Modelling Configurable Products and Software Product Families[*]

Tomi Männistö[1,2], Timo Soininen[1] and Reijo Sulonen[1]
[1]Helsinki University of Technology, Software Business and Engineering Institute
PO Box 9600, FIN-02015 HUT, Finland
[2]Visiting Nokia Research Center, Burlington, MA, USA

Abstract

Software product families are an emerging and important area of software engineering, whereas product configuration of traditional products, i.e., mechanical and electronic, has a slightly longer history as a specific area of research and business. This paper presents a preliminary comparison of concepts for modelling variety and evolution in both fields. This comparison shows remarkable similarities in these areas, but also leads to proposals on how results could be transferred from one area to the other.

1 Introduction

Product families, product configuration and product data management are research areas with great deal of practical importance for traditional products, i.e., mechanical and electronic products. In software engineering, product families have also become an active area of research as the number of variants of a software product have in many occasions increased due to various requirements from different markets, hardware platforms, customer specific individualisation, and so on.

This paper compares the work on traditional configurable products and work on software architecture in order, on one hand, to understand the differences of the fields and, on the other hand, to find potential areas for transfer of results between the fields. In addition to modelling variety, we also consider novel ways of capturing evolution within the same framework, which is hardly ever included in configuration modelling approaches of traditional products.

A *configurable product*, or a *product family*, is such that each *product individual* is adapted to the *requirements* of a particular customer order on the basis of a predefined *configuration model*, which describes the set of legal *product variants* [Sabin *et al.*, 1998; Soininen, 2000]. Because of combinatorial explosion, the number of legal variants of a traditional configurable product is typically large enough that listing them one by one is infeasible. A specification of a product individual, i.e., a *configuration*, is produced from the configuration model and particular customer requirements in a *configuration task*. The configuration task is routine, that is, the generation of the product individual does not involve creative design or design of new components.

Software product families with a large number of variants resemble traditional configurable products in many respects. Therefore, we believe that it is worth applying the results from the area of traditional products to that of software product families, and vice versa.

This paper focuses on viewing a software product as a configurable product family that potentially includes a very large number of variants. Such a software-engineering paradigm becomes relevant, for example, when software is embedded in a configurable product and the software must adapt to the hardware configuration. If the available memory is limited, the loaded software cannot include all possible variability and dynamically adapt to the hardware. Textbook examples of products for which software variability is important include alarm systems [Bosch, 2000], oscillators [Shaw *et al.*, 1996] and television sets [Jazayeri *et al.*, 2000]. Similar strategies are also currently sought for enterprise resource planning (ERP) systems, which are large configurable information system packages [Kumar *et al.*, 2000].

In the following, we first describe the field of product configuration of traditional products. Thereafter, we discuss modelling software product families and some possibilities of importing results from one area to the other.

2 Configurable Traditional Products

Configurable products clearly separate between the process of designing a product family and the process of generating a product individual according to the product configuration model. This places configurable products in between mass-products and one-of-a-kind products by enabling customer specific adaptation without losing all the economical benefits of mass-products [Tiihonen *et al.*, 1998].

Knowledge based systems for configuration tasks, product *configurators*, have recently become an important application of artificial intelligence techniques for companies selling products adapted to customer needs [Darr *et al.*, 1998; Faltings *et al.*, 1998]. The purpose of a configurator is to allow managing the configuration models and support the

[*] This paper is an extended and revised version of “Product Configuration View to Software Product Families” presented at Software Configuration Workshop (SCM-10) of ICSE01, Toronto, Canada, May 2001 [Männistö *et al.*, 2001].

configuration task. Product configuration tasks and configurators have been investigated for at least two decades [McDermott, 1982]. Several approaches have defined specific configuration domain oriented conceptual foundations. These include the three main conceptualisations of configuration knowledge as resource balancing [Heinrich *et al.*, 1991; Jüngst *et al.*, 1998], product structure [Cunis *et al.*, 1989; van Veen, 1991] and connections within a product [Mittal *et al.*, 1989; Soininen *et al.*, 1998]. Next, we first give an overview to a combined conceptualisation of configuration knowledge synthesising these approaches, reported in detail in [Soininen *et al.*, 1998; Soininen, 2000]. Then we elaborate on approach to modelling evolution of traditional configurable products. The basic approach has been reported in [Männistö *et al.*, 1996; Männistö, 2000], and it builds on the ideas of supporting design object versioning in the field of product design [Katz *et al.*, 1986].

