On the Notion of Variability in Software Product Lines

Jilles van Gurp, Jan Bosch
Department of Mathematics and Computing Science, University of Groningen, PO Box 800, 9700 AV The Netherlands,
[jilles|jan.bosch]@cs.rug.nl, http://www.cs.rug.nl/Research/SE

Mikael Svahnberg
Department of Software Engineering & Computer Science, Blekinge Technical University, 372 25 Ronneby, Sweden
msv@ipd.hk-r.se, http://www.ipd.hk-r.se/msv

Abstract

In this paper, we discuss the notion of variability. We have experienced that this concept has so far been underdefined. Although, we have observed that variability techniques become increasingly important. A clear indication of this trend is the recent emergence of software product lines. Software product lines are large, industrial software systems intended to specialize into specific software products. Our contribution in this paper is that we provide the reader with a framework of terminology and concepts regarding variability. In addition, we present three recurring patterns of variability. Finally, we suggest a method for managing variability in software product lines.

1 Introduction

Over time, variability in software assets has become increasingly important in software engineering. Whereas software systems originally were relatively static and it was accepted that any required change would demand potentially extensive editing of the existing source code, this is no longer acceptable for contemporary software systems. Instead, although covering a wide variety in suggested solutions, newer approaches to software design share as a common denominator that the point at which design decisions concerning the supported functionality and quality requirements are made is delayed to later stages.

A typical example of such delayed design decisions is provided by software product lines. Rather than deciding on what product to build beforehand, in software product lines, a software architecture and set of components is defined and implemented that can be configured to match the requirements of a family of software products. A second example is the emergence of software systems that can dynamically adapt their behavior at run-time, either by selecting alternatives embedded in the software system or by accepting new code modules during operation, e.g. plug-and-play functionality. These systems are required to contain so-called 'dynamic software architectures' [20].

The consequence of the developments described above is that whereas earlier decisions concerning the actual functionality provided by the software system were made during requirement specification and had no effect on the software system itself, new software systems are required to employ various variability mechanisms that allow the software architects and engineers to delay the decisions concerning the variants to the point in the development cycle that optimizes overall business goals. For example, in some cases, this leads to the situation where the decision concerning some variation points is delayed until run-time, resulting in customer- or user-performed configuration of the software system. In other cases, variability can be handled before compilation, thus removing complexity of the final product.

Figure 1 illustrates how the variability of a software system is constrained during development. The space between the arrows of the funnel denotes the amount of variability in the system. When the development starts, there are no constraints on the system (i.e. any system can be built). This is visualized in Figure 1 by having inifinite space between the arrows. During development, the number of potential systems decreases (so there is increasingly less variability) until finally at run-time there is exactly one system (i.e. the running and configured system). At each step in the development, design decisions are made. Each decision constrains the number of possible systems (this is also argued in [13]). When software product lines are considered, it is beneficial to delay some decisions so that products implemented using the shared product line assets can be varied. We refer to these delayed design decisions as variation points.

Figure 1 displays two stereotypical variability funnels. One represents a situation where a lot of variability is removed from the system early on (left), the other one rep-

