

Towards Context-Aware Product-Family Architectures

Mohammed Salifu Bashar Nuseibeh Lucia Rapanotti
The Open University, Milton Keynes, UK
{M.Salifu, B.Nuseibeh, L.Rapanotti}@open.ac.uk

Abstract

The product-family paradigm is predicated on the definition of a general product architecture from which a number of different architectures can be derived, each addressing the needs of a separate market segment, with a high level of component reuse. The way this is achieved is through the identification of variability points within a generic product architecture, from which variant product architectures are generated, each product with a specific range of functionality.

This position paper proposes an approach to variability for software product-families that extends traditional product-family variability mechanisms to deal with context awareness. The approach is based on the identification of relevant relationships between context properties and user requirements on the one hand, and architectural variability types on the other; and the definition of design notations and principles for the representation and trade-off analysis between different variability types within a product-family architecture. In broad terms, the approach links requirements to software architectures using the product-family paradigm as the underlying organising principle. This is envisaged to provide context-aware product-family system developers a conceptual means to capture requirements and in linking such requirements to architectural design choices in the form of variability point types. This will contribute to software systems design and management during development and subsequent evolution.

Keywords

Product-line architectures; variability points; static adaptability; dynamic adaptability

1. Introduction

The notion of product-line informally describes a situation in which the supplier of a product seeks to maximise profit by identifying and supplying different

categories of the product to different segments of consumers of the product [5, 8, 25]. The process of identifying or defining segments of consumers is referred to as market segmentation [8].

The product-line paradigm is predicated on the definition of a general product architecture from which a number of different architectures can be derived, each architecture addressing the needs of a separate market segment, with a high level of component reuse. In order for the generic architecture [30] to support different market segments [5, 8], variability points [31, 37] are identified in the architecture at which the differences are realized. The list of possible architectural elements (components, connectors, constraints on them) from which a variability point will ultimately be assigned value(s) are referred to as variants [17, 18, 35, 36]. When the decision as to which variant to use at a variability point is taken and all other variants previously available at this point are discarded, the point is said to be bound or closed [18]. (An unbound variability point is referred to as an open point.) The realisation of individual products is achieved by statically binding the variability points before they are delivered to consumers. We refer to this type of variability points as static and the process of realising the individual products as static adaptability.

On the other hand, context-aware devices are expected to change their behaviour in response to changes in their operating environments or contexts. Reasons for changes in the operating environment are many, from fluctuating resources upon which the device relies (e.g., reduced bandwidth for a mobile phone) to different operating locations (e.g., a mobile user travelling long distance) or the presence of other devices (e.g., Bluetooth-enabled phones) [4, 12, 29, 34]. Changes may also be caused by users' preferences; for example, users of a mobile phone may require a particular set of features to be available to them while at work and a different set of features while at home. Adaptation to a changing environment may well require the restructuring of a device's software

architecture as different bindings of components may be involved. We refer to the restructuring of an application's architecture in response to contextual changes while the application is in operation as dynamic adaptability and the binding or unbinding of architectural elements as dynamic binding or unbinding.

There is an increasing expectation for software intensive devices to be context-aware, and many consumers' devices, such as mobile phones, which are developed as product families, are expected to follow this trend. To overcome the limitation of static binding, there is a need for approaches to product-family design which allow for both type of variability to be dealt with at the architectural level, as this is considered to be more efficient and cost effective [28]. Appropriate design notations and analytical tools will be required to enable trade-offs between static and dynamic variability points depending on characteristic of operating contexts and their associated requirements. This would allow appropriate variants to be chosen at variability points depending on the context for which they are intended. We argue that the definition of an architectural notation and a reasoning framework for dealing with various types of variability points and their instantiation, using a problem-oriented approach (such as problem frames [14]), is a necessary prerequisite to the successful design of product-families of context-aware software devices.

To the best of our knowledge, a framework which deals with these issues has yet to be developed. Therefore, we propose an approach that extends traditional product-family variability mechanisms to deal with context awareness. The approach is based on: the identification of relevant relationships between context properties and user requirements on the one hand, and architectural variability types on the other; and the definition of design notations and principles for the representation and trade-off analysis of different variability types within a product-family architecture.

The rest of this paper looks at background and related work and briefly describes our proposed approach.

2. Background & Related Work

This section briefly discusses current classification and representation of variability points and dependency relations between variants. This will be followed by a discussion of current approaches to reconfiguring product families.

