Representing variability in a family of MRI scanners

M. Jaring\textsuperscript{1,*}, R. L. Krikhaar\textsuperscript{2} and J. Bosch\textsuperscript{1}

\textsuperscript{1}University of Groningen, Department of Mathematics and Computing Science, Blauwborgje 3, 9747 AC Groningen, The Netherlands
\textsuperscript{2}Philips Medical Systems, MR Software, Veenpluis 4-6, 5684 PC Best, The Netherlands

SUMMARY

Promoting software reuse is probably the most promising approach to the cost-effective development and evolution of quality software. An example of reuse is the successful adoption of software product families in industry. In a product family context, software architects anticipate product variation and design architectures that support product derivation in both space (multiple contexts) and time (changing contexts). Product derivation is based on the concept of variability: a single architecture and a set of components support a family of products. Modern software product families need to support increasing amounts of variability, leading to a situation where variability engineering becomes of primary concern. Variability is often introduced as an “add-on” to the system without taking the consequences for more than one lifecycle phase such as design or architecture into account. This paper suggests (1) a Variability Categorization and Classification Model (VCCM) for representing variability in the software lifecycle and (2) discusses a case study of a large-scale software product family of MRI scanners developed by Philips Medical Systems. The study illustrates how variability can be made an integral part of system development at different levels of abstraction. VCCM has been applied to the scanner family as an analysis tool. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: software product families; product derivation; variability modeling; dependencies

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. See, e.g., Reference [3, 4]. Promoting software reuse is probably the most promising approach to the cost-effective development and evolution of quality software. The adoption of software product families in industry is a successful example of reuse in practice. 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. Software architects try to prepare the family architecture for different product contexts, i.e., they prepare the architecture to support product diversification. Variability is the basis for product diversification and appears in all software family artifacts where the behavior of an artifact can be changed, adapted or extended.

The relationship between variability and software product families has been identified in research, see, e.g., Reference [5, 6], but insight into how this relationship applies to industrial software product families is still limited. The current line in variability research is to abstract the concept of variability, whereas in industry the focus is on how to deal with variability in (large-scale) software development. Many software systems have been developed in the past and all of these systems support variability to a certain extent. This means that software architects have dealt with variability in software development, often in an intuitive manner, and it is therefore not unlikely that practice may lead theory at this point [7]. A close cooperation between academia and industry is necessary to understand the current state of practice, the questions that arise in industry and to validate theories developed in academia. Several case studies such as Reference [8, 9, 10] report on experiences with software product families in an industrial context from a bird’s-eye view. Mapping out the products in advance as suggested in Reference [11] is not always possible, e.g., it is not always clear what product configurations are required [12]. Other work includes Reference [13, 14], but none of these publications describe product derivation in detail. In short, variability in an industrial setting has only been addressed recently and is often referred to as diversity, composability or variety [15, 16, 17].

Supporting variability requires a well-defined architecture and insight into the consequences of adapting a software system, meaning that adding variations in an ad hoc manner will typically erode the original design [18]. The architect who developed the software family architecture generally has a concept for what to do when a particular product architecture is needed and also understands its implications. If this architect is involved in the analysis of the required variation, the result is probably sufficient [19]. However, in industrial software engineering, the adaptors of the architecture are most often not the creators [20] and this may result in less than optimal solutions. A single variation may influence the system as a whole in subtle ways, ranging from component dependencies to different real-time behavior. It is for this reason that approaches like boxes-and-lines diagrams are not well-suited to map out variability in software systems, i.e., the implicit assumption is often that the boxes are the points of variation [19]. In our opinion, such approaches represent the architecture, the possible variation and, most importantly, variation dependencies in a too abstract (naïve) manner. A domain-independent model of software system design and construction is presented in Reference [21]. This model is based on variation in the form of interchangeable software components, aims for large-scale reuse and is a good example for pointing out that approaches like the aforementioned boxes-and-lines diagrams are too limited to deal with variability. Therefore, as also argued
in Reference [22, 23], software architecture design and implementation demands for proper
documentation, methods, techniques and guidelines to deal with variability.

Modern software product families need to support increasing amounts of variability, leading
to a situation where variability engineering becomes of primary concern. Variability is often
introduced as an “add-on” to the system without taking the consequences for more than one
cycle phase such as design or architecture into account. In addition, the interpretation of
the concept of variability tends to depend on (the area of expertise of) the people involved. To
engineer for variability requires a basic understanding of the variability consequences across
the different development phases by all parties involved such as architects, engineers, managers
etc. If not, typical, variability related problems may arise:

- The consequences associated with a variant are hard to predict, e.g., complex feature
  interactions are not fully understood before they are implemented [24].
- Integrating variants into the system is done before the optimal variant can be selected,
  e.g., adding variants during compile-time or run-time requires an early decision, but
deciding which variant to use should preferably be done at a late point in time [16].
- The architectural abstraction of the system does not scale-up to the increasing number
  of features and variants, e.g., subsystems that implement specific variants are represented
  as independent boxes in the top layer of an architecture [25].
- It is difficult to integrate newly developed variants into the system and to design new
  features into existing variants, e.g., variants are adapted after product instantiation to
  customize the product [25].
- Third party variants are adapted before integrating them into the system, e.g., an
  architectural mismatch due to a reusable part that is based on the wrong assumptions
  about the system it is to be part of [26].

The first part of this paper suggests a Variability Categorization and Classification Model
(VCCM) for representing variability in the software lifecycle according to criteria common to
virtually all software development projects. VCCM can be used as an analysis tool and does not
assume in-depth engineering or management knowledge on the system at hand. The second
part discusses a case study of a large-scale software product family of Magnetic Resonance
Imaging scanners developed by Philips Medical Systems. The study illustrates how variability
can be made an integral part of system development at different levels of abstraction. VCCM
has been applied to the scanner family as an analysis tool.

VARIABILITY

This section discusses the concept of variability, the main variability concerns in industry and
the need for identifying variability in software systems.

Delaying design decisions

The core idea of software family engineering is to develop a reuse infrastructure. Software
architects and engineers anticipate the different product contexts and design reusable elements
into the system by delaying design decisions [27, 6] to a later moment in the software development process. Each design decision constrains the number of possible systems that can be built [4], e.g., deciding for UNIX as the main platform omits all products not running on UNIX. A software system that dynamically links modules during run-time is a typical example of a delayed design decision. The architecture of such a system is dynamic, i.e., it is neither static during development nor at run-time. Dynamic software architectures [28] are an extreme form of variability, meaning that design decisions can not be taken beyond runtime. Conditional expressions are sometimes used to implement variability, which implies that virtually all software systems exhibit a certain amount of dynamic behavior.

So-called variation points have been introduced in Reference [3] and are often used to express variability. A variation point relates to one or more delayed design decisions and indicates a specific moment in the development process. A typical example of a variation point is the preprocessor directive shown in Figure 1. During early development there are many design decisions that have not been deliberately left open and therefore do not relate to a particular variation point [27], e.g., if development is started from scratch, all design decisions are still open and any system can be built. Thus, a variation point is in one of the following states:

- **Implicit**: the design decision is not deliberately left open.
- **Explicit**: the design decision is deliberately left open:
  - **Unbound**: a variant is not yet selected from the set of variants associated with the variation point, i.e., the design decision is open until a variant is bound.
  - **Bound**: a particular variant is selected from the set of variants associated with the variation point and inserted into the system, i.e., the design decision is final.

