationship between a variation point and a variant may impose dependencies on other variation points and variants. For example, if vp_x is bound then **both** vp_y **and** vp_z should also be bound is expressed as **if** vp_x **then** $(vp_y \wedge vp_z)$. Similarly, if vp_x is bound then vp_y **or** vp_z **or** both should be bound is expressed with the inclusive OR operation as **if** vp_x **then** $(vp_y \vee vp_z)$. To complete the nomenclature, the exclusive OR operation, meaning that **either** vp_y **or** vp_z is bound, is written as **if** vp_x **then** $(vp_y \vee\vee vp_z)$. Table 1 summarizes the logic operators in the variability nomenclature.

Operator	Description	Input 1	Input 2	Output
\neg	negation	false true	n.a. n.a.	true false
\wedge	AND	false false true true	false true false true	false false false true
\vee	inclusive OR	false false true true	false true false true	false true true true
$\underline{\vee}$	exclusive OR	false false true true	false true false true	false true true false

Table 1. Logic operators in the variability nomenclature.

2.3. Taxonomy of variability dependencies

Binding (or not binding) a variation point generally imposes dependencies on other variation points and variants. As shown in Table 2, we have identified four main types of variability dependencies, each of which consisting of four subtypes. Each type has specific characteristics, e.g., a dependency between variation points has consequences that are different from a dependency between variants. In accordance with the concept of system variability, the taxonomy relates to both hardware neutral and hardware enforced variability, e.g., binding a *hardware* variant may impose dependencies on a *software* variation point. Please note that binding a variation point is not a dependency, but a relationship that may impose or is subject to dependencies instead.

3. Case study: PMEH

This section discusses a case study in designing a software component for an embedded legacy system. It outlines a generic Program Monitor and Exception Handler (PMEH) for (very) resource critical applications such as, e.g., a real-time process scheduler for a multi-processor environment. The study describes variability in constraint specification language style and uses the taxonomy to pinpoint the type of dependency that characterizes the reuse infrastructure of the component. The requirements were available in casual format, which we have adapted slightly to aid the reader's understanding. Interestingly, we have underestimated the complexity of this study in the first instance. Particularly the number of implied requirements and dependencies was greater than we thought.

Type	Subtypes
Dependencies between variation points vp_x and vp_y	if vp_x then vp_y if vp_x then $\neg vp_y$ if $\neg vp_x$ then vp_y if $\neg vp_x$ then $\neg vp_y$
Dependencies between variation point vp_x and variant v_{yn}	if vp_x then (vp_y, v_{yn}) if vp_x then $\neg(vp_y, v_{yn})$ if $\neg vp_x$ then (vp_y, v_{yn}) if $\neg vp_x$ then $\neg(vp_y, v_{yn})$
Dependencies between variant v_{xn} and variation point vp_y	if (vp_x, v_{xn}) then vp_y if (vp_x, v_{xn}) then $\neg vp_y$ if $\neg(vp_x, v_{xn})$ then vp_y if $\neg(vp_x, v_{xn})$ then $\neg vp_y$
Dependencies between variants v_{xn} and v_{ym}	if (vp_x, v_{xn}) then (vp_y, v_{ym}) if (vp_x, v_{xn}) then $\neg(vp_y, v_{ym})$ if $\neg(vp_x, v_{xn})$ then (vp_y, v_{ym}) if $\neg(vp_x, v_{xn})$ then $\neg(vp_y, v_{ym})$

Table 2. Taxonomy of variability dependencies in product family engineering.

3.1. Requirements

The PMEH consists of two parts, namely the monitor and the handler. The program monitor traces program flow (procedure call tree) and resource usage, i.e., space (memory) and time (CPU ticks). The exception handler takes control of system program flow in case of unexpected situations or erroneous conditions and attempts to execute the appropriate recovery actions. When the handler is finished, the program continues at a predetermined location or terminates. Programming languages like C do not have exception handling mechanisms, meaning that exceptions are handled through error codes or by calling library routines. As shown in Figure 2, the PMEH is to be integrated into a legacy software system written in C or C++ without changing existing functionality.

```

void legacy(int y)
{
    PMEH(..., func_id, nr_ticks, mem_size, ...);
    PMEH(..., y, ...); /* pre-condition check */
    /* legacy */
    PMEH(..., y, ...); /* post-condition check */
    PMEH(..., func_id, nr_ticks, mem_size, ...);
}

```

Figure 2. Exception handling with the PMEH in a C legacy system.