2.1 Variability points and dependencies

Bachmann and Bass [2] have identified six sources of variability points. These are variations in function or features, such as a word-processor with a voice-recognition system (high-end) and one without (low-end); variation in the data used by components for communication, such as using a stack or queue and variation in control flow such as two components communicating using a publish-subscribe communication pattern [33] or a dedicated communication channel. Other sources of variabilities are variations in technology, variations in quality requirements and variations in the target operating platforms. Irrespective of the source of variability, variants available at variability points have been largely classified into three main types [2, 24, 37]: optional, single and multiple variants. An *optional* variant is included in some product members but not in others, for instance including a camera in a mobile phone. In the case of *single* variant, exactly one variant is chosen from a candidate set and all others discarded. In the case of *multiple* variants, more than one variant must be selected from a candidate list, for example dual communication protocols for a mobile phone, thereby giving it a wider area of operational access. There are various combinations of these basic types reported in [2]. This rigid classification is unsatisfactory in the presence of contextual changes as functionalities in a given set and for a single device may be classified as single variant (i.e. mutually exclusive) in one context and multiple variant (i.e. to be selected together) in another context.

Jaring and Bosch [17] have looked at the classification of variants and argued that whether a variant is optional or not changes based on relational dependencies. This is inline with earlier observation by [3] when they looked into using use cases [6] to communicate variability in product-family members to customers. In this context, a relational dependency refers to the changing of the type of a variant from optional to mandatory, or vice-versa, following the selection of another variant. That is, the selection of one variant may automatically trigger the inclusion of another variant which was previously classified as optional. For example, in the Nokia phone product-family, as currently implemented, the selection of the FM radio feature will automatically require the inclusion of the series 40 operating platform [26], as it is the only platform that supports this functionality. Therefore, at design time, it is necessary to consider all dependencies which derive from all contexts within which a product-family member is to operate. These include dependency relations between variants at a

given variability point and among variants at different variability points. Conflicts also need to be resolved.

The representation of variability points in software product-families, as currently practised, is achieved primarily through feature diagrams [16, 37]. A feature diagram shows the set of features available in the generic architecture from which the features of individual product members are selected. In this context, a feature is a functional unit that is visible to the consumer, such as a camera or FM radio in a mobile phone. The representation of variability point using use cases by Bühne et al. [3] is comparable to those based on feature diagrams. On the other hand, a graphical notation which is not based on feature has been proposed by Bachmann and Bass [2]. This is based on a generic UML (like) structure notation. What all these have in common is that, none of them explicitly consider the properties of the operating contexts and the domains existing in these contexts. We argue that the explicit representation of contextual properties and requirements, using a problem-oriented approach [14], is a prerequisite to successful development of context-aware product-family architectures.

In addition, all these notations are aimed at product-families whose architectures are statically bound and therefore made up of static variability points. Part of our work will seek to explore possible extensions for representing both static and dynamic variability points and their associated variants, using a problem-oriented approach.

2.2 Reconfigurable product families

There have been attempts [1, 10, 11, 21] at designing reconfigurable product-family architectures, each of which will be discussed next. The configuration of an architecture refers to its set of components, their interconnections and the constraints defining the behaviour of this architecture [13]. The replacement of such a configuration with a new (or different) one after it has been released or during the operation of the applications based on it, is referred to as reconfiguration [22, 28].

The work of Gomaa and Hussein [10, 11] is based on the use of architectural styles or patterns, such as the client server architectural style, to construct what they refer to as a reconfiguration pattern (based on the style of the generic architecture). A reconfiguration pattern is used to guide the process of automatically deriving one product-line member from a different one. This can be argued to be a generalised form of

parameterisation [30] as all instances of this product-family must conform to the style and different members are instantiated by changing the values of parameters. The focus of Gomaa and Hussein [10, 11] is on managing state transitions from one product member to a different member during reconfiguration, with no explicit discussion of the impact of dynamically changing the operating context on defining and reasoning about static and dynamic variability points. This approach fits the definition of static binding as the reconfiguration takes place as a one-off, and instantiated members do not undergo any dynamic adaptation of their structure during operation. It therefore appears not to be suitable for context-aware applications.