The set of variants associated with a given variation point defines the type of variation point:

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

Delaying design decisions is always a trade-off between required variability and available resources such as expertise and time. In general, the later a variation point is bound, e.g., during run-time instead of compilation, the more costly it is to implement this decision.
Software lifecycle

A software system is the result of a number of development phases. Each phase relates to its own representation, e.g., requirements collection relates to requirements specification, whereas implementation relates to source code. Software development consists of transformations of these representations and each phase is considered as a different abstraction level of the system. As illustrated in Figure 2, the lifecycle of a software system is a combination of development and deployment phases during which design decisions are made. Please note that the arrows are unidirectional, but that traversing back and forth through the phases is common in case of iterative software development. In relation to Figure 1, the choice for embedded or simulation mode becomes final during compile-time (binding the variation point), but was already taken into account before compilation during the architectural process (introduction of an unbound variation point). In short, a variation point is made explicit, i.e., introduced, at a specific phase and then bound at this same phase or at a later one in the software lifecycle.

In Reference [6], design patterns are used to model variability in product families, but with a focus on the design phase. Design patterns are elements of reusable software and are discussed in detail in Reference [29]. However, as explained, variability is not related to one particular phase such as design or architecture. Instead, it relates to several phases, e.g., in Figure 1, the variability concerning embedded or simulation mode ranges from architecture, design and implementation to compilation, i.e., both modes have to be taken into account between the moment of introducing and binding this variation point.

Software product families

A typical example of delayed design decisions is found in software product families. 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. See also, e.g., Reference [31].
Methods with a focus on software product family engineering such as the Family-Oriented Abstraction, Specification and Translation (FAST) process as described in Reference [30] have found their way to industry.

There are two relatively independent development cycles in software product family engineering, viz. domain and application engineering. Domain engineering is responsible for the design, development and evolution of the reusable artifacts, i.e., the product family architecture and shared components. Application engineering, on the other hand, is about adapting the product family architecture to fit the architecture of the required product.

As discussed in Reference [32], variability in software product families can be observed in two dimensions, viz. space, i.e., software artifacts appear in several products, and time, i.e., software artifacts change over time. For instance, in Figure 3, product P1 is instantiated at time T1 and exists during T1, T2 and T3, whereas P2 is re-instantiated at T4 to, e.g., bind an improved version of a variant. Product P3 is shipped at time T4, but not in the future. At time T3, product P1, P2 and P4 co-exist.

In Reference [3], features have been suggested as a useful abstraction to describe variability. Features apply to both the space and time dimension and are defined as follows. A feature is a logical unit of behavior that is specified by a set of functional and quality requirements [31]. The underlying idea is that a feature groups related requirements, meaning that features abstract from requirements. Features can be seen as incremental units of development [33], meaning that they do not relate to a single development phase, but to a number of transformation processes instead. A feature that depends on other features is a so-called crosscutting feature. Crosscutting features make it difficult to properly implement variability and once more illustrate the need for a systematic and reproducible approach to software system design. A categorization of features, among which variant features, is suggested in Reference [34]. A variant feature is an abstraction for a set of related features and is either optional or mandatory, e.g., in Figure 1, the variant features are embedded and simulation mode.
Variability concerns

We have identified three variability concerns that are relevant in software product family engineering in our earlier work, viz. variability identification, variability dependencies and tool support Reference [25].

Variability identification: mechanism. Variability is typically mapped to a single mechanism such as compile-time binding, which tends to simplify the design phase. However, the process of selecting a variability mechanism is often based on earlier experience, but the associated costs and impact on the development process as a whole 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, but then it is probably too costly to change the mechanism, if possible at all.

Variability identification: 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 family 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.

Variability dependencies: representation. A notation to describe variability is not available and insight into variability dependencies 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 to describe and discuss variability.

Variability dependencies: complexity. In general, the dependencies between variation points and features are not made explicit, i.e., they are often (indirectly) implied, not documented or even unintended. This increases complexity throughout development, e.g., it is not always clear what artifacts should or should not be selected to configure a product with the required functionality. This problem is strongly related to variability representation, i.e., to describe the dependencies it is, in the first instance, necessary to describe variability.

Tool support. The number of variation points, the associated variants as well as the dependencies tend to increase rapidly and tool support for maintaining variability information is necessary, e.g., to verify if the architecture meets the required variation. However, current software configuration and management tools have not been developed from a lifecycle perspective and often underestimate the complexity of dependencies, meaning that most tools refer to a particular development phase and not to interacting transformation processes.
Understanding variability throughout the lifecycle

Variability appears throughout the software lifecycle, from requirements to run-time, but how it is perceived often depends on the development phase(s) people are focusing (working) on. The variability aspects encountered during design may seem very different from the aspects encountered at run-time, but they do have a common cause, i.e., variability that cross-cuts the different transformation processes and their representations. To deal with variability across areas of expertise, actually across lifecycle phases, a shared understanding of its concept is necessary at first, which then may support methods, techniques, tools and guidelines that are needed in variability engineering and management.

For example, a platform independent product family has a layered architecture and its bottom layer is divided into two halves, UNIX and Windows NT, respectively. The architects may claim that they have taken the two platform variants into account, but it does not support the architectural implementation that is needed by the software engineers, i.e., like the aforementioned boxes-and-lines diagram, the solution is too abstract. In fact, it could even result into two separately developed products and still meet this architecture. In this example, architects and software engineers observe variability from their own area of expertise, but both parties have not taken the relationship between the transformation processes into consideration. The result is a lack of insight into the variant dependencies, which undermines the software reuse process.

Summarizing, variability appears in all lifecycle phases and developing a product family requires a shared understanding of its concept across these phases. This understanding should emphasize that variability cross-cuts the software lifecycle from requirements to run-time.

REPRESENTING VARIABILITY

This section suggests a Variability Categorization and Classification Model (VCCM) for representing variability in the software lifecycle according to criteria common to virtually all software development projects. VCCM can be used as an analysis tool and does not assume in-depth engineering or management knowledge on the system at hand.

Categorizing variability

There are many ways to support variability, ranging from conditional statements, preprocessor directives and inheritance to run-time selection of the appropriate components. See, e.g., Reference [35] for a taxonomy of variability realization techniques. Variability realization techniques are diverse, but characterized by their binding phase in the software lifecycle.