3.1.1. Program monitor. The PMEHE traces program flow by logging the following program monitor data in an output file `PMEH_output.txt` at the beginning and end of each procedure:

- Procedure name;
- Number of CPU ticks since program invocation;
- Size of allocated program memory;
- Descriptor to indicate whether the procedure is entered or exited.

3.1.2. Exception handler. The PMEHE is also used to check the pre- and post-conditions of the procedure arguments. The following exception handler data is logged in `PMEH_output.txt` at the beginning and end of each procedure:

- Variable name;
- Variable value;
- Non-critical system variables are checked against minimum and maximum values that are passed with the PMEHE call;
- System critical variables are checked against a list available to the PMEHE for minimum and maximum values. This list also includes preferred value ranges for normal operation.

In case of a false pre- or post-condition, the PMEHE decides on the progress of the program to handle the unexpected situation. Exception information is written in `PMEH_output.txt`. The PMEHE may take one or more of the following actions:

- Cancel current input vector;
- Skip all input vectors;
- Clear memory data space, backtrack to one step before the exception and copy preferred values into data space;
- Terminate program (emergency exit).

3.1.3. Additional information. The exception handler detects erroneous input and is essential for preventing system failure. System failure includes both soft- and hardware aspects, but in some cases also mechanical damage of physical machine components due to faulty software parameters. The program monitor, on the other hand, is used for debugging purposes, but can not prevent system failure. Disabling (parts of) the PMEHE is necessary for research and testing purposes. The exception handler is mandatory and the program monitor is strongly preferred for normal operation. The PMEHE itself is resource intensive and constrains the operating range of the system.

3.2. Variability analysis

As shown in Figure 3, two variation points have been identified by analyzing the requirements, namely the Program Monitor (PM) and the Exception Handler (EH). Each point has one variant, the actual program monitor and exception handler, respectively. This first analysis resulted in the following definition for the PMEHE:

- $PMEH \equiv VP = \{PM, EH\}$
- $PM = \{PM\}$
- $EH = \{EH\}$

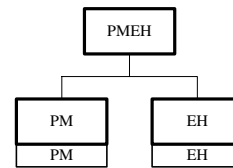


Figure 3. PMEHE first analysis. PM: Program Monitor. EH: Exception Handler.

As obvious as this analysis may seem, it does not meet the requirements. For example, during normal operation the exception handler has to be available, but this is not addressed in the analysis. To complicate matters, this condition does not concur with the requirement that the PMEHE can be (partly) turned on or off for research and testing purposes. Such conflicting requirements are common in variability engineering, but simple solutions like Figure 3 are often used as a basis for implementation. If the PMEHE would have been implemented according to this first analysis, the result would have been several implicit dependencies at best. Summarizing, research and testing mode is without additional conditions, but normal operation requires that $PMEH = \text{true}$ to bind both the program monitor and exception handler variation point:

- The program monitor is strongly preferred:
if $PMEH$ then PM
- The exception handler is mandatory:
if $PMEH$ then EH

Another requirement for the exception handler reads: Clear memory data space, backtrack to one step before the exception and copy preferred values into data space. This is core functionality of the exception handler, but it implies that backtrack information is available. After consulting the responsible architect it became clear that the program monitor provides this information, meaning that if the exception handler is mandatory, then so is the program monitor. This would result in one single variation point that has the entire PMEHE as the only possible option. However, this was considered unacceptable by the architect - such a 'dominant'

variation point that does not allow for selecting particular pieces of functionality undermines system flexibility. A solution was found by not identifying program monitor and exception handler variants, but feedback and feedforward variants instead. The feedback variant does not affect system program flow, as opposed to the feedforward variant. In other words, by looking at the PME_H from the perspective of the system it has to interact with, feedback and feedforward are the main actions.