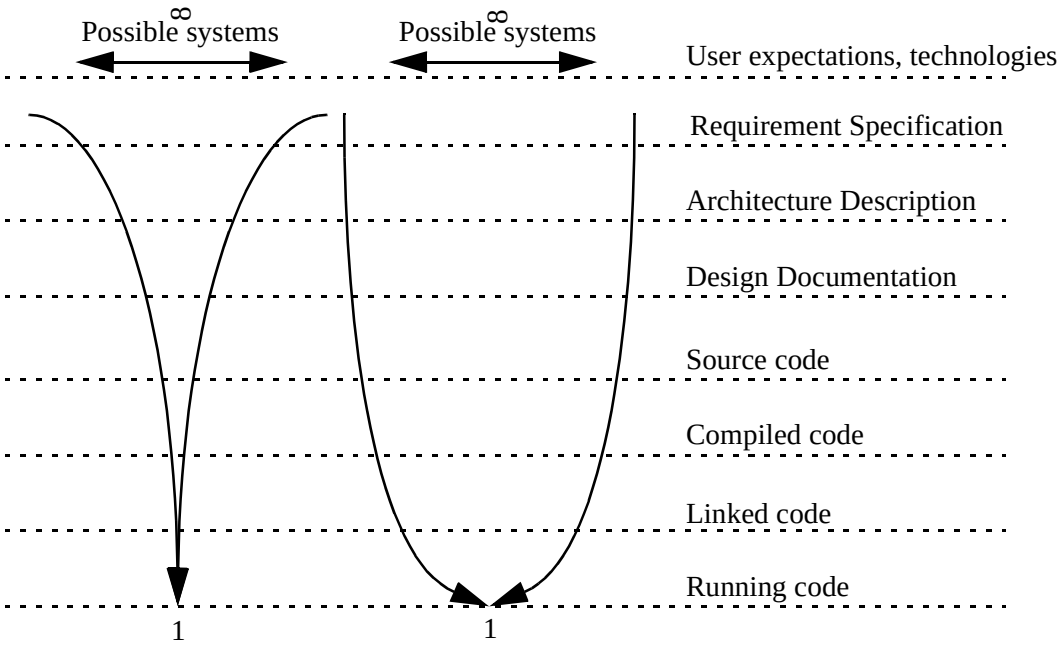

Figure 1 The Variability Funnel with early and delayed variability

resents a situation where a lot of effort has been made to preserve variability until very late in the development process. Arguably, the left funnel system is easier to develop, however the right funnel system provides greater reusability and flexibility.

1.1 Software Product Lines

The goal of a software product line is to minimize the cost of developing and evolving software products that are part of a product family. A software product line captures commonalities between software products for the product family. By using a software product line, product developers are able to focus on product specific issues rather than issues that are common to all products.

The process of creating a specific software product using a software product line is referred to as product instantiation. Typically there are multiple relatively independent development cycles in companies that use software product lines: one for the software product line itself (often referred to as domain engineering); and one for each product instantiation.

Instantiating a software product line typically means taking a snapshot of the current software product line and using that as a starting point for developing a product. Basically, there are two steps in the instantiation:

- **Selection.** In this phase the architecture is stripped from all unneeded functionality. Where possible pre-implemented variants are selected for the variation points in the software product line.
- **Extension.** In this phase additional variants are created for the remaining variation points.

From this we can see that there are two conflicting goals for a product line. On one hand a product line has to be flexible in order to allow for diverse product line instantiations. On the other hand a product line has to provide functionality that can be used out of the box to create products with a minimal effort. The conflict is that the process of customizing the product line is costly and consequently it is simpler to develop conrete products that fall within the scope of the product line if the product line offers just enough variabiliy. The scope of the product line widens if more variability is added to the product line. However this also increases the cost of product derivation.

1.2 Problem statement

The increased use of variability mechanisms is a trend that has been present in software engineering for a long time, but typically ad-hoc solutions have been proposed and used. To the best of our knowledge, few attempts have been made to organize the existing approaches and mechanisms in a framework or taxonomy, nor suggested design principles for selecting appropriate techniques for achieving variability. The aim and contribution of this paper is to address this problem by providing a set of concepts and terminology as well as a process for managing variability.

1.3. Related work

Software Product Lines. Our work was largely inspired

by earlier work in our research group. One of the authors published a book about designing and using software product lines [5]. This book was largely based on case studies and experience reports such as [3, 4, 23, 24]. From these reports we learned that evolution in software product lines is more complicated than in stand alone products because of the dependencies between the various products and because of the fact that there may be conflicting requirements between the different products.

Empirical research such as [22], suggests that a software product line approach stimulates reuse in organizations. In addition, a follow up paper by [21] provides empirical evidence for the hypothesis that organizations get most reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product line developing companies to invest time and money in performing methods such as in Section 4.

Variability Patterns. We were not the first to look for variability patterns. In [16], patterns are used to model variability in product families. Unlike us, they limit themselves to the detailed design phase. Instead we try to cover the entire development process, thus gaining the advantage of discovering variation points earlier (as pointed out above).

Also in [13], a number of variability mechanisms are discussed. This book also discusses how subsequent design decicions remove variability from an architecture. However it fails to put these mechanisms in the context of a variation management method like we do. Also, variability is not linked to features. This is an important characteristic of our approach as it is an important means for early identification (i.e. before architecture design) of variability needs in the future system.

Requirements. Our argument for introducing the external feature in Section 2 is based on [25]. They argue that a requirement specification should contain nothing but information about the environment. The rationale behind this is that a requirement specification should not be biased by implementation. Since features are an interpretation of the requirements, there is a need to map implementation independent requirements to implementation aware features.

Feature Modeling. Our extended feature graph is based on the work presented in [9]. The main difference, aside from graphical differences, between our notation and theirs is the external feature and the addition of binding time. In [10] the feature graph notation is used as an important asset in a method for implementing software product lines. Combined with our management method, the feature graph notation may be an important contribution to building software product lines.