We categorize variability realization techniques according to this phase, i.e., a category relates to one specific binding phase and contains all variability realization techniques that bind at this phase. As shown in Figure 4, the nine lifecycle phases result in nine discrete categories of variability realization techniques. It can be argued that the lifecycle consists of more than nine phases, but this does not change the idea behind categorizing variability realization techniques.
As said, a design decision, whether it is delayed or not, constrains the number of possible systems that can be built, i.e., early binding of variation points reduces the amount of variability. It is for this reason that category R indicates minimum variability (entirely static) and category E maximum variability (entirely dynamic), e.g., category C is more variable than category D, but less than L. For instance, in Figure 1, the variation point for embedded or simulation mode is bound by means of preprocessor directives, which has compilation as its binding phase and therefore belongs to category C. Before binding this point, both embedded and simulation mode were possible variants, but after binding it, one variant has been omitted, i.e., the possible variation in the system has decreased. In addition, both variants have been introduced as an unbound variation point during the architectural process, i.e., the variants are explicit in the period between architecture and compilation. This is illustrated by the arrows in Figure 4. Please note that the phrase “decreased variation” does not imply that more variation is preferred to (or is better than) less variation.

The combination of variability categories that is found in a software system profiles the variability of this system. For example, the variation points in a software system map onto category C and L, meaning that it relies on variability realization techniques that have compilation and linking as their binding phases. If a newly introduced variation point maps onto category S (start-up), totally different variability realization techniques are required, e.g., dynamic linked libraries instead of preprocessor directives and makefiles. In many software systems, one or two categories are responsible for most of the variation, i.e., the majority of variation points are often bound (accumulates) at one or two phases in the software lifecycle.

Classifying variability

Various definitions of software architecture exist nowadays, e.g., see Reference [36], but they often have overlapping descriptions. In addition, and although the IEEE definition [37] is often referred to in literature, most organizations still use their own definitions. Therefore, without assuming a particular architecture, we classify software systems in maturity levels by pinpointing where the variants are bound in the software lifecycle. As long as the variation
points are not bound, the development process is domain oriented, but shifts towards an application oriented approach after binding them. The ability to delay the selection of application specific artifacts means that the different system contexts have been generalized, which in its turn implies the use of an architecture. The degree of generalization is an indication of system maturity, i.e., a higher level of maturity implies more domain engineering (multiple context development), but less application engineering (single context development) and vice versa. The binding phases indicate to what extent the process of variant binding is integrated into the system. For example, if variant binding is not part of a software system, each product is developed independently, but if variant binding is an integral part of the system, it can share software artifacts across several products. The first implies a less mature system in comparison to the latter, i.e., less domain engineering and more application engineering, respectively. We have identified the following levels of system maturity in our earlier work [38]:

I. **Independent product.** Products are developed independently from each other and do not share artifacts. Sharing of code among products is coincidental and development is not necessarily based on the same infrastructure.

II. **Standardized infrastructure.** A first step towards exploiting commonalities in products is to standardize the infrastructure on which the products are based. This infrastructure typically consists of the operating system in combination with commercial components such as a database management system. Domain specific components from external sources are generally integrated through proprietary glue code.

III. **Platform.** The next level is to establish a platform that is used as the basis for product development. A platform typically includes a standardized infrastructure and captures functionality that is common to all products. The common functionality that is not provided by the infrastructure is implemented in-house, but the platform is treated in application development as if it were a third party infrastructure.

IV. **Software product line.** The stage of software product line is reached when the amount of functionality provided by the platform is increased to the level where functionality common to several but not all products becomes part of the shared artifacts. Product specific functionality is introduced by adapting the products after instantiating them from the product line.

V. **Configurable product base.** There is a need to further develop product derivation support, especially in relatively stable domains where many product instances are required. The organization develops a single, configurable product base that is configured into the product during deployment phases such as installation and run-time. For instance, some companies ship the same (installed) code base to all of their customers. This code base configures itself according to a license key at the customer site and can be adapted after installation to, e.g., incorporate new language support.

A given maturity level does not bind variants beyond a certain phase. For example, linking libraries at run-time is typical for a configurable product base, but atypical for a standardized infrastructure. We have derived the closing (maximum) phase at which variants can be bound for each maturity level in a top-down manner:

V. **Configurable product base.** Variation can not be bound after run-time, meaning that run-time is the closing binding phase for a configurable product base to bind its variants.
Table I. Closing binding phase for each maturity level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Max. binding phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Independent product Architecture</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Standardized infrastructure Design</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>Platform</td>
<td>Implementation</td>
</tr>
<tr>
<td>IV</td>
<td>Software product Line</td>
<td>Link</td>
</tr>
<tr>
<td>V</td>
<td>Configurable product base</td>
<td>Run-time</td>
</tr>
</tbody>
</table>

IV. Software product line. Unlike a configurable product base, a software product line does not configure its products during deployment. Product instantiation is part of the development process and is generally done before delivery. The linking phase is therefore the closing binding phase since the remaining phases take place at the customer site, i.e., they are deployment phases. See Figure 2 for development and deployment phases.

III. Platform. A platform captures functionality that appears in all products and products are created by adding product specific code to the shared functionality. As opposed to a software product line, shared artifacts can not be in- or excluded, i.e., the shared functionality is fixed for all products. Adding product specifics by hard coding them into a product implies that variation is not bound after the implementation phase. In other words, implementation is the closing binding phase for a platform.

II. Standardized infrastructure. A standardized infrastructure selects a collection of third party components such as a database management system and uses glue code to integrate it into a product. Just as a configurable product base that binds its variants by selecting, e.g., components at run-time, a standardized infrastructure binds its variants when the externally developed components are selected from the suppliers. The only variation available after selecting these external components is in the form of predetermined, third party design rules like querying a database. Since a standardized infrastructure does not implement proprietary software, design is the closing binding phase for this level.

I. Independent product. This is the only maturity level that does not support reuse. In other words, variation is bound directly after specifying the requirements, because binding variation any later would imply reuse. The closing binding phase is therefore the architectural level, which is practically a one-to-one translation of the requirements, i.e., without taking commonalities into account.

Table I lists the maturity levels and the closing binding phase for each level. Each of these phases is inferred from the maturity level definitions, i.e., not being able to bind variants beyond a certain phase is the logical consequence of having a maturity level. In other words, a closing binding phase is the maximum binding phase for a maturity level, meaning that it does not exclude variant binding at an earlier phase unless this earlier phase is part of lower maturity level. The closing binding phases indicate sections, actually time frames, in the software lifecycle. Since each section refers to a particular maturity level, (parts of) a software...
A software system can be classified in one out of five maturity levels if its binding phases are known. Figure 5 shows the maturity levels in relation to the lifecycle, i.e., in accordance with Table I, a maturity level can not bind variants beyond a certain phase. Please note that different parts of a software system may apply to different maturity levels. Binding phases are often easier to pinpoint in a software system than the actual underlying variability realization techniques, especially if a system uses many techniques that interact throughout the system. It is for this reason that we use categories of variability realization techniques to identify variability in the software lifecycle. As explained, a category relates to a specific binding phase and contains all techniques that bind at this phase, i.e., a binding phase and its associated variability realization techniques are one instance.

Summarizing, variability in a software system is represented in the software lifecycle by categorizing variability realization techniques according to their binding phase. Depending on where these binding phases are located in the lifecycle, the system (or parts of it) is classified in one out of five maturity levels. In other words, classifying (parts of) a software system consists of three steps, i.e., (1) pinpoint the binding phases, (2) locate them in the software lifecycle and (3) read the corresponding maturity level. We refer to the model presented in this section as VCCM, i.e., Variability Categorization and Classification Model.