Consulting the architect also revealed an additional requirement. To increase system reliability, a valid product configuration should bind all variation points to make the product configuration process as explicit as possible. This new requirement was met by adding the empty variant to each variant set in the PME_H. As shown in Figure 4, the analysis now identifies a system (PME_H) and a subsystem level (FB and FF), meaning that the PME_H can be a system level variation point without taking the specifics of the program monitor and exception handler into account.

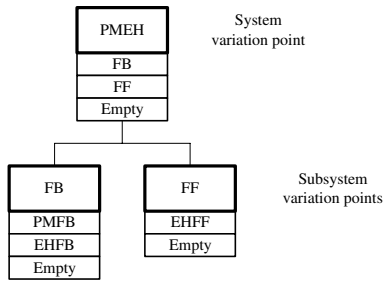


Figure 4. PME_H second analysis. FB: Feedback. FF: Feedforward. PMFB: Program Monitor Feedback. EHFB: Exception Handler Feedback. EHFF: Exception Handler Feedforward.

Dependencies for normal mode are:

- The PME_H has to be bound with a non-empty variant:
 $(PMEH, Empty) = \text{false}$
- The program monitor is strongly preferred:
if $\neg(PMEH, Empty)$ **then** $(FB, PMFB)$
- The exception handler is mandatory:
if $\neg(PMEH, Empty)$ **then** $((FB, EHFB) \wedge (FF, EHFF))$

The notation shows that variation point PME_H has two variants (FB and FF) that both have to be bound in case of normal mode. Variation point FB is subject to a similar, but impossible condition - a variation point can bind only one variant at a given moment. Such discrepancies are easy to find with the proper tools and although tooling is not necessary at the moment, computerized analysis of the dependencies is an absolute must in case of large-scale software sys-

tems. The results of the third and final analysis are shown in Figure 5. The feedback and feedforward variants are formalized as follows:

- $VP = \{PMEH, PMFB, EHFB, EHFF\}$
- $PMEH = \{FB, FF, Empty\}$
- $PMFB = \{PMFB, Empty\}$
- $EHFB = \{EHFB, Empty\}$
- $EHFF = \{EHFF, Empty\}$

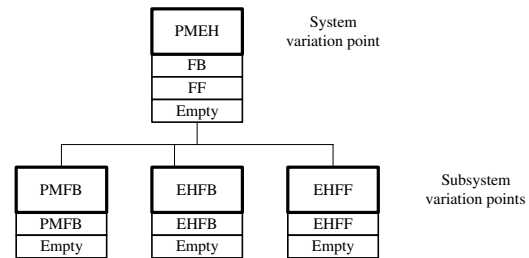


Figure 5. PME_H third analysis. FB: Feedback. FF: Feedforward. PMFB: Program Monitor Feedback. EHFF: Exception Handler Feedforward.

Dependencies for normal mode:

- $(PMEH, FF) = \text{true}$
- **if** $(PMEH, FF)$ **then** $((PMFB, PMFB) \wedge (EHFB, EHFB) \wedge (EHFF, EHFF))$

Dependencies for research and testing mode:

- **if** $(PMEH, FF)$ **then** $((PMFB, PMFB) \wedge (EHFB, EHFB) \wedge (EHFF, EHFF))$

The feedforward option requires both the program monitor and exception handler, which concurs with normal mode. Please note that the dependencies are virtually the same for both normal and research and testing mode. The only difference is the condition that the feedforward option has to be selected for variation point PME_H in normal mode, i.e., $(PMEH, FF) = \text{true}$. As said, a valid product configuration should bind all variation points to ensure all points will be considered during the configuration process. This results in the following additional dependencies:

- **if** $\neg(PMEH, FB) \wedge \neg(PMEH, FF)$ **then** $(PMEH, Empty)$
- **if** $(PMEH, Empty)$ **then** $((PMFB, Empty) \wedge (EHFB, Empty) \wedge (EHFF, Empty))$

3.3. Discussion

Although relatively small-scale, the PMEH case is a good example of the complexity that comes with variation. The architectural analysis illustrates that apparently simple requirements can entail complicated, in the first instance unforeseen variability relationships and dependencies. Solutions often appear as obvious, but may have hidden aspects that are important for preventing implicit dependencies. The complexity associated with the required PMEH variation is underestimated in the first and second analysis. The unwanted result of underestimating complexity is implicit dependencies and an architecture that does not meet the requirements. The third analysis is (probably) free from implicit dependencies and therefore more architecturally robust, but it is not the immediate solution that comes to mind when analyzing the PMEH requirements.