2.1 Configuration modelling concepts

We believe that structured modelling is essential in keeping product configuration models understandable. Therefore, we concentrate here on configuration modelling methods that are based on explicit description of structural information in an object-oriented manner [Peltonen *et al.*, 1994; Männistö *et al.*, 1996; Soininen *et al.*, 1998]. We present the concepts for configuration modelling by means of slightly extended UML (Unified Modeling Language) [Fowler, 1997]. The details of the representation are not that important here, as we are primarily interested in the modelling concepts.

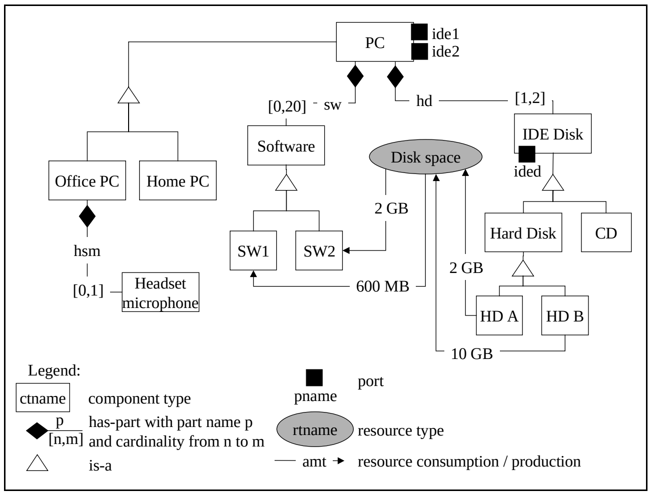

Figure 1. Illustration of configuration concepts by a simplified PC.

The modelling is based on *component types* that form an *is-a hierarchy* (for simplicity, only with single inheritance). Component types may have *property definitions*, such as part, port and resource production and resource usage definitions. Next, these main concepts and their intuitive semantics are very briefly introduced with some examples shown in Figure 1, which shows an imaginary PC product family.

Structural composition is modelled by means of *part definitions.* A part definition is augmented with *part name* and *cardinality*, e.g., 'PC' has one or two 'IDE Disk' as parts named 'hd'. Zero minimal cardinality represents an *optional part*. In addition, *alternative parts* may be defined. This is not shown in the example, but, for example, if there were more hard disk types, only some of them might be defined as alternatives for 'home PC'.

Connections between components can be modelled by defining *ports* (representing *interfaces*) for component types, with the idea that in a product individual each port may be *connected* to another port. In the figure, there are sample ports for connecting IDE controllers of 'PC' and 'IDE disk' (for simplicity, *port types* are not shown in the figure).

A configuration model may define *resources*, and component types may define that their individuals *produce* or *use* resources. For example, in the example hard disks produce disk space, whereas software uses it. In a legal configuration, the production and usage of resources should be *satisfied*. For example, there should be enough disk space to accommodate all software. In addition, a *context* may be defined in which a resource must be satisfied, e.g., the use of a resource may need to be satisfied by production within the same subsystem.

The definitions in component types are *inherited* in the is-a hierarchy. When 'PC' defines that it has part 'Disk', this means that also its *subtypes* 'Office PC' and 'Home PC' have 'Disk' as part.

The is-a hierarchy also induces variability. That is, defining a component type to be a part means that any of its subtypes can be a part. For example, either of the subtypes of 'Disk' is a valid choice as (type of) a part of 'PC'.