**CASE STUDY: PHILIPS MEDICAL SYSTEMS**

This section details a case study conducted in cooperation with Philips Medical Systems. The study describes an exemplar of a large-scale software product family of Magnetic Resonance Imaging scanners and uses VCCM to analyze its variability aspects.

**Methodology**

Case study research is a qualitative research method depending on sources such as observation, interviews, documents and the researcher’s impression. It is particularly applicable in an
industrial context and, as pointed out in [39], can be positivist. The positivist research paradigm argues that the methods of the physical sciences are applicable to the social sciences, i.e., social science researchers can seek to find laws that govern social interactions, similar to the way in which physical scientists can find laws governing physical phenomena. We primarily have a background in quantitative research, which explains our preference for positivist research. To understand the implications of case study research we found Reference [40] and [41] especially worthwhile. These publications summarize case study research from different perspectives and allow the reader to compare alternatives. We followed the recommendations that relate to Reference [39]. In this fashion, we have conducted a case study to collect, analyze and interpret large amounts of data in an exploratory manner, i.e., to extract a structure from the source material which in the best case can be formed as a rule that governs all the observations and is not known earlier. The positivist research paradigm together with the exploratory approach is appropriate if hardly anything is known about the matter at the outset of the project, which is often the case in a less researched and established area of study.

The goal of the study is to gain insight into the variability aspects in large-scale software development. The objectives deriving from this goal are:

- To document an example of variability in a large-scale software product family.
- To verify the applicability of VCCM in a real-world context.

Relevant to Philips Medical Systems, the research questions arising from these objectives were:

- Where and in what form occurs variability in the software lifecycle?
- What method(s) is (are) used to deal with variability?
- How are the variability concerns dealt with?
- Is VCCM useful for analyzing the variability aspects of a software system?

The study has been conducted in cooperation with the Product Group MR of Philips Medical Systems Nederland B.V. We obtained first-hand information on the software system by collecting company documents covering topics such as system requirements and specification, supported configurations, building block architecture, software options and execution architecture. In addition, we collected information through interviews with key persons in the organization using a four-tier approach, i.e., we conducted a number of interviews with system architects, software architects, system designers and software engineers. The interviewees were selected according to their experience and track record. Information collection took approximately six months and the interviews were held at Philips Medical Systems. Prior to the interviews, an introductory text was sent to the interviewees to explain the reason and topic of the interview. It also shortly explained variability in general terms to provide a context, because the many connotations of the word “variability” may otherwise result in an unfocused interview [25]. The introductory text also allowed the interviewees to prepare themselves, e.g., to collect the architectural overviews to be shown during the interviews. Only individuals have been interviewed to increase the possibility for having in-depth discussions and ensuring that the interviewee could speak freely. Minimally guiding the interviewee often results in less biased information. It is for this reason that we think the interviewee should elaborate as much as possible by asking him, the study included only men, open instead of closed questions.
The interviews were structured as follows:

- **Introduction:**
  - Interviewer: summarizes the aims and background of the study.
  - Interviewee: describes his job at Philips Medical Systems.
  - Interviewer: refers to the introductory text and, as a point of reference in accordance with this text, briefly explains variability to baseline the interview.

- By answering open questions, the interviewee explains how his work relates to variability and what he considers important in variability engineering.

- **Conclusion:**
  - Interviewer: summarizes the interview and asks for feedback.
  - Interviewer: concludes the interview.

To conclude the study, we have surveyed the document and software archive. This was especially useful to verify if the general structure of the code base concurs with the documentation hierarchy and vice versa. In addition, comparing the collected documents with the results of the interviews provided insight into the quality and consistency of the information obtained. Data that could not be verified have been omitted from the study.

**Company Background**

Philips Medical Systems are a division of Royal Philips Electronics, Europe’s largest electronics company. As a world leader in integrated diagnostic imaging systems and related services, Philips Medical Systems deliver portfolios of medical systems for diagnosis and treatment. Their product line includes technologies in X-ray, ultrasound, magnetic resonance, computed tomography, nuclear medicine, positron emission tomography, radiation oncology systems, patient monitoring, information management and resuscitation products [42].

With the recent acquisitions of Marconi Medical Systems, Agilent’s Healthcare Solutions Group, ADAC Laboratories and ATL Ultrasound, Philips Medical Systems are established as the global leader in most of its markets. Philips Medical Systems have pro forma sales of Euro 6.5 billion, employs approximately 22,000 people, research and advanced development is at 22 Philips sites and over 40 medical and technical institutions worldwide, sales and service operations in 63 countries and the countries of distribution exceeds 100 [42].

The main clinical product lines developed by Philips Medical Systems, for the remainder of this paper referred to as Philips, are based on X-ray, computed tomography and magnetic resonance imaging. These product lines are related to each other, i.e., X-ray is the basis for computed tomography and magnetic resonance imaging started out as a tomographic imaging technique, but uses principles fundamentally different from X-ray. The focus of the study is on the magnetic resonance scanners product line and a brief explanation of the background of magnetic resonance imaging is therefore useful.
Magnetic Resonance Imaging
Magnetic Resonance Imaging (MRI) [43] is an imaging technique used primarily in medical settings to produce high quality images of the inside of the human body, e.g., to examine the heart, assess blood flow, detect tumors, diagnose cancer, detecting brain damage in multiple sclerosis patients etc. MRI is based on the principles of Nuclear Magnetic Resonance (NMR), a spectroscopic technique that was discovered in 1946 and initially used to obtain microscopic chemical and physical information about molecules. In 1971, research showed that the magnetic relaxation times of different tissues differ, which opted magnetic resonance for scanning the inside of the human body. Four years later, phase and frequency encoding and the Fourier Transform were suggested to increase the performance of NMR. These techniques are still used today, although nuclear magnetic resonance imaging is often called magnetic resonance imaging instead due to the negative connotations associated with the word “nuclear”. It was during the late sixties that Philips started to conduct their own MRI research and they produced the world’s first head images using the MR principle in 1972 [42].

An MRI scanner [44] has a large electromagnet around a human-sized tube (bore) in which the patient or test subject lies. The electromagnet consists of coiled wires made of super-conductive material and the larger the number of windings, the greater the magnetic field. Superconductors at the right temperature offer no electrical resistance, i.e., once an electric current is introduced into the coils, the current will continue at full strength for years without the aid of additional electrical input. Magnetic field strength is measured in Tesla (T). Clinical imaging scanners have magnets with field strengths ranging from 0.02 to 3.0T. For comparison, the earth’s magnetic field is of the order of $10^{-4}$T. These fields are so strong that they immediately sweep up (heavy) metal objects in the vicinity of the magnet. So-called gradient coils are used to vary the strength of the main magnetic field. Radio frequency coils that transmit and receive radio signals are used in combination with the magnetic field to induce and measure a resonant field within the patient or test subject. The patient’s protons align themselves in the same direction as the magnetic field applied by the scanner. Under the control of the operator, the radio frequency coils in the scanner emit short bursts of radio waves that cause the protons to re-align themselves. When the radio waves and the protons vibrate at the same frequency, the protons absorb some of the radio wave energy. This is called resonance, and it is from this phenomenon that the term magnetic resonance is derived. Each time that the radio frequency coils are turned off, the protons go through a process of returning to their initial orientation. As the protons do so, they give up energy. This energy generates a voltage in a receiving wire antenna and is then converted into a digital signal that serves as the basis for MR images. The different tissues in the body give off signals whose strength depends on their chemical composition and location.