The main type of variability dependency in the PMEH is between variants, i.e., dependencies are not at the more abstract level of variation points, but at the level of specific variants. Having dependencies at the variant level results from the architectural decision to divide the PMEH into a system and subsystem level. This decision has changed the main type of dependency from **if** vp_x **then** vp_y to **if** (vp_x, v_{xn}) **then** (vp_y, v_{ym}) (see also Table 2). Dependencies are now primarily within the PMEH and independent from the legacy system. Separating the PMEH dependencies from the legacy system reduces the architectural erosion of both the PMEH and the legacy system.

Summarizing, the overall quality of the PMEH architecture is improved by moving dependencies from the variation point level to the variant level or, in more general terms, *architectural robustness is related to the type of variability dependencies*.

4. Future work

Each analysis of the PMEH is basically an educated guess expressed in constraint specification language style that is then verified by hand to find any inconsistencies. This approach has at least two problems: (1) an educated guess is based on experience and implicit knowledge instead of explicit, documented facts and (2) verification by hand may result in overlooking inconsistencies and is practically impossible for larger systems. A more useful approach would be to define the variability infrastructure in constraint specification language style and automate the process of finding and verifying architectural constructs. However, this requires not only taking the constraints of variation points and variants into account, but also their relative *impact*, i.e., some dependencies have more consequences in the form of, e.g., functionality and complexity than other dependencies. The feedback received on an earlier version of this paper [11] emphasized a similar need for formalizing the *architectural weight* of variability dependencies.

4.1. Hierarchy of variability dependencies

We attempt to formalize the *impact* of variability dependencies by complementing the taxonomy with a hierarchy of variability dependencies. The idea is that a relationship or dependency at a higher architectural layer (more abstraction) has more impact (more consequences) on the system than a relationship or a dependency at a lower architectural level (less abstraction). We are working on a four-tier hierarchy that is based on a generalization of the so-called Building Block Method. This method has been developed in-house by Philips Medical Systems and identifies soft- and hardware variability in a layered decomposition of system functionality. See our earlier work for an in-depth discussion of the Building Block Method and its variability aspects [7].

4.2. Dynamically reconfigurable architectures

The nomenclature and taxonomy are used in a proof-of-concept case study in designing a Real-Time Adaptive Signal Processing (RASP) system. System stability and change management are key issues in the development of RASP and depend strongly on run-time variability dependencies. RASP's software architecture defines the variability infrastructure and is designed to *configure a specific product family member* at system start-up and to *reconfigure the signal processing algorithm* at run-time, whereas its hardware architecture acts as a generic platform to accommodate the product configuration and algorithm reconfiguration process. In short, RASP has a dynamically reconfigurable software architecture (the product family architecture) and a static hardware architecture.

The aim of the RASP study is to describe the variability infrastructure of a dynamically reconfigurable system in constraint specification language style in such a way that the description can be used for design, implementation and documentation purposes.

5. Related work

Software reuse is a key concept in the software engineering community and dates back to the late 1960s [1] and 1970s [2]. Particularly the object-oriented paradigm became popular during the 1980s and led to object-oriented frameworks. These frameworks have evolved in several directions, one of which software product families. The notion of software product families has received attention in both research and industry since the 1990s. Software engineering research projects such as ARES, PRAISE, ESAPS and CAFÉ [9] and the software product line initiative at the Software Engineering Institute (SEI) [12] have introduced the software product family concept to main stream software engineering. For example, methods with a focus on software product family engineering as presented in, e.g., [13] have found their way to industry.

Software product families are based on the concept of variability: a single architecture and a set of components support a family of products. Variability is often expressed in terms of variation points, which notion has been introduced in [5] as part of the reuse process. Several variability mechanisms are discussed in [14]. This book also discusses how design decisions remove variability from an architecture. Patterns are used to model variability in software product families in [15]. Design patterns are elements of reusable software and discussed in detail in [16]. Features have been suggested in [5] as a useful abstraction to describe variability. A feature is defined in [3], although rather informally, as a logical unit of behavior that is specified by a set of functional and quality requirements.