Configuration modelling languages also typically have a mechanism for expressing *conditions* or *constraints* on combining component types. Typical constraints are *incompatibility* and *requires*. That is, when two component types are incompatible, individuals of both types cannot occur in a legal configuration. If a component type requires another component type, a configuration containing an individual of the requiring type is legal only if it also contains individual of the required type.

In addition to what is shown in Figure 1, the configuration model typically also includes some means for expressing the *functions* (or *features*), as seen by the customers. These allow the customers to express their interest in more convenient terms than, for example, by directly selecting particular component types.

2.2 Evolution and component internal variation

The long-term management of configuration knowledge has always been a major problem for product configurators. One of the earliest examples, the XCON configurator, was reported to provide work for tens of developers and maintainers of configuration knowledge after a few years of operation [Barker *et al.*, 1989]. Therefore, it is also important to address the evolution in product family modelling.

In this section, we identify few steps in improving the management of configuration models.

Capturing the History

The First step is to capture the history. This can be done, for example, by *time stamping* everything in a configuration model when it changes. One actually needs two time stamps that form an *effectivity period*, which allows ending the effectivity of an object and principally also some planning for the future, if effectivities may extend to the future. With historical information one can reconstruct the configuration model at a particular point in time and also compare what has changed between two time points.

Capturing Versions and Variants

In addition, one needs to know when two objects of different times are in fact versions of the same thing. This is can be modelled by means of naming conventions, e.g., A-1.0 and A-2.0 could be different versions of the same A. To have more specific semantics for a related set of versions, the set, e.g., the A, can be explicitly represented by a *generic object*. We will next introduce the generic object and then elaborate further on the change management with the help of generic objects.

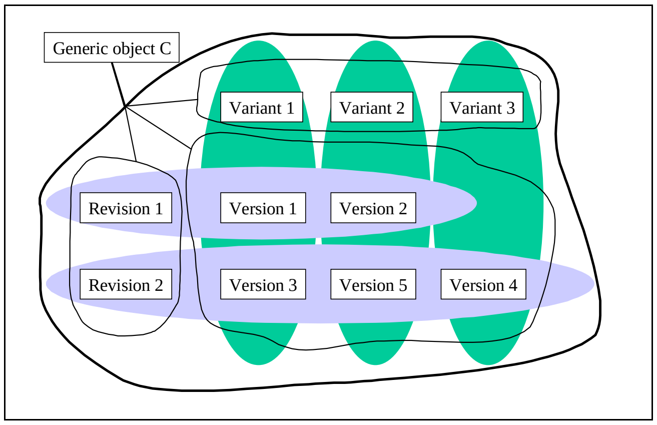

Figure 2. Generic object with versions, variants and revisions.

A *version* is a concept for capturing the state of a *generic object* at a particular time and the evolution of a generic object is captured by a set of versions [Katz *et al.*, 1986]. In order to be versions of the same generic object, the versions share something in common, such as an interface or substitutability in use, e.g., so that newer versions can be used in place of older versions. In a configuration model, component types could be seen as generic objects.

When variability with respect to some property of an object is also considered, versioning is divided to concepts *variant* and *revision*, i.e., parallel and consecutive versions of the generic object, respectively. Variants are intended to coexist, whereas revisions capture the evolution in time.

The concepts ‘variant’ and ‘revision’ organise the set of versions, as illustrated in Figure 2. For example, Variant 1 has its own revision history, and the explicit representation of revisions bears the notion that Version 3 and Version 5 are in some sense the corresponding revisions of Variant 1 and Variant 2, respectively. They may, e.g., have the same major error fixed or implement the same main functionality.

There is an ordering between revisions, e.g., Revision 2 *succeeds* Revision 1, which means that in Variant 2, Version 5 succeeds Version 2. This ordering of revisions is linear, instead of a tree or directed acyclic graph, as it is sometimes viewed, since variants are represented separately.

The generic object also serves as a point of reference to the set of versions. Such *generic references* do not specify the version; it is left open to be bound at an appropriate time. A *version binding mechanism* enables selecting both the appropriate variant and the revision of a generic component. For example, the selection could be based on a label, such as ‘1.1.2’, a time point or defaults, e.g., the “current revision of default variant”.