The rate at which images can be acquired is determined by the speed at which the gradient coils are turned on and off. Early MRI scanners needed hundreds of pulses to acquire data and patients had to spend an hour or more inside the machine. Today, gradient coils operate much more rapidly and even allow for real-time scanning. Figure 6 shows an example of an MR image taken with a modern scanner. The image shows 9 slices out of the 25 that have been acquired at a rate of 15 slices per second.
Figure 6. MR image of transverse sections of the patient’s body (liver, kidneys etc.). Image courtesy of Philips Medical Systems.

Stakeholders

Many MRI stakeholders, each of which having specific requirements, have been identified by Philips. For example, a cardiologist wants to perform a so-called stress test by making the patient cycle while scanning the patient’s thoracic (chest) area. For a radiologist, the patient throughput is a main issue, the more patients can be examined the more profitable it is, i.e., easy patient handling is an important requirement. Clinical scientists want to adapt new scan methods to improve medical imaging using an existing MRI system. Meeting factory requirements for (series) production demands a standardized approach for producing and assembling the various components. Service engineers want to diagnose the MRI system quickly and to replace an erroneous board without difficulties in a hospital environment. Developers have additional access rights when using the system, e.g., for system debugging purposes or for performing a feasibility study into a new scan method. These examples illustrate the many (conflicting) requirements that are fundamental for proper MRI system development, production, installation, configuration, deployment and maintenance. To pinpoint all (feasible) requirements, Philips have identified seven main groups of MRI stakeholders to structure the requirements collection process.

- **Patients.** The subject to be scanned prefers a fast and comfortable MRI scanning process.
- **Specialists.** Physicians in hospitals such as radiologists need the actual MR image.
- **Researchers.** Clinical scientists rely on an easily adaptable MRI system.
- **Operators.** The technician controls the MRI scanning process and has to fine-tune it.
- **Developers.** Engineers, physicists etc., depend on access to the MRI system and archives.
- **Manufacturers.** (Series) production means meeting factory requirements and constraints.
- **Service engineers.** Servicing requires easy diagnosis and replacement of MRI components.

System background

As opposed to their hardware platform, Philips’ software is (almost entirely) proprietary and has been under development for approximately twenty years. An important step was the department-wide introduction of object oriented programming in 1998, replacing C with C++
as one of the main programming languages. C and other languages are still used, especially when hardware specific code is required, but C++ is preferred if possible. Lately, C is also replaced with C#, an object-oriented language to build applications for the .NET platform. The total size of the system is approximately 3000 kilo lines of code (KLOC) divided over a few thousand modules and hundreds of components. Reuse of common medical software has been achieved, i.e., Philips have adopted the Capability Maturity Model (CMM) and have recently been assessed as CMM level 3 by an accredited third party.

Due to market demands, competition considerations and new technologies, a new member of the product family, a so-called preferred configuration, is released once a year. The code base continues to evolve and has adopted many new technologies of which COM and .NET are the most important ones. The Component Object Model (COM) is a widely used component software model for developing software from separately developed components. It is a group of conventions and libraries that allows interaction between different pieces of software in a consistent, object-oriented way and is language independent. .NET is a language-neutral environment for writing programs (small “building blocks”) that can easily and securely interoperate. Rather than targeting a particular hardware/operating system combination, programs will instead target the .NET environment and will run wherever .NET is available.

**Deployment view**

Figure 7 is an overview of the computing nodes in the MRI system. It is not a hardware design, but rather a simplified view on the scanner back-end. Software processes run primarily on the host node, as well as on the reconstructor node, which converts scanner data into raw image data, and the Data Acquisition System (DAS) node. Most of the executable code is stored locally on hard disk.

**Execution view**

As shown in Figure 8, the run-time structure of the MRI system recognizes seven process categories. Each category groups the processes according to functionality. Except for the
reconstructor and DAS processes, these processes run on the host node during normal operation. The arrows indicate unidirectional dependencies between processes.

Platform processes perform non-MRI specific tasks such as booting, logging and maintaining system configuration data. The central process in the DAC software is the Structured Query Language (SQL) server process. This process maintains the image and patient database, which is the general repository of the MRI system. Other processes in the DAC category focus on administrative tasks such as providing a user interface for selecting patient data and for archiving on the hospital information system. The reconstructor processes receive data from the DAS node, convert the data into raw image data and send the results to the host node. An alternative scenario receives data from the host node, processes the data and redirects the results to the host. The scanner processes define and control the actual scanning sequence. In other words, they control where and when a scan has to be performed. The semi real-time data viewing processes receive and visualize the reconstructed image data on-the-fly, meaning that real-time aspects are a key issue. The (post-)viewing processes apply data imaging techniques and render image data to output devices such as screen, printer, video and film.

Architecture view (building block method)

Philips have developed a method for software development, deployment and maintenance with an emphasis on large-scale software architecture construction, the so-called building block method. The building block method is component oriented and follows the product family approach, i.e., components are part of a construction set from which different products can be configured. The MRI system uses the building block method and has two main properties, viz. configurability and reusability. Stakeholder requirements are taken into account with general requirements such as maintainability, flexibility, stability, robustness etc. Early work on creating architectures with building blocks in Philips, although not directly related, is available in Reference [45]. The building block method is based on the following design concepts [46]:

- **Building block.** The software is structured into components, the building blocks, which are identifiable units during software development, but also on the target hardware.
• Interface. A building block has an interface specified in the Building Block Interconnection Language (BBIL), which hides the internals of a building block from the environment. Any language can be used to implement the building block.

• Layer. A layer consists of a number of building blocks and is used to structure the system horizontally. A building block can only use entities such as data exported by building blocks in lower layers. Layers are the basis for incremental loading and system testing.

• Application and Generic. A building block identifies two roles, application specific and generic. Application specific applies to product specific behavior, whereas generic applies to behavior that is common to a number of products.

• Subsystem. A subsystem consists of a number of building blocks, meaning that it covers a substantial part of the functionality of the system such as the operating system.

• Data Classification. System data is distributed across all building blocks and can not be accessed directly from outside a building block. Data outside a building block is accessed via specific procedures and is classified into three categories, viz. hard (can not be modified at run-time), soft (have to be kept stable and consistent) and dynamic data (may change during run-time).

• System Aspect. System aspects classify substantial parts of the system in terms of general demands such as flexibility and real-time constraints.

• OMA-view. The OMA-view groups the main system aspects and defines three system observations, i.e., Operational, Maintenance and Administration. The OMA-view is used throughout system development, e.g., to verify the architecture and structure system testing, and abstracts from the functional structure of the system.