As pointed out in [17], higher level abstraction and parameterization techniques are not well-suited for dealing with ad hoc variation of features found in different members of a product family. Managing variability requires a scalable approach such as discussed in, e.g., [18] and [19], but with the ability to represent the variability infrastructure in an independent manner from a particular system, method or organization throughout the development process.

6. Conclusions

This paper presents a formalization of variability dependencies in the form of a variability nomenclature and a taxonomy of variability dependencies in an attempt to answer the question: what types of variability dependencies exist in software product family engineering?

The nomenclature and taxonomy can be used to describe both soft- and hardware variability dependencies in constraint specification language style. Describing variability in a constraint specification language is particularly useful for automating the product configuration and verification process as well as for documentation purposes. Product configuration is about composing a product, whereas product verification is about determining if a particular configuration is possible.

The nomenclature and taxonomy have been applied in a case study in designing a program monitor and exception handler (PMEH) for an embedded legacy system. The overall PMEH structure (architecture) results from analyzing and identifying variation points and variants together with their relationships and dependencies while trying to retain the architectural robustness of both the PMEH and the legacy system. The architectural analysis of the PMEH illustrates that apparently simple system requirements can entail complicated, in the first instance unforeseen variability relationships and dependencies. The main type of variability dependency in the PMEH is between variants - dependencies are not at the more abstract level of variation points, but at the level of specific variants instead. Applying the taxonomy to the three analyses of the PMEH shows that *architectural robustness is related to the type of variability dependencies*.

References

- [1] M. D. McIlroy, "Mass Produced Software Components", *Proceedings of the NATO Software Engineering Conference*, Garmisch, Germany, pp. 138-155, 1968.
- [2] D. L. Parnas, "On the Design and Development of Product Families", *IEEE Transactions on Software Engineering*, Vol. 2, No. 1 pp. 1-9, 1976.
- [3] J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.
- [4] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, "An Architecture-based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, Vol. 14, No. 3, pp. 54-62, 1999.
- [5] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.
- [6] 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*, Amsterdam, The Netherlands, pp. 45-54, 2001.
- [7] M. Jaring, R. L. Krikhaar, J. Bosch, "Representing Variability in a Family of MRI Scanners", *Software: Practice and Experience*, Vol. 34, No.1, pp. 69-100, 2004.
- [8] A. Maccari, A. Heie, "Managing Infinite Variability", *Workshop on Software Variability Management*, Groningen, The Netherlands, 2003.
- [9] F. J. Linden, "Software Product Families in Europe: The ESAPS & CAFÉ Projects", *IEEE Software*, Vol. 19, No. 4, pp. 41-49, 2002.
- [10] F. J. Buckley, *Implementing Configuration Management: Hardware, Software and Firmware*, IEEE Press, 1996.
- [11] M. Jaring, J. Bosch, "Variability Dependencies in Product Family Engineering", *Proceedings of the Fifth International Workshop on Product Family Engineering*, Siena, Italy, pp. 81- 98, 2003.
- [12] P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.
- [13] D. M. Weiss, C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, 1999.
- [14] M. Jazayeri, A. Ran, F van der Linden, *Software Architecture For Product Families: Putting Research into Practice*, Addison-Wesley, 2000.
- [15] B. Keepence, M. Mannion, "Using Patterns to Model Variability in Product Families", *IEEE Software*, Vol. 16, No. 4, pp. 102-108, 1999.
- [16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [17] A. Karhinen, A. Ran, T. Tallgren, "Configuring Designs for Reuse", *Proceedings of the Symposium on Software Reusability*, Boston, USA, pp. 199-208, 1997.
- [18] R. van Ommering, F van der Linden, J. Kramer, J. Magee, "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, Vol. 33, No. 3, pp. 78-85, 2000.
- [19] F. J. van der Linden, J. K. Müller, "Creating Architectures with Building Blocks", *IEEE Software*, Vol. 12, No. 6, pp. 51-60, 1995.