Variants of a generic object implement what we call *internal variation*, i.e., variation according to which variant of a selected component type is in the configuration. Another form of internal variation is *parameters* of a component type, such as the length of an axis or colour, for which the configuration task must determine appropriate values. Internal variation is in contrast to *structural variation*, which captures the choices between component types, as discussed in Section 2.1 in relation to Figure 1.

Capturing the Semantics of Changes to Configuration Models and Configurations

We will now take the next step in discussing the use of generic objects in modelling the evolution. The basic idea is to consider everything in a configuration model that evolves a generic object. This captures the evolution of the particular generic object, e.g., a component type. The interesting question is how the evolutions of different generic objects relate. For simplicity, we only consider the evolution of components in this and the following section.

First we separate the evolution of a configuration model and the evolution of the individuals created according to the configuration model. These evolutionary processes are in principle independent, that is, changes to the configuration model are not propagated to old individuals and individuals can evolve beyond their original configuration model. Therefore, one relation we consider in capturing the evolution is the relation of individuals to the configuration models, i.e., the *is-instance-of* relation.

In addition to is-instance-of relation, we are also particularly interested in two other relations, namely *has-part* and *is-a*. We want to specify, for example, what happens to component types that are parts or subtypes of a particular component type of which a new version is created.

Figure 3 illustrates the situation where component types as well as component individuals are represented as generic objects. For simplicity, we only consider one variant and assume that each change to a generic object is realised as a new version and a new revision. Thus, we have a one-to-one mapping between revisions and versions of a generic object, which is why only versions are shown in the figure. Each revision has an effectivity period and the effectivities of consecutive revisions of a generic object are required to meet. The effectivity of a version means the effectivity of the corresponding revision. Effectivities are not aligned between generic objects in the figure.

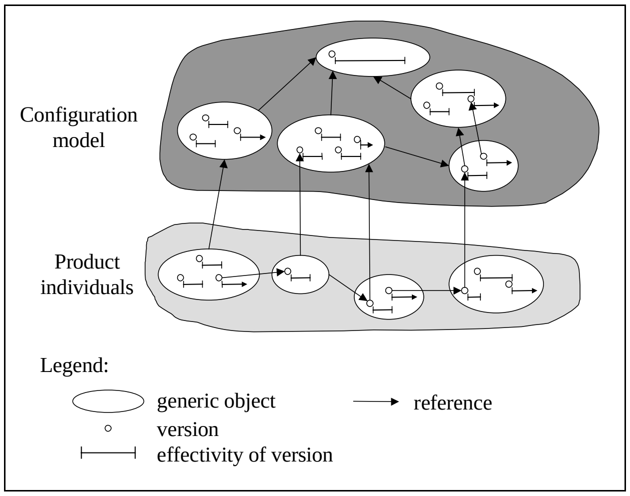

Figure 3. Evolution of types and individuals.

Relations are represented as references, which may be generic, i.e., refer to a generic object, or be a property of a generic object, i.e., start from a generic object, in addition to starting from or referring to a specific version.

By defining invariants on how the revisions of various generic objects relate we want to capture some semantics of the evolution of generic objects. For example, a company may decide that a new revision must be usable in place of old revisions it succeeds. This results in a convenient lack of need of propagating a change to wholes by creating new versions of them as well when a part is changed. This kind of change management policy can be captured by an invariant that only requires existence of an effective version in the whole for the entire effectivity of a the part, but does not require this separately for each version (which would require propagation of each change). Similar invariants can be defined for the other relations and policies as well, as has been done in [Männistö, 2000]; however, further discussion is omitted here for brevity.

Supporting Configuration Task over Time

After providing for capturing history and change management policies, the next step is to incorporate configuration models and configuration tasks that span time periods. References could specify multiple revisions, such as “revision $\leq$ 2.0 of A”. Consequently, the search space of a configuration task would contain versions of multiple revisions of a single generic object.