A building block is a hierarchically organized, functional unit of the MRI system. It may consist of a number of hierarchically lower building blocks and has a unique name and description. The hierarchy is a tree structure, i.e., building blocks do not overlap and are either a leaf building block or can be decomposed into additional building blocks. A building block does not necessarily add functionality, they are also used to group hierarchically lower building blocks. Besides a hierarchical relationship, there is a non-hierarchical use relationship between building blocks. In other words, the building block method recognizes part-of relationships to represent the hierarchy and use relationships to represent dependencies. The building block interfaces are specified in BBIL, but the internals of a building block can be implemented in any programming language using a development method that fits the building block specifics. The document and software archives are structured according to the building block hierarchy, resulting in abstract (top-level) archiving and detailed (low-level) archiving. Top-level archiving is about grouping functionality, whereas low-level focuses on the actual implementation. The building block method is also used as a management tool to track development. Project deliverables are expressed in building blocks and each block is subject to design, implementation and test standards. Each building block has one owner assigned who is responsible for the contents of this block. Building blocks that require more than one area of expertise, e.g., soft- and hardware disciplines, have multiple owners assigned. Building block owners are listed in a database that can be accessed throughout the organization. System architects are responsible for the building block hierarchy.
In short, the building block method is a structuring principle for a number of development related activities. Each building block acts as a viewpoint on a part of the MRI system from an abstract (documentation), implementation (software) and project management perspective (tracking the development process), where hierarchical (part-of relationships) and dependency aspects (use relationships) characterize each viewpoint.

The MRI system is decomposed into second-level building blocks in Figure 9. The top-level building block represents the MRI system as a whole and consists of the following subsystems:

- **Magnet.** Provides the main electromagnetic field, i.e., a constant, homogeneous field.
- **RF system.** Induces and measures a resonant field in the patient’s body in combination with the main magnetic field, i.e., radio waves and protons vibrate at the same frequency.
- **Gradient.** Varies the strength of the main magnetic field, i.e., the image acquisition rate is determined by the speed at which the gradient coils are turned on and off.
- **Acquisition control.** Acquisition and control of scan data in cooperation with the reconstructor, i.e., receiving scan data and converting it into raw image data.
- **Platform.** Operating system, i.e., non-proprietary software such as Windows and VMS.
- **Viewing.** Optimizes the raw data from the reconstructor, i.e., digital image processing.
- **Patient handling.** Patient positioning and observation, i.e., reading physiology sensors.
- **Patient administration.** Archiving of patient specific data, i.e., database interaction.

Philips define the MRI architecture as the organization and cooperation between system components. The building block method is considered as an informal way to describe this architecture, meaning that it contains concepts that are related to the architecture of the system such as building blocks, interfaces, and layers. The building block hierarchy represents the MRI architecture in a top-down manner and both the document and software archive follow the tree structure in a one-to-one fashion. In other words, the MRI architecture is not just an abstract concept, it is the blueprint for the entire development process.

**Product Family**

The MRI product family is based on composing soft- and hardware building blocks. Most building blocks provide variants, e.g., in Figure 10, the variants for the system-level building
Figure 10. MRI product family: composing soft- and hardware building blocks.

block MRI System are Intera (closed MRI systems) and Panorama (open MRI systems). In other words, composing a specific MRI configuration is about selecting the appropriate building block variants. A configuration is either preferred or possible, meaning that is either a standard member of the product family or a customer specific configuration. Building block dependencies limit the number of systems that can be built, i.e., knowing these dependencies is a key issue in MRI configuration management.

Variability mechanisms

Building block variants are selected by means of a software key and a configuration file. In general, the software key focuses on software variants that support “pure” software functionality and the configuration file on software variants that support hardware components.

The MRI code base is installed in compiled form on every system that is shipped. The software remains inaccessible until it is unlocked by a valid software key, i.e., software option bits in the key enable or disable certain building block variants. Two types of option bits exist, viz. product and non-product option bits. Product bits define a specific MRI configuration, a so-called package, whereas non-product bits are not meant for commercial use, but for internal, service or clinical science purposes instead. Examples of non-product option bits are test mode, patch mode, factory mode etc. An alternative software key, the clinical science key, is for selected customers only. These customers cooperate with Philips in clinical case studies, which
provides valuable feedback for improving the MRI system. The clinical science key unlocks additional functionality necessary for MRI research, e.g., it disables certain restrictions or enables newly developed soft- or hardware for testing purposes.

The hardware configuration installed at the customer site is checked against the configuration file. This file lists the configurations that are feasible, i.e., technically possible. A certain choice of hardware has consequences for the software, e.g., most hardware components require specific drivers. Together with the initialization parameters, such dependencies are also stored in the configuration file. Hardware configuration verification takes place as one of the first steps during system start-up and, if approved, is followed by processing the software key. The appropriate building block variants are selected if the key does not occur in a list of keys that define invalid relationships between building block variants and if the variants defined by the option bits are in accordance with the hardware configuration. Normal operation is reached after the variation points have been bound.

**Hardware neutral versus hardware enforced variability**

As said, the MRI configuration process depends on the software key and the configuration file. Both of these variability mechanisms operate on software variants, but have a different focus, i.e., the software key is focused on software options, whereas the configuration file is focused on hardware options. This difference in focus implies at least two types of software variability:

- **Hardware neutral variability:** software variability independent from the hardware configuration, e.g., multiple language support.
- **Hardware enforced variability:**
  - Software variability dependent on the hardware configuration, e.g., parts of the clinical viewing packages.
  - Software variability required to enable the hardware configuration, e.g., hardware drivers for the gradient coils.

A variant clearly appears in the form of a building block option in the development process, which makes pinpointing where what happens in the lifecycle relatively easy. Both types of variation points are anticipated early in the development process, are then made explicit and finally bound during system start-up. Hardware neutral and enforced variation points share system start-up as their binding phase, but are made explicit at different moments in the lifecycle. Hardware enforced variation points generally become explicit during implementation, whereas hardware neutral variation points become explicit during design or earlier. Hardware neutral and enforced variability are both important in MRI system development, although, as opposed to the past, hardware neutral variability is probably more complex than hardware enforced variability, especially in terms of dependencies.

Both hardware neutral and enforced variability are for the larger part characterized by similar binding processes. As shown in Table II, variant binding consists of several steps, but can be summarized as loading a particular piece of software after a condition on variable during system start-up. Although they share the same binding phase, hardware neutral variants are taken into account prior to hardware enforced variants, i.e., during design and implementation,
Table II. MRI software variability aspects.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mechanism</th>
<th>Binding phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware neutral variability</td>
<td>(1) Read software key, (2) verify software key, (3) verify configuration, (4) select building block variants and (5) load software component/module/library</td>
<td>System start-up</td>
</tr>
<tr>
<td>Hardware enforced variability</td>
<td>(1) Read configuration file, (2) verify configuration, (3) select building block variants and (4) load software driver/component/module/library</td>
<td>System start-up</td>
</tr>
</tbody>
</table>

respectively. As explained, as soon as a variation point is made explicit, the number of possible products is limited to the variants associated with this variation point, regardless of whether the variation point is bound. The earlier product specific features are introduced in the development process, the less generic this development process is. Consequently, hardware enforced variability is dealt with in a more generic manner than hardware neutral variability.