The work of Kim et al. [21] is similar to that of Gomaa and Hussein, hence exhibits the same limitations. The key difference is that Kim et al. provide an architectural description language for describing architectures and modifications to be applied to them during reconfiguration. The example in [21] adopts a pipe-and-filter architectural style and could therefore be argued to be a specialised case of the work of Gomaa and Hussein with the addition of a means to describe the architecture and its modifications. Again, as with Gomaa and Hussein, the focus of this approach is on managing the state transition from one product member to a different member during reconfiguration. Hence, it also fits the definition of static binding as members do not undergo any dynamic adaptation of their structure during operation. Therefore, this approach too appears not to be suitable for context-aware applications.

The work of Apel and Bohm [1] is based on the use of a layered architectural style to design a reconfigurable middleware for a context-aware environment. In this work, context-aware environment refers to an operating environment with network bandwidth fluctuations, connection interruptions, device mobility and resource-constrained devices. The services provided by this architecture are operating system-based and largely limited to the network infrastructure. In designing the reconfigurable middleware architecture, Apel et al. have adopted the product-line paradigm and produced a generic architecture from which specific architectures tailored to different environments are produced as and when the context requires, during runtime. Hence, this approach fits the definition of dynamic binding as the general architecture undergoes dynamic adaptation during operation. However, note that the whole generic architecture undergoes the dynamic adaptation and not the individual product members. This distinction is

significant as it implies that the entire generic architecture is loaded onto each target device and each adaptation represents a different product-family member satisfying the requirements of a different context. Therefore, this approach only applies to devices that are capable of running the entire generic architecture. To summarise, there is only one target context-aware generic architecture, loaded in all devices and the only notion of variance is in terms of differences in operating contexts.

We argue that this approach has severe limitations for developing consumer electronic devices, such as mobile phones, with which the success of the application of the product-family paradigm is largely associated. For example, using this approach to transform the current Nokia static product-family architecture into one that is context-aware, will require all the 1000 or so different handsets [23] to carry the same generic architecture (including its complete set of features). This is not realistic as different devices have different capabilities, such as memory size, screen size, processor speeds, operating platforms and hardware features. Hence, the architecture of every device must be optimised to take full advantage of its hardware resources. This explains why Nokia has at least five different platforms each with several versions [26] aimed at optimising the hardware resources of every device. It is also a requirement by the Open Mobile Alliance community-OMA (a consortium of mobile phone applications developers and operators, such as Nokia and Phillips) [27] that no single architecture is delivered to different devices with different hardware resources and features as this leads to some devices under utilising their resources while others are over burdened with software resources they never use. This requirement is stated as an architectural principle (#4) by the OMA as:

“It is not acceptable for OMA Architecture and Specifications to take a lowest common denominator approach that prevents use of advanced features in more capable devices and networks. At the same time, users of less capable devices should enjoy the best possible experience that those devices can provide...”[27]

We therefore argue the need to (partially) statically bind individual product-family members (containing only the specific required architecture) before they are delivered to consumers, and to dynamically bind these architectures during runtimes in response to changing contexts. Hence some variability points should be statically bound (e.g. defining the specific type of phone and architecture) while others dynamically

bound during operation in response to changes in context (i.e. adapting the specific architecture for the given context). The overall aim of our research is product-family architectures capable of supporting a mixture of both static and dynamic variability points.

3. Towards Context-Aware Product Family Architectures

In this section, we briefly describe the main objectives of our proposed research. These are:

1. to classify and represent variability point types and their application to context-aware devices;
2. to elicit, define, and represent relationships between context properties and requirements, and types of variability (points);
3. to develop conceptual tools for trade-off analysis in the choice of different types of variability points within a product family architecture;
4. to evaluate the approach through relevant exemplars and case studies.

Classification of variability points and their application to context-aware devices

This will be based on the exploration of existing work on variability points such as the works of Bachmann and Bass [2], Svahnberg et al [37], Bühne et al [3] and Jaring et al. [18], which are currently based on statically bound generic architectures, in view of possible extensions to cover dynamic variability points.

Variability points representation

This will require the exploration of current notations, such as that proposed in [2], used for static variability points representation, in view of a possible extension. Alternatively, a customisation of a UML structural notation may be adopted. The aim is a notation suitable for representing both static and dynamic variability points.