Also related is the FODA method discussed in [14]. In this domain analysis method, feature graphs play an important role. The FORM method presented in [15] can be seen as an elaboration of this method. In this work feature graphs are recognized as a tool for identifying commonality between products. We take the point of view that it is more important to identify the variability between architectures than to identify the commonalities since the goal of developing a software product line is to be able to change the resulting system. In order to do that, the system has to be flexible enough to support the changes. The FORM method uses four layers to classify features (capability, operating environment, domain technology and implementation technique). We use a more fine-grained layering by using the different representations (architectural design, detailed design, source code, compiled code, linked code and running system) as abstractions. The advantage of this is that we can the relate variation points to different moments in the development. We consider this to be one of the contributions of our paper.

Our hierarchical feature graph bears some resemblance to the integral hierarchical and diversity model presented in [11]. Unlike their model, we use variation points to model variability. The notion of variation points was first introduced in [12]. Their model uses a similar layering as can be found in [1]. In this paper, three distinct granularities of reuse are identified (component, class and algorithm) that correspond to our architecture design, detailed design and implementation levels.

Feature interaction. Feature interaction can be modeled in a feature graph as dependencies between different features [10]. Since features can be seen as incremental units of development [8], dependencies make it impossible to link all features to a single component or class. As a consequence, source code of large systems such as software product lines tends to be tangled. Features that are associated with several other features are called crosscutting features. Variability in such features is very hard to implement and often requires that a system is designed using for example design patterns [10].

Methodology. Our method for managing variability bears some resemblance to the architecture development method outlined in [17]. The first steps in this method are to select a few cases to find major abstractions. Our method of creating a feature graph based on a number of cases in order to find variation points can be seen as a refinement of these steps.

Another method that is related to ours is the FAST (Family-Oriented Abstraction, Specification and Translation) method that is discussed in [7]. This empirically tested method uses the SCV (Scope, Commonality and Variability) analysis method to identify and document commonality and variability in a system. The result of this analysis is a textual document. A notation modeling variability in terms of features, such as provided in this paper,

is not used in their work. An important lesson learned in our paper is that variation points should be bound early in order to save on development cost.

1.4 Remainder of the paper

In the remainder of this paper we will in discuss features as a useful abstraction for describing variability (Section 2). After that we will introduce our framework of terminology (Section 3). In Section 4 we provide a method for managing variability. In Section 5 we illustrate our terminology with a few examples of variability techniques in the Mozilla browser architecture and we conclude our paper in Section 6.

2 Features: increments of evolution

One of the issues that need to be addressed is how to express variability. In this section we suggest that features are a useful abstraction for doing so. In [5], we define features as follows: "*a logical unit of behavior that is specified by a set of functional and quality requirements*". The point of view taken in this book is that a feature is a construct used to group related requirements ("*there should at least be an order of magnitude difference between the number of features and the number of requirements for a product line member*").

In other words, features are an abstraction from requirements. In our view, constructing a feature set is the first step of interpreting and ordering the requirements. In the process of constructing a feature set, the first design decisions about the future system are already taken. In [8], features are identified as units of incrementation as systems evolve. It is important to realize that there is an n to m relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features and that a particular feature typically meets more than one requirement (e.g. a functional requirement and a couple of quality requirements).

A software product line provides a central architecture that can be evolved and specialized into concrete products. The differences between those products can be discussed in terms of features. Consequently, a software product line must support variability for those features that tend to differ from product to product.

In [9] the following categorization of features is suggested:

- **Mandatory Features.** These are the features that identify a product. E.g. the ability type in a message and send it to the mail server is essential for an email client application.
- **Optional Features.** These are features that, when enabled, add some value to the core features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential feature and not all users will use it but it is nice to have it in the product.
- **Variant Features.** A variant feature is an abstraction for a set of related features (optional or mandatory). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients offer the feature of having a user configurable editor.

We have added a fourth category:

- **External Features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the OS on which the client runs.

Our choice of introducing external features is further motivated by [25]. In this work it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions from requirements, we need external features to link requirements to features. Using this categorization we have adapted the notation suggested by [9] to support external features. In addition we have integrated the notion of binding time which we will discuss in detail in Section 3. An example of our enhanced notation can be found in Figure 2. In this feature graph, the features of an email client are illustrated. The notation uses various constructs to indicate optional features; variant features in that exclude each other (xor) and variant features that may be used both (or).

The example in Figure 2 demonstrates how these different constructs can be used to indicate where variability is needed. The receive message feature, for instance, is a mandatory variant feature that has pop3 and imap as its variants. The choice as to which is used is delayed until runtime, meaning that users of the email client can configure to use either variant. Making this sort of details clear early on helps identify the spots in the system where variability is needed early on. The receive message feature might be implemented using an abstract receive message class that has two subclasses, one for each variant.