The building blocks and their variants shown in Figure 11 baseline MRI functionality. Use relationships are not depicted in the building block hierarchy, but documented separately instead. A building block typically has either hardware neutral or enforced variants, e.g., the building block Magnet has hardware enforced variants, but the variants of Patient Administration are mainly hardware neutral.

VCCM: categorizing variability

Binding phases are often easier to pinpoint in a software system than the actual underlying variability realization techniques, especially if a system uses many techniques that interact throughout the system. Pinpointing binding phases can be done by observing the system as a result of its variability realization techniques or more precisely, by observing where variants are selected in the lifecycle. Interestingly, binding phases can be verified by checking if the code base implements the appropriate variability realization techniques.

Prior to surveying the document and software archive, we inferred the MRI binding phases by analyzing the system itself. The MRI code base is available in compiled form on each system that is shipped and is configured at the customer site. Variation points are therefore not bound before the installation phase, leaving installation, start-up and run-time as possible binding phases. The installation process primarily consists of assembling the MRI hardware, inserting the software key and composing the configuration file. In other words, the installation process does not interact with the code base and therefore can not bind any of the variants that the code base provides. Due to MRI system constraints, system critical variants have
to be verified, selected, bound and initialized before normal operation, i.e., before run-time. Normal operation requires that system critical components such as the magnet are available in terms of soft- and hardware. In general, hardware enforced variability is system critical, meaning that the MRI system can not reach normal run-time behavior if the hardware is not supported. The only remaining binding phase for this type of variability is therefore system start-up. Hardware neutral variability, on the other hand, does not require product specific hardware components, which means that it is less demanding than hardware enforced variability, although it often is system critical and required for normal operation. Typically, system critical variability determines where variation points are bound in the lifecycle, i.e., the variability realization techniques used for system critical variation points are also used for variation points that are not system critical [25]. We therefore inferred, prior to surveying the document and software archive, that the MRI variation points, whether they are hardware neutral or enforced, most likely bind their variants during system start-up.

We have verified our findings with the document and software archive and concluded that they concur. Due to the building block structure of the archives, it is relatively easy to trace and “physically” pinpoint variation points in the MRI code base. Summarizing, the MRI variation points, regardless of type, are bound during system start-up, i.e., the main MRI variability category is category S (start-up). See also Figure 12.

Figure 11. MRI subsystem building blocks and their variants: primary configuration elements. RF System: Radio Frequency System. DAS: Data Acquisition System. HIS: Hospital Information System. RIS: Radiology Information System. PACS: Picture Archiving and Communication System.
As discussed, hardware neutral and enforced variability are anticipated early in the development process and share system start-up as their binding phase. However, hardware neutral variability is made explicit in the form of distinct variants prior to hardware enforced variability, i.e., during implementation and design, respectively. This difference in introducing unbound variation points is illustrated by the arrows in Figure 12.

**VCCM: classifying variability**

The MRI system can be classified in one out of five maturity levels since category S (start-up) has been pinpointed as the main MRI variability category. This category relates to the binding phase system start-up and contains all variability realization techniques that bind at this phase, i.e., it abstracts from system specific variability realization techniques. Taking Table I into account, category S indicates that the overall maturity level of Philips’ MRI code base concurs with level V. In other words, the MRI software product family amply classifies as a configurable product base, meaning that it has domain engineering, software reuse and product derivation support as key characteristics. Please note that the classification process uses objective data, i.e., variation point binding phases and variability realization techniques can be verified, regardless of the researcher’s bias or prejudices. The combined results of first categorizing and then classifying MRI variability are shown in Figure 13.

VCCM is especially useful for analysis purposes. For example, binding hardware enforced variability during compilation instead of system start-up does not concur with the overall variability profile in Figure 13. Product configuration would then involve two binding phases and not just one, i.e., variability dependencies would also appear in the time dimension and this makes them more complex. Selecting specific variants would require compiling the code base once again, but this is generally done at the organization site and not at the customer site. In short, the system would not classify as configurable product base, but as a less mature combination of a software product line and a configurable product base.
DISCUSSION

This section discusses the variability concerns in relation to Philips’ MRI product family. The concerns are described in the variability section.

Variability identification: mechanism

The MRI system is structured into components, so-called building blocks, which are identifiable units during software development, but also on the target hardware. The building block method is a structuring principle for a number of development related activities and has an emphasis on the construction of the software architecture. Each building block acts as a viewpoint on a part of the MRI system from an abstract (documentation), implementation (software) and project management perspective (tracking the development process), where hierarchical (part-of relationships) and dependency aspects (use relationships) characterize each viewpoint. The building block interfaces are specified in an interface definition language, but the internals of a building block can be implemented in any programming language using a development method that fits the building block specifics. The building block documentation recognizes four levels in the software development process, each of which consisting of a number of lifecycle phases, together with the associated tools and techniques.

The building block hierarchy represents the MRI architecture in a top-down manner and both the document and software archive follow the tree structure in a one-to-one fashion.
In other words, the MRI architecture is not just an abstract concept, it is the blueprint for the entire development process. Most building blocks provide variants, e.g., in Figure 10, the variants for the system-level building block MRI System are Intera (closed MRI systems) and Panorama (open MRI systems). A software key in combination with a configuration file is used to select the appropriate building block variants. Both of these variability mechanisms operate on software variants, but have a different focus, i.e., the software key is focused on software options, whereas the configuration file is focused on hardware options. This difference in focus implies at least two types of software variability:

- **Hardware neutral variability**: software variability independent from the hardware configuration, e.g., multiple language support.
- **Hardware enforced variability**: 
  - Software variability dependent on the hardware configuration, e.g., parts of the clinical viewing packages.
  - Software variability required to enable the hardware configuration, e.g., hardware drivers for the gradient coils.

As shown by the arrows in Figure 12, both types of variability are anticipated early in the development process and share system start-up as their binding phase. However, hardware neutral variability is identified in the form of distinct variants prior to hardware enforced variability, i.e., hardware neutral variability is made explicit during design and hardware enforced variability during implementation. For both types of variability, the variability mechanisms typically load a particular piece of software after a condition on variable during system start-up.

**Variability identification: phase**

Properly applying the building block method means that even during the design phase building blocks or variants can be added to or removed from the structure. This ability to delay the introduction of variability implies that it is not necessary to make variability explicit before the implementation phase. It is probably for this reason that the implementation phase is aimed for to bring in the MRI variability. However, hardware neutral variability is made explicit during design or an earlier phase, but not during implementation. The earlier product specific features are introduced in the development process, the less generic this development process is. Consequently, hardware enforced variability is dealt with in a more generic manner than hardware neutral variability. It seems that, in comparison to hardware neutral variability, hardware enforced variability is based on long-term experience and is often considered as less abstract, particularly when it comes to dependencies. Hardware enforced variability uses software for a type of variability that already existed before software became large-scale, whereas hardware neutral variability is the result of being able to use software in products, i.e., the latter is a more recent type of variability. Although they are different types of variation, they are often dealt with similarly.
Variability dependencies: representation

The building block structure represents MRI variability from system level to functional units such as soft- and hardware components. The building blocks, building block variants and the part-of relationships are the basic elements for a notation to describe variability from an abstract to a detailed perspective, i.e., from the root building block to the leaf building blocks. In other words, variability representation is an integral part of MRI system development and not limited to a single development phase such as architecture or design. As shown in Figure 10 and 11, variability clearly appears at different levels of abstraction, although without depicting use relationships. Use relationships are documented separately instead. The building block method does not address where to introduce and bind variation points in the lifecycle. It does imply that variation should not be introduced before the implementation phase, but it does not actually represent or enforce it.