Definition, capture and representation of relationships between context properties and requirements, and types of variability (points)

To this end, we are proposing to adopt a problem-oriented approach based on Jackson’s problem frames [14] and their extensions (such as [32]). This would provide a conceptual framework for capturing context information and requirements, and relating them to architectural solution structures (in particular, types of variability points in the architecture). This is because of their underlying principle of separation between the problem domain (the context of a software solution and

the requirement) and the solution domain (where software resides) [15]. Key strengths of the approach could be argued to be its adoption of a multi-paradigm approach to problems and solutions descriptions [7] and a clear separation of indicative from optative descriptions; that is the separation of what is true of the development world with or without the presence of the software to be constructed and that of the desired behaviour wished upon the operating environment. The problem frames approach is preferred over other approaches such as goal-based [38] or scenarios [6], as these do not allow for an explicit representation of context and would therefore not be suitable in addressing the kind of contextual information we need to represent.

Definition of conceptual tools for trade-off analysis

We will explore current (generic) architectural techniques for trade-off analysis such as ATAM [20] or SAAM [19] for possible application to the analysis of variability points trade-offs.

Evaluation of the approach through relevant exemplars and case studies

To this end, we will elicit case studies found in the product-family literature such as those published as part of the “Engineering Software Architectures, Processes and Platforms for System-Families (ESAPS)” [9] European project (and where possible solicit industrial partner support) and use them both in the development and evaluation of our techniques.

4. Summary

Our research proposes an approach that extends traditional product-family variability mechanisms to deal with context awareness. As can be seen from the discussion so far, the proposed research touches three key research areas. These are (1) requirements and contextual properties impacting (or constraining) their satisfaction, (2) software architectures in the form product-families architectures and variability points mechanisms and (3) architecture level trade-off analysis. Due to space limitations, the related work considered is largely based on product-families and variability points, in terms of architectural design space. However, we argue that for a context-aware product-family development tool or conceptual framework to be useful, it must cover these three key areas.

As the research is in its early stage, it is impossible to say if the outcome will lead to an entirely new product-family development paradigm. However, we argue that at the very least, context-awareness will increase the

complexity of product-family based systems development. Even though the increased complexity may not be visible in all phases of software system management, it is likely to significantly affect the development and management of such systems during requirement specification through to implementation. It is these phases of system development that a reasoning framework, such as the one envisaged, will provide a means in dealing with the increased complexity.

References

1. Apel, S. and K. Böhm. *Towards the Development of Ubiquitous Middleware Product Lines*. in *Software Engineering and Middleware: 4th International Workshop, SEM 2004*. 2005. Linz, Austria.
2. Bachmann, F. and L. Bass. *Managing Variability in Software Architectures*. in *SSR'01*. 2001. Toronto, Ontario, Canada: ACM Press.
3. Bühne, S., G. Halmans, and K. Pohl. *Modelling Dependencies between Variation Points in Use Case Diagrams*. in *Proceedings of the 9th International Workshop on Requirements Engineering Foundation for Software Quality(REFSQ)*. 2003. Klagenfurt, Austria.
4. Chen, G. and D. Kotz, *A Survey of Context-Aware Mobile Computing Research*, in *Technical Report TR2000-381*. 2000, Dartmouth Computer Science.
5. Claycamp, H.J. and W.F. Massy, *A Theory of Market Segmentation*. *Journal of Marketing Research*, 1968. **5**(4): p. 388-394.
6. Cockburn, A., *Writing effective use cases*. 1st ed. 2001, Longman, Upper Saddle River, NJ: Addison-Wesley. 1-304.
7. Cox, K., Jon G. Hall, and L. Rapanotti, *A Roadmap of Problem Frames research*, in *To appear: Journal of Information and Software Technology*. 2005.
8. Dickson, P.R. and J.L. Ginter, *Market Segmentation, Product Differentiation, and Marketing Strategy*. *Journal of Marketing*, 1987. **51**(2): p. 1-10.
9. ESAPS, *Overview*. 2002, Engineering Software Architectures, Processes and Platforms for System-Families: <http://www.esi.es/esaps/overview.html>.
10. Gomaa, H. and M. Hussein, *Dynamic Software Reconfiguration in Software Product Families*. Lecture Notes in Computer Science, 2004. **3014/2004**: p. 435 - 444.
11. Gomaa, H. and M. Hussein. *Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures*. in *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*. 2004: IEEE CNF.
12. Harter, n., et al., *The Anatomy of a Context-Aware Application*. *Wireless Networks*, 2002. **8**(2-3): p. 187 - 197.
13. Hoek, A.v.d. *Configurable Software Architecture in Support of Configuration Management and Software Deployment*. in *INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*. 1999.