Our feature decomposition may give readers the impression that a conversion to a component design is straightforward. Unfortunately, due to a phenomenon called feature interaction, this is not true. Feature interaction is a well-known problem in specifying systems. It is virtually impossible to give a complete specification of a

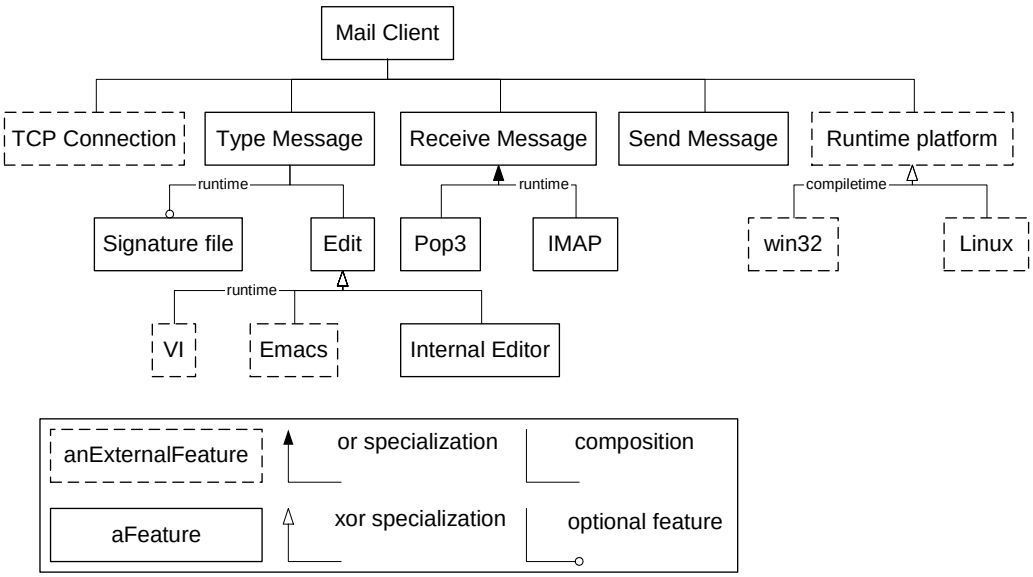

Figure 2 Example feature graph

system using features because the features cannot be considered independently. Adding or removing a feature to a system has an impact on other features. In [8], feature interaction is defined as a characteristic of *"a system whose complete behavior does not satisfy the separate specifications of all its features"*. In [10], the feature interaction problem is characterized as follows: *"The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved."*. This applies in particular to so-called crosscutting features (i.e. features that are applicable to classes and components throughout the entire system).

However, constructing feature diagrams may help developers identify which parts of their system need to support variability. Since the only prerequisite for building a feature diagram is the requirement specification, this can be done very early in the development process. Because of this, we argue that feature diagrams are an important tool for the management of variability.

3 Variability

Variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. Reusability and flexibility have been the driving forces behind the development of such techniques as object orientation, object oriented frameworks and software product lines. Consequently these techniques allow us to delay certain design decisions to a later point in the development.

Now that we are able to identify variability using the feature graph notation, we can examine the notion of variability more closely. We have found that when discussing a concrete variation point in a system, certain characteristics reappear. In this section we will introduce these characteristics and introduce suitable terminology.

3.1 Abstraction levels

During software development, a software system goes through a number of development phases. Each development phase has its own representations. One could say that development consists of transformations of these representations. E.g. a requirement specification is transformed in to a feature graph. After that, the feature graph forms the basis for the architecture design, which in turn forms the basis of the detailed design. After detailed design, source code is created. This source code is compiled, linked and finally run.

These different representations can be regarded as different abstraction levels of the system. Initially developers work with high-level models describing the requirements and features of the future system. Based on these high-level representations, the first design decisions are taken and an architecture design is created, etc. Consequently development can be characterized as going from abstract representations of a system to more concrete detailed descriptions. During each transformation design decisions are taken. But more importantly, some design decisions are delayed and left open for variability deliberately. These open design decisions are referred to as variation points.