Variability dependencies: complexity

Dependencies between building blocks are specified in addition to the building block structure, e.g., in Figure 11, a 21.3 MHz RF-system depends on a standard gradient coil and is documented in a separate document. However, not all dependencies are documented, undocumented dependencies are implied and sometimes difficult to trace. As variability is characterized by a hardware neutral and enforced perspective, so are the dependencies associated with it. Hardware dependencies are generally clearer and better documented than software dependencies. This is most likely due to how the MRI system is decomposed into building blocks, i.e., the functional decomposition has an emphasis on hardware. A hardware approach for software-based building blocks may result in underestimating the complexity of software dependencies. Hardware-based functionality implies a building block decomposition based on (standardized) hardware components, whereas such a predetermined decomposition is not available for software-based functionality. It is safe to assume that all variability dependencies start out as implicit and that hardware dependencies become explicit, literally “materialize”, prior to software dependencies. The sooner dependencies are known, the easier it is to delay design decisions, i.e., the easier it is to delay the introduction and binding of variation points. This may explain why hardware neutral variability is made explicit prior to hardware enforced variability in Figure 12. Software development often faces more problems pinpointing requirements and dependencies than hardware development, which has prompted for a more elaborate and explicit software development process in the past. As a result, the MRI software development process tends to be more organized than its hardware counterpart due to establishing basic project management processes to track cost, schedule and functionality.

Tools

Due to the number of variation points, variants and dependencies in the MRI system, it is necessary to maintain variability information to understand what configurations are possible. The building block method outlines this information as part-of and to some extent use relationships. The building block structure is accessible department-wide (intranet) and
adheres to strict check in/check out procedures. The repository stores the most recent version of the document and software archive and also maintains the changes that have been made to these archives. Building block specifics, i.e., documentation, source code etc, are available by selecting a particular block from the building block structure. Just as the building block method, tool support has been developed in-house and addresses various levels of software development, i.e., requirements analysis, architectural and functional design and building block design and coding. Tool support for configuration validation is independent from the building block method, which indicates that configuration management is not an integral part of the method. The building block tools do not provide support for tracing affected blocks in case of changing requirements and constraints, resulting in an implicit approach for finding these dependencies. This is particularly an issue in case of hardware neutral variability dependencies.

Research questions

Most variability concerns have found an answer in Philips through the continuously evolving building block method. However, an important concern is variability dependencies, particularly in case of hardware neutral variability. In short, software dependencies are often dealt with from a hardware perspective and this may result in oversimplifying software-based relationships. Together with Philips, we have identified the following research questions for future studies:

- What are the technical and business consequences of having variability dependencies?
- What types of variability dependencies exist?
- How to identify, verify and validate variability dependencies?

We are currently working on a notation for defining and describing variability in the context of the building block method. This notation should be easy to understand, well-suited for tooling purposes and should have the characteristics of a constraint specification language.

CONCLUSIONS

In a product family context, software architects anticipate product variation and design architectures that support product derivation in both space (multiple contexts) and time (changing contexts). Modern software product families need to support increasing amounts of variability, leading to a situation where variability engineering becomes of primary concern. Variability is often introduced as an “add-on” to the system without taking the consequences for more than one lifecycle phase such as design or architecture into account. The first part of this paper suggests a Variability Categorization and Classification Model (VCCM) for representing variability in the software lifecycle according to criteria common to virtually all software development projects. VCCM can be used as an analysis tool and does not assume in-depth engineering or management knowledge on the system at hand. The second part discusses a case study of a large-scale software product family of Magnetic Resonance Imaging (MRI) scanners developed by Philips Medical Systems. The study illustrates how variability can be made an integral part of system development at different levels of abstraction. Table III summarizes the overall case study results.
Table III. Variability concerns in Philips Medical Systems’ MRI scanner development.

<table>
<thead>
<tr>
<th>Concern</th>
<th>Analysis</th>
</tr>
</thead>
</table>
| Variability identification | • Variability is identified in a layered, functional decomposition of the MRI system, viz. the in-house developed building block method, which has an emphasis on large-scale software architecture construction. Each building block acts as a viewpoint on a part of the MRI system from an abstract (documentation), implementation (software) and project management perspective (tracking the development process).<br>
• The building block structure generally follows a decomposition from a hardware perspective for both soft- and hardware functionality.<br>
• An implicit distinction is made between hardware neutral variability, e.g., multiple language support, and hardware enforced variability, e.g., hardware drivers for the gradient coils.<br>
• Hardware neutral variability is identified in the form of distinct variants prior to hardware enforced variability in the development process, i.e., during design and implementation, respectively. Both types of variability share system start-up as their binding phase. |
| Variability dependencies | • Hierarchical (part-of relationships) and dependency aspects (use relationships) characterize each viewpoint in the building block method and are dealt with independently from each other.<br>
• As opposed to hardware enforced variability, the dependency aspects of hardware neutral variability tend to be less explicit.<br>
• Hardware neutral variability is independent from hardware enforced variability and vice versa. |
| Tool support | • Several tools have been developed in-house to support the building block method and configuration validation.<br>
• Information from the different viewpoints is readily available through an intranet for each building building block by selecting the appropriate block from the tree hierarchy.<br>
• Configuration validation is done independently from the building block structure, i.e., configuration information is stored apart from the structure. |

Philips Medical Systems recognize variability engineering and management as essential for developing a software product family. This has resulted in a continuously improving development process that is characterized by identifying commonalities, i.e., so-called building blocks, and exceptions, i.e., variants that are part of a building block. VCCM has been applied to the MRI scanner family as an analysis tool. The MRI scanner family classifies as a configurable product base and shows a difference in handling software variability independent from the hardware configuration and software variability dependent on the hardware configuration. This difference is probably the result of a decomposition process that is similar for both soft- and hardware functionality and may result in oversimplifying software-based relationships. A key issue for future research is variability dependencies, particularly how to define and describe them in a notation.
ACKNOWLEDGEMENTS

This research has been conducted in cooperation with the Product Group MR of Philips Medical Systems Nederland B.V. We thank the interviewees from Philips Medical Systems for their time and contributions, Jan Gerben Wijnstra from Philips Research Laboratories for his feedback on the study and the anonymous referees for their comments and suggestions on earlier versions of this paper.

REFERENCES

33. Griss ML, Favaro J, d’Allessandro M. Integrating Feature Modeling with the RSEB. *Fifth International Conference on Software Reuse* 1998; 76–85.