14. Jackson, M., *Problem Frames: Analyzing and structuring software development problems*. 1st ed. 2001b, New York, Oxford: Addison-Wesley. 390.
15. Jackson, M. *The World and the Machine*. in *ICSE'95*. 1995. Seattle, Washington, USA: ACM.
16. Jaring, M. and J. Bosch. *Representing Variability in Software Product Lines: A Case Study*. in *SPLC2 2002, LNC2379*. 2002: Springer-Verlag Berlin Heidelberg.
17. Jaring, M. and J. Bosch. *A Taxonomy and Hierarchy of Variability Dependencies in Software Product Family Engineering*. in *Proc. of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*. 2004: IEEE CNF.
18. Jaring, M., R.L. Krikharr, and J. Bosch, *Representing variabilities in a family of MRI scanners*. *Software-Practice And Experience*, 2004. **34**: p. 69-100.
19. Kazman, R., et al. *SAAM: A Method for Analyzing the Properties of Software Architectures*. in *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference*. 1994.
20. Kazman, R., Mark Klein, and Mario Barbacci. *The Architecture Tradeoff Analysis Method*. in *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference*. 1998.
21. Kim, M., J. Jeong, and S. Park. *From Product Lines to Self-Managed Systems: An Architecture-Based Runtime Reconfiguration Framework*. in *Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)*. 2005: ACM Press.
22. Kramer, J. and J. Magee, *The evolving philosophers problem: dynamic change management*. *IEEE Transactions on Software Engineering*, 1990. **16**(11).
23. Maccari, A. and A. Heie, *Managing infinite variability in mobile terminal software*. *SOFTWARE—PRACTICE AND EXPERIENCE*, 2005. **35**(6): p. 513–53.
24. Mannion, M., et al. *Representing Requirements on Generic Software in an Application Family Model*. in *ICSR*. 2000: Springer-Verlag.
25. Moorthy, K.S., *Market Segmentation, Self-Selection, and Product Line Design*. *Marketing Science*, 1984. **3**(4): p. 288-307.
26. Nokia, *Nokia Architecture - Platform Architectures*. 2005, Nokia - http://europe.nokia.com/nokia/0_62611_00.html; Online. p. 1-2.
27. OMA, O.M.A., *OMA Architecture Principles V1.2*. 2004, Open Mobile Alliance - OMA Ltd. p. 1-10.
28. Oreizy, P., et al., *An architecture-based approach to self-adaptive software*. *Intelligent Systems and Their Applications*, IEEE [see also *IEEE Intelligent Systems*, 1999. **14**(3)].
29. Pascoe, J. *Adding Generic Contextual Capabilities to Wearable Computers*. in *Digest of Papers. Second International Symposium on...* 1998. Pittsburgh, PA, USA.
30. Perry, D.E. *Generic Architecture Descriptions for Product Lines*. in *Lecture Notes in Computer Science*. 1998: Springer Berlin / Heidelberg.
31. Pohl, K., G. Böckle, and F.J.v.d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. 1 ed. 2005: Springer. 468.
32. Rapanotti, L., et al. *Architecture-driven Problem Decomposition*. in *Proceedings of the IEEE RE 04 conference*. 2004.
33. Rozanski, N. and E. Woods, *Software Systems Architecture*. 2005, London: Addison-Wesley Profession. 27-37.
34. Schilit, B.N., N. Adams, and R. Want. *Context-Aware Computing Applications*. in *Workshop on Mobile Computing Systems and Applications*. 1994.
35. Sinnema, M., S. Deelstra, and J. Bosch. *COVAMOF: A Framework for Modelling Variabilities in Software Product Families*. in *SPLC'04*. 2004: Springer-Verlag.
36. Sinnema, M., et al. *Modeling Dependencies in Product Families with COVAMOF*. in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*. 2006: IEEE CNF.
37. Svahnberg, M., J.v. Gorp, and J. Bosch, *A taxonomy of variability realization techniques*. *Software: Practice and Experience*, 2005. **35**(8): p. 705-754.
38. Van Lamsweerde, A., R. Darimont, and P. Massonet. *Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt*. in *Proceedings of the Second IEEE International Symposium on*. 1995.