In Figure 3, we have listed a number of representations

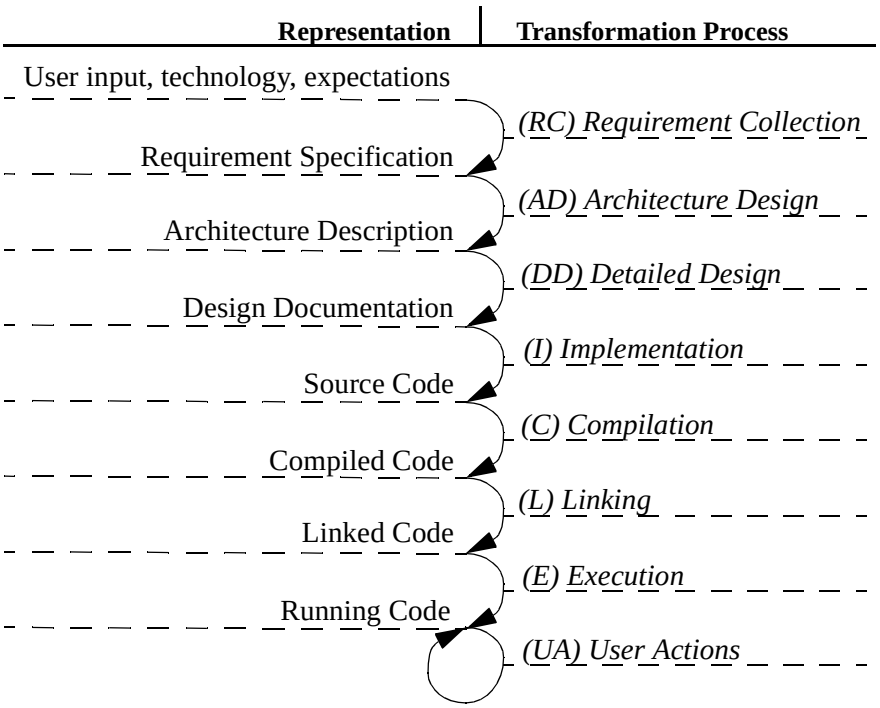

Figure 3 Representation & transformation processes

a system goes through and the associated processes that transform these representations. Note that transformation processes are not equal to development phases. Although, these transformation processes recurr in most development methods (for example iterative methods such as for instance Extreme Programming [2] but also waterfall model based methods) there are considerable differences between methods as to how much time and effort is spent on the various processes and when they are executed.

Especially for the later transformation processes it is very much technology dependent when these processes are executed. If we compare the use of an interpreted language like Python and a compiled language like C, we see that a C program is compiled and linked before product delivery whereas with Python compiling and linking are done while the system is executed. Yet, the variability techniques involved are very much the same. This also shows the advantage of an interpreted language: the user has more variability techniques at hand, simply because there are more transformations (i.e. compilation, linking) at run-time.

3.2 Variation point properties

Now that we have established that variability can be associated with different abstraction levels, we can introduce some additional properties of variability. A variation point can be in three, mutually exclusive states:

- **Implicit.** In Figure 1, we illustrated how during development a system is constrained. In the early phases of development there are many open design decisions, and consequently a there is a lot of variability. However, these decisions have not been deliberately left open so there is not a single point in the system that we can denote as a variation point. We refer to this type of variation points as implicit.
- **Designed.** As soon as the design decision is left open deliberately we say that the variation point is designed.
- **Bound.** The intention of designing a variation point in a system is to be able to insert a variant at a later stage. As soon as this happens, the variation point is bound to a variant.

Usually, when a variation point is designed there is also some idea about how and when variants are to be added to the system. In order to support this notion, we make a distinction between:

- **Open variation points.** Each variation point is associated with a set of variants that can be bound to it. In an open variation point, new variants may be added to this set.
- **Closed variation points.** In a closed variation point, no new variants can be added.

Usually, a variation point is only open in specific representations. An example of a variation point is an abstract class. This variation point is designed during detailed design. During detailed design it is also open since new subclasses can still be added during that phase. However,

after linking takes place the variation point is closed since it is impossible to add new subclasses to the system without at least re-linking the system. It should be noted that in modern programming languages, linking may be done dynamically in which case the variability point remains open. The consequence of this is that sub classes may be added after product delivery.

Using the properties defined in this section, we can accurately describe variability in a system. We can also compare and evaluate different techniques of implementing variability. In Section 5, we will do this for a number of techniques used in the Mozilla architecture.

3.3 Recurring patterns of variability

We have observed that when representation and development phase are abstracted from, variability follows certain patterns. To the best of our knowledge, variability always follows one of the following three patterns:

Single variant. With this pattern of variability, there is a set of variants. At binding time a single variant is picked from this set of available variants. Single variants can be identified in feature diagrams by looking for the xor specialization construct.

Optional variant. Optional variant is a special case of single variant since here the set of available variants only contains one variant and using it is optional. Optional variants are indicated in feature diagrams with an open circel on the relation end.

Multiple parallel variants. When multiple parallel variants are used, the variation point is not permanently bound to a variant but rather, the variant selection and binding process is executed every time the variation point is accessed. This type of variation point can be recognized by the use of the or specialization construct in feature diagrams.

Note that combinations of these patterns are possible, (e.g. an optional single variant). Using these patterns of variability and the properties of variation points, we can make a classification of different variability realization techniques. Unfortunately, doing so is beyond the scope of this paper. However, we are currently working on a paper discussing a taxonomy of variability mechanisms based on the patterns described above.