A configuration task spanning multiple time points may require additional assumptions along the lines of “only single revision of a generic object can be in a configuration”, “a version of a newer revision can always be used instead of an older one”, and so on. In addition, one would probably want to prefer solutions with versions of later revisions (assuming an ordering of solutions with respect to time exists), which preference could also guide the search itself.

As a next step further, one might want to support reconfiguration of old individuals over the history of configuration models. We do not, however, discuss that here.

Evolution of Other Concepts

The above discussion did not address the evolution of port types, resource types or constraints. We expect that port types and resource types can be treated quite similarly as component types, the interesting relations to consider being from a port type to a component type defining a port and from resource type to a component type defining a production or use of the resource.

The treatment of constraints depends on how they are modelled. Constraints can be seen as properties of component types that can refer to the parts of the component type. A constraint C referring to component types A and B could be placed to the least common ancestor of A and B in the compositional structure. This can be the root representing the whole product individual if the constraint is global. From change management perspective, however, this is somewhat odd because it is hard to see a change to the constraint C as a change to the whole product. Another alternative would be to consider constraints that are not clearly a property of any particular component type as global. One may argue that this is semantically equivalent to considering them properties of the whole product. From the change management viewpoint, however, there is a difference. Considering a constraint C as a generic object of its own means that changing it leads to a new revision of C and potentially the referred component types A and B. This seems to be closer to the intended semantics of the change than a change to the whole product.

3 Comparison to Modelling Software Product Families

There are many ways of achieving variety in software. Software variety can be implemented as a ‘universal’ software product that contains all variants, and can thus behave as any specific variant. An alternative approach is to use pre-processor directives to optionally include pieces of source code. This approach however, easily loses the big picture over the entire product family, as the representation of variation is distributed in the source code. With large software products, a typical adaptation approach is to take an existing variant as a basis and then modify it accordingly [Karhinen *et al.*, 1997]. Such “copy, paste and modify” of architectural components easily blurs the original ideas behind the architecture and consequently deteriorates the overall product architecture [Dikel *et al.*, 1997].

Variability may also be achieved by selecting appropriate components to a family architecture [Hayes-Roth *et al.*, 1995; Karhinen *et al.*, 1997; van Ommering *et al.*, 2000; Syrjänen, 2000; Wijnstra, 2000]. In the case of software product families, the architecture can thus be called a *product family software architecture* (PFSA). In fact, there are many different phases in the life cycle of a product individual (to distinguish from the life cycle of product family), e.g., construction of configuration, build and execution, for which different modelling concepts may be needed [Ran, 2000]. However, no generally applicable conceptual basis for modelling software product families seems to exist.

We will next compare the configuration modelling concepts from the previous section with software product family modelling.

3.1 Functions or Features

Customers view a product differently than its designers; customers prefer speaking of features that are of value to them, not necessarily about technical details of implementing the features. This means that a configuration model should allow customers to use features for inputting the system requirements. As there are approaches for modelling software product families on the basis of features [Hein *et al.*, 2000], it seems that from a modelling perspective, features of software seem to correspond to functions of traditional products.

3.2 Compositional structure and taxonomy

Compositional structure and taxonomy are very powerful and suitable means for representing configurable product families. The decomposition of a system into subsystems is also important for the management of software product families and their configuration knowledge. For example, work at Philips [Wijnstra, 2000] defines a skeletal architecture that has plug-in components and optional implementations of components (called units in the paper). This provides similar, although simpler, variation for a software product family as optional and alternative parts in a configuration model.

Van der Hoek et al. [1999; 2000] propose a model that also provides means for modelling the compositional structure and has great similarities with some of the concepts discussed above; the approach is discussed in more detail in the next section.