4 Variability management

Based on the previous sections, we suggest the following method for managing variability during the development that consists of the following steps:

- Identification
- Constraining
- Implementation
- Managing the variants

Identification of variability. The first step in the process is to identify where variability is needed. We suggest that the feature diagram notation we introduced in this paper is a good approach for doing so. From such a diagram, the important variation points can be identified.

Constraining variability. Once a variation point has been identified, it needs to be constrained. After all the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way. For constraining a variation point, the following activities need to take place:

- Choose a binding time for each variation point. Should the user be able to choose the variant or can developers do this before product delivery?
- Decide when and how variants are to be added to the system.
- Pick a variability pattern for each point. If the feature diagram notation was used, this information can be obtained from the diagram.
- Pick representation for realization of the variation point. Relevant for this decision is the way new variants are to be added.

Implementing variability. Based on the previous a suitable realization technique needs to be selected. In Section 5 we provided the reader with a few examples of such techniques. However, there are many more techniques. We intend to provide a taxonomy of mechanisms and techniques in future work. Providing such a taxonomy here would be beyond the scope of this paper.

Managing the variants. Depending on whether a variation point is open or not, some sort of variant management is needed. In some cases variants may be added manually. But it is also common for modern systems to download and install new variants over the internet. An example of a management in software is the XPInstall component in the Mozilla architecture (see Section 5). This component automates the downloading and installation of component variants. Especially when the multiple parallel variant pattern is used, a software management system will be needed to manage the variants.

5 Case study: Mozilla

As an example of variability in practice we analyzed the architecture of the Mozilla browser. The Mozilla browser has been developed as a so-called open source project. Consequently, information is readily available about its design. In addition, many variability techniques are applied in the Mozilla architecture, which makes it an interesting subject in the context of this paper.

The Mozilla project [18] was started in 1998 when

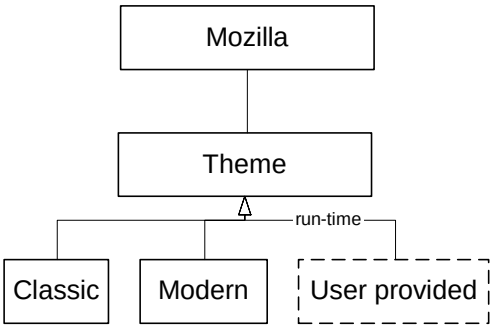

Figure 4 Theme support in Mozilla

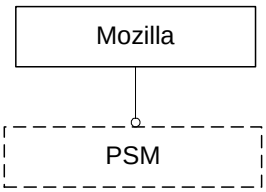

Figure 5 Mozilla's Personal Security Manager

Netscape [19] decided to make the source code of Netscape 4 available under an open source license. About half a year later, it was decided to redevelop the browser from scratch since the original source code was tangled beyond repair. At the moment of writing, the first commercial product based on the Mozilla source code (i.e. Netscape 6) has been released.

The main goal for the Mozilla project was not to provide a browser but rather a product line for building web applications. In the remainder of this section we will list a number of Mozilla features and analyze them, using the terminology and concepts introduced in this paper.

5.1 Mozilla features

Themes. An important feature of Mozilla is its support for user interface themes. Figure 4 illustrates this feature with a feature diagram. Mozilla implements the model view controller architectural pattern. Consequently the theme support variation point was designed during architectural design. As indicated by the feature diagram, this variation point is bound at run-time. By default two themes are bundled with Mozilla. However, users can download third party themes as well (i.e. the variation point is open at run-time). Since there has to be at least one theme (otherwise the application wouldn't have look and feel), the variation point follows the single variant pattern.

Security. Security in Mozilla is handled through a component called Personal Security Manager (PSM). This is an optional**.** component that can be added to the system by users (see Figure 5). The PSM provides such services as managing certificates for components, encryption/decryption of email messages etc. Variability for this feature was deliberately built into the architecture to allow third parties to add their own proprietary security components. Consequently, the security variation point was designed during architectural design. The variation point is bound at link-time. Although currently the PSM is the only available variant, the variation point is open at run-time (mozilla uses dynamic linking) so users can install a different security component should such an alternative become available.

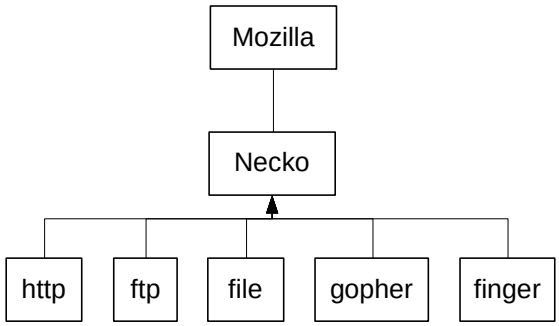

Figure 6 Necko

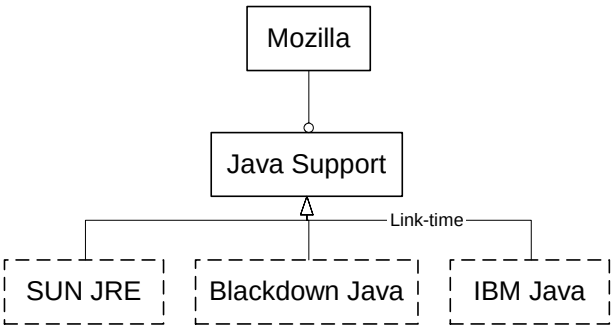

Figure 7 Java support in Mozilla

Network. A variation point that follows the multiple parallel variant pattern can be found in the way mozilla retrieves its files. Files are retrieved using the so-called Necko component (see Figure 6). This component uses URIs (uniform resource identifier) and protocol handlers to retrieve information from websites, ftp sites, the local filesystem, a jar file or any other supported protocol. The Necko variation point is designed during architecture design, it is open during detailed design and since it is an instance of the multiple parallel variant pattern, it is bound at run-time on a per call basis (i.e. each time something needs to be retrieved, a suitable protocol handler is bound to the variation point).

Java Support. Mozilla can optionally support Java. In Figure 7, we illustrate this feature with a feature diagram. From this figure we learn that there is a variation point in the Mozilla architecture for Java Support. Also, the variation point combines both the single variant pattern and the optional variant pattern. The binding of this variation point is optional and the variants are external to the architecture and binding typically happens at linking time. In the feature graph we list three common Java implementations

available under Linux.

In the implementation of Mozilla, all interaction with the JVM (java virtual machine) is done through the OJI (open java interface) interface. Since this interface was introduced during architecture design, the Java support variation point became designed during architecture design. Furthermore, since users can install OJI compliant java implementations, the variation point is open at run-time. The system is capable of downloading and linking the necessary binary components without requiring a restart of the application.

5.2 The underlying techniques

Of course the techniques used in Mozilla are not unique for Mozilla. Most of the mechanisms employed in Mozilla are based on common techniques. In this section we give a brief overview and indicate what their advantages are with respect to variability.

The broker pattern. Mozilla has its own component architecture XPCOM which closely resembles COM (the component infrastructure included with MS Windows). The XPCOM architecture is an instance of the broker pattern described in [6]. This pattern provides a variability mechanism following the 'single variant pattern' we describe in this paper. Rather than hard coding references between components, components have to request the broker (i.e. XPCOM) for a reference of a suitable component. This allows developers to replace the called component without having to change the calling component. It also allows them to provide more than one component for a given interface. The OJI interface discussed above is an example of an application of this technique. The browser accesses the JVM through this interface. Consequently, any OJI compliant JVM can be plugged into the XPCOM architecture.

Blackbox components. The main advantage of using the XPCOM architecture is that it forces developers to use XPCOM components in a blackbox fashion. The component bus constrains the use of a component to what has been specified in the IDL interfaces. This prevents that code of different components gets tangled too much. It also allows for delaying binding until linking rather than compilation. Since there are no source code dependencies between components, all dependency related variability is bound after compilation.

Dynamic binding. Another important technique is dynamic binding. Without dynamic binding, the system would not be able to use new components at run-time. The system would have to be shut down, patched recompiled and restarted each time a new component is registered with the XPCOM bus. Dynamic linking gives users the flexibility to use all variability techniques that are associated with linking. Traditionally, in statically linked systems these techniques had to be applied before product delivery, whereas with dynamic linking they can be applied after product delivery.

Scripting. A technique that goes beyond the use of dynamic binding is the use of interpreted languages. The advantage of interpreted languages over compiled languages in the context of variability is that scripts can be changed at run-time.

Domain specific languages. One prominent feature of the Mozilla architecture is the use of XML. Mozilla uses XML as a format for storing and exchanging structured data. Rather than specifying things like a user interface as C code or even javascript code, an XML representation called XUL is used. XUL is an example of a domain specific language (the domain in this case is user interfaces).

5.3 Summary

In this section we explained some of the variability techniques applied in the Mozilla architecture. The variation points we selected in the Mozilla architecture illustrate the three patterns we identified. A fourth example (i.e. java support) shows that the patterns can be combined in various ways. Using our terminology in combination with the feature diagram, we are able to discuss these techniques on a high level and without discussing any implementation details.