For managing component types and configuration knowledge, we also see great importance in organising component types in a class hierarchy with inheritance, especially with structural decomposition, ports and resources. Inheritance and subtyping provide powerful mechanisms for representing abstract modules and classifying varying component types. In the conceptualisation of Section 2.1, subtyping is used for modelling the variants of component types, whereas in software architecture modelling, it is often seen as a means for extending the architecture, i.e., a mechanism for supporting the evolution of component types [Medvidovic *et al.*, 2000].

3.3 Connections and Resources

There are many possibilities in conceptualising connections. The basic idea is to get components connected, which requires a point of connection in components. These are called ports in configuration modelling, similarly as in software architecture domain [see, e.g., Shaw *et al.*, 1996].

Koala architecture description language presents constructs for capturing the variability of software systems [van Ommering *et al.*, 2000]. The main mechanism for expressing the configuration knowledge in that approach is based on interfaces, i.e., ports, and their correct connections. The approach makes a distinction between internal variety of a component (they call it internal diversity of a component) and structural variety. For internal variation Koala offers diversity interfaces, which essentially describe the parameters by which the variant of the component (type) can be determined. For structural variety it offers switches that model alternative connections between interfaces.

The model of van der Hoek et al. [1999; 2000] includes component types and connectors, and supports variability and evolution. The optionality proposed in the approach takes the form of structural variety according to the terminology above. The variability of components, on the other hand, is explicitly represented by special variant components, which consist of a set of components with same interface. For a configuration, one variant is selected from the set of variants of the component according to some control variables. This corresponds to the internal variation of a component, as discussed above.

Another issue in modelling connections is whether separate entities are used to represent connections between the components as in architecture description languages or if components and their connections similar to configuration modelling suffice. When connectors are used, they also need points of connection, called *roles* [see, e.g., Shaw *et al.*, 1996]. Most software architecture description languages model *connectors* that connect component types as first-class entities [Medvidovic *et al.*, 2000], which is not the case in most configuration modelling approaches.

It may be that some mechanisms are also needed for describing the legal ways of attaching ports to roles. For this purpose, port types and role types may be defined, and the legal connections defined between them. If it seems appropriate, these concepts can be simplified by representing the connections only as a relation directly between ports.

To summarise, it seems that the concepts proposed for modelling software architecture and interfaces of software classes or components are rather similar to the concepts proposed for traditional products and it should be easy to utilise results from the latter to model and manage software product families. On the other hand, modelling of connections is one potential area in which results could be transferred from software architectures to configuration modelling of traditional products.

Furthermore, in certain cases, such as method calls to components, it may be adequate to represent the interfaces as resources and only check that all needs are satisfied without making the explicit connections. Module Interconnection Languages (MIL) use resources similar to those described above for matching modules by interfaces of services they provide and need [see, e.g., Shaw *et al.*, 1996]. In configuration modelling, the satisfaction of resources, however, is typically more complex than simple matching. There may be multiple products and users of the same resource and thus the amounts produces and uses must be calculated, possibly within a context. Combining resources and compositional structure seems a new interesting way of modelling software product families.

3.4 Constraints

Additional constraints describe the valid configurations, e.g., by pruning out ones including component individuals of incompatible component types. For software, additional constraint could specify dynamic behaviour. However, constructing detailed behavioural models is a remarkable effort and may require formalisms that make them computationally infeasible. In configuration modelling of traditional products, the computational complexity has been studied and a working balance between expressive power can be established—e.g., it is not necessary to model the behaviour of the products for configuration and product data management purposes [Soininen, 2000].

3.5 Evolution

In general, the software architecture for a product family is seen as a mechanism for enabling software reuse and introduction of new variants to the product family [Bosch, 2000]. Most approaches to modelling traditional configurable products, however, have only considered the representation of the current variation of the product family. That is, they do not have concepts for modelling evolution. Evolution of a product family, however, is inevitable and, therefore, we feel it should be taken seriously in configuration modelling. In fact, evolution is one area in which the ideas and results from software product families could be useful for traditional products.

The evolution in Koala is defined to preserve the stability of interfaces, which corresponds well with the semantics the generic objects are meant to capture [van Ommering *et al.*, 2000]. That is, the new versions may be created for the component as long as the common part described in the generic object, e.g., the interface, remains.

In the approach by van der Hoek et al [1999; 2000], variability and revisioning are separated in a manner that is semantically equivalent to generic object of Figure 2. Our proposal is in this respect a superset of their approach, and allows, e.g., the variability to be represented as structural or internal.

Other research in software configuration management has also addressed the orthogonal nature of revisions and variants. An *n*-dimensional grid is one way of representing variants of software [Conradi *et al.*, 1997]. Estublier and Casallas [1995] identified the dimensions: historical (i.e., revisions), logical (i.e., variants) and co-operative (i.e., concurrent work intended to be merged) for the version space. VOODOO system, on the other hand, models versions by a cube that has the following dimensions: components, revisions and variants [Reichenberger, 1995].

Generic objects seem suitable for modelling the evolution of software configuration models. They make the explicit distinction between revisions and variants of single design objects. Furthermore, as in traditional products, generic references for modelling compositional structure with a version binding mechanism can also be used for software. As suggested for traditional products, generic objects may be used in modelling the compositional structure of software product families as well as the evolution with respect to taxonomy, although this area requires further research [Männistö, 2000].

4 Conclusions

There are remarkable similarities between the concepts proposed for modelling software architectures or software product families and those used for modelling configurable traditional products. For traditional products, various methods exist that allow modelling product families essentially with the concepts of Figure 1, with some approach-specific variations. These concepts for modelling the variety of traditional products seem to suit modelling software product families as well. Utilising the concepts from traditional product families and adapting them for representing the architecture and variation of software product families, we believe, would lead to concise and manageable models. In addition, such models would open a way to using the AI methods developed in the field of product configuration to support the generation of product variants on the basis of the models.

The potential areas of transferring knowledge from configuration of traditional products to software product families include the following.

First is the appropriate level of abstraction in representing product families. In traditional products, kinematics and stress analyses, for example, are abstracted away from configuration models. How about software—What is the appropriate level of abstraction of dynamic behaviour for software product families? Does this change if configuration modelling is extended to capture the dynamic re-configuration of software?

Second, the conceptualisation of connections requires investigating whether the more complex concepts are needed or if a simpler conceptualisation is adequate.

Further, one issue to consider is whether the integration of compositional structure with taxonomy and resources would provide practically feasible product family modelling tools for software engineers and architects.

In general, it is rather safe to say that methods for modelling and management of variation of traditional products is more advanced than that of software product families. Nevertheless, two major areas were identified for transferring results from software product families to traditional configurable products.

First, connectors are modelled in more detail in software architectures than in traditional products, and thus, there is potential for utilising that work for modelling traditional configurable products. For example, explicit representation of connector types and individuals, and possibly also of the information that is transferred via each connection could also be useful for some traditional products.

Second, the evolution of product family descriptions (or architectures) has received more emphasis in software product families than in modelling of traditional configurable products. The incorporation of evolution to product family modelling is of great importance. However, regardless of much work in modelling evolution in various areas, including design data modelling, product data management, software configuration management, schema evolution of databases and temporal databases, there is still plenty to be done before a mature practice for capturing the evolution of product families can be defined.

Acknowledgements

We gratefully acknowledge the financial support of the Academy of Finland (grant 51394). We also thank Alexander Ran for his valuable comments on an earlier draft of the paper.

References

Barker, V.E., O'Connor, D.E., Expert systems for configuration at Digital: XCON and beyond, *CACM,* 32(3):298–318, 1989.

Bosch, J., *Design and use of software architectures—adopting and evolving a product-line approach,* Addison-Wesley, 2000.

Conradi, R., Westfechtel, B., Towards a Uniform Version Model for Software Management. In *Proc. of SCM-97, LNCS 1235,* pages 1–17, 1997. Springer.

Cunis, R., Günter, A., Syska, I., Peters, H., Bode, H., PLAKON — An approach to domain-independent construction. In *Proc. of IEA/AIE-89,* pages 866–874, 1989.

Darr, T., McGuinness, D., Klein, M., Special Issue on Configuration Design. *AI EDAM 12,* 1998.

Dikel, D., Kane, D., Ornburn, S., Loftus, W., Wilson, J., Applying software product-line architecture, *Computer,* 30(8):49–61, 1997.

Estublier, J., Casallas, R., Three dimensional versioning. In *ICSE SCM-4 and SCM-5 Workshops, LNCS 1005,* J.Estublier, ed. pages 118–135, 1995. Springer-Verlag.

Faltings, B., Freuder, E.C., Guest editors' introduction: Configuration, *IEEE intelligent systems & their applications,* 13(4), 1998.

Fowler, M., *UML distilled Applying the standard object modeling language,* Addison-Wesley, 1997.

Hayes-Roth, B., Pfelger, K., Lalanda, P., Morignot, P., Blabanovic, M., A domain-specific software architecture for adaptive intelligent systems, *IEEE Transactions on software engineering,* 21(4):288–301, 1995.

Hein, A., Schlick, M., Vinga-Martins, R., Applying feature models in industrial settings. In *Software product lines—Experience and research directions,* pages 47–70, 2000. Kluwer Academic Publishers.

Heinrich, M., Jüngst, W., A resource-based paradigm for the configuring of technical systems from modular components. In *Proc. of the seventh IEEE conference on artificial intelligence applications,* pages 257–264, 1991.

Jazayeri, M., Ran, A., van den Linden, F., *Software architecture for product families: Principles and practice,* Addison Wesley, 2000.

Jüngst, W., Heinrich, M., Using resource balancing to configure modular systems, *IEEE intelligent systems & their applications,* 13(4):50–58, 1998.

Karhinen, A., Ran, A., Tallgren, T., Configuring design for reuse. In *Proceedings ICSE'97,* pages 701–710, 1997.

Katz, R.H., Chang, E., Bhateja, R., Version modeling concepts for computer-aided design databases. In *Proceedings of the SIGMOD,* pages 379–386, 1986.

Kumar, K., van Hillegersberg, J., Enterprise resource planning—experiences and evolution, *CACM,* 43(4):22–26, 2000.

Männistö, T., A conceptual modelling approach to product families and their evolution. Doctoral thesis. Helsinki University of Technology, 2000.

Männistö, T., Peltonen, H., Sulonen, R., View to product configuration knowledge modelling and evolution. In *Configuration—papers from the 1996 AAAI Fall Symposium,* pages 111–118, 1996. AAAI Press.

Männistö, T., Soininen, T., Sulonen, R., Product Configuration View to Software Product Families. In *Software Configuration Management Workshop (SCM-10),* 2001.

McDermott, J., R1: a rule-based configurer of computer systems, *Artificial Intelligence,* 19(1), 1982.

Medvidovic, N., Taylor, R.N., A classification and comparison framework for software architecture description languages, *IEEE Transactions on software engineering,* 26(1):70–93, 2000.

Mittal, S., Frayman, F., Towards a generic model of configuration tasks. In *Proc. of IJCAI,* pages 1395–1401, 1989.

Peltonen, H., Männistö, T., Alho, K., Sulonen, R., Product configurations—An application for prototype object approach. In *Proc. ECOOP '94,* LNCS 0821, pages 513–534, 1994. Springer-Verlag.

Ran, A., ARES conceptual framework for software architecture, *in: Software Architecture for Product Families,* pages 1–29, 2000. Addison Wesley.

Reichenberger, C., VOODOO A Tool for orthogonal version management. In *Proc. SCM-4 and SCM-5, LNCS. 1005,* pages 61–79, 1995. Springer-Verlag.

Sabin, D., Weigel, R., Product configuration Frameworks—A survey, *IEEE intelligent systems & their applications,* 13(4):42–49, 1998.

Shaw, M., Garlan, D., *Software architecture - Perspectives on an emerging discipline,* Prentice-Hall, 1996.

Soininen, T., An approach to knowledge representation