One of the observations we can make about variability in the Mozilla architecture is that most of the variation points are bound at run-time. Because of this, Mozilla is highly customizable. A second observation is that most variation points are kept open until after product delivery. Both observations fit in with the trend of delaying design decisions we illustrated in Figure 1.

6 Conclusion

The motivation for writing this paper was that we observed an increase in the application of various variability techniques. Furthermore we observed that these techniques are often applied in an adhoc fashion. This paper makes a number of contributions to address these issues:

- The main contribution of this paper is that it provides a framework of terminology and concepts regarding variabilitiy. Our framework of terminology provides the reader with the tools to describe variability in a software system in terms of variation points and variants. In addition we associate binding times with variation points. To the best of our knowledge this paper is the first that generalizes the notion of variability in such a way.

- A second contribution of our paper is the introduction of recurring patterns of variability.
- A third contribution is the variability management method described in Section 4. An integral part of our method is our adapted version of the feature graph notation first discussed in [9]. Our adaptations consist of adding binding time information to the feature graph constructs and the addition of the external feature construct.

Using our terminology, patterns and variability management method, software developers can recognize where variability is needed in their system early on and design their systems accordingly. Furthermore they can communicate their intentions with other developers and motivate design choices without going into detail about the implementation.

References

[1] D. Batory, S. O'Malley, "The Design and implementation of Hierarchical Software Systems with Reusable Components", in *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp. 355-398.

[2] K. Beck, "*Extreme Programming Explained*", Addison Wesley 1999.

[3] J. Bosch, "Product-Line Architectures in Industry: A Case Study", in *Proceedings of the 21st International Conference on Software Engineering*, November 1998.

[4] J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", in *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

[5] J. Bosch, "*Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*", Addison-Wesley, ISBN 020167494-7, 2000.

[6] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, "*Pattern-Oriented Software Architecture - A System of Patterns*", John Wiley & Sons, 1996.

[7] J. Coplien, D. Hoffman, D. Weiss, "Commonality and variability in software engineering", *IEEE Software*, November/December 1999, pp. 37-45.

[8] J. P. Gibson,"Feature Requirements Models: Understanding Interactions", in Feature Interactions In Telecommunications IV, Montreal, Canada, June 1997, IOS Press.

[9] M. L. Griss, J. Favaro, M. d'Alessandro, "Integrating feature modeling with the RSEB", *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

[10] M. L. Griss, "Implementing Product line Features with Component Reuse", to appear in *Proceedings of 6th International Conference on Software Reuse*, Vienna, Austria, June 2000.

[11] P. van de Hamer, F.J. van der Linden, A. saunders, H. te Sligte, "N Integral Hierarchy and Diversity Model for Describing Product Family architecture", in *Proceedings of the 2nd ARES Workshop: Development and evolution of Software Architectures for Product Families*, Springer Verlag, Berlin Germany, 1998.

[12] I. Jacobson, M. Griss, P. Johnson, "*Software Reuse: Architecture, Process and Organization for Business success*", Addison-Wesley, 1997.

[13] M. Jazayeri, A. Ran, P. Van der Linden, "*Software Architecture For Product Families: Putting Research into Practice*", Addison-Wesley, May 2000.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, "*Feature Oriented Domain Analysis (FODA) Feasibility Study*", Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegy Mellon University, Pittsburgh, PA.

[15] K.C. Kang, "FORM: a feature-oriented reuse method withdomain-specific architectures", in *Annals of Software Engineering*, V5, pp. 354-355.

[16] B. Keepence, M. Mannion, "Using Patterns to Model Variability in Product Families",in *IEEE Software*, July/ August 1999, pp 102-108.

[17] P.B. Kruchten, "The 4+1 View Model of Architecture", in *IEEE Software*, November 1995, pp. 42-50.

[18] Mozilla website, *http://www.mozilla.org/*.

[19] Netscape website, *http://www.netscape.com/*.

[20] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, "Self-Adaptive Software: An Architecture-based Approach", in *IEEE Intelligent Systems*, 1999.

[21] D. C. Rine, N. Nada, "An empirical study of a software reuse reference model", in *Information and Software Technology*, nr 42, pp. 47-65, Elsevier, 2000.

[22] D. C. Rine, R. M. Sonnemann, "Investments in reusable software. A study of software reuse investment success factors", in *The journal of systems and software*, nr. 41, pp 17-32, Elsevier, 1998.

[23] M. Svahnberg, J. Bosch, "Evolution in Software Product Lines: Two Cases", in *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422, 1999.

[24] M. Svahnberg, J. Bosch, "Characterizing Evolution in Product Line Architectures", in *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 92-97, 1999.

[25] P. Zave, M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, Vol. 6. No. 1, Januari 1997, p. 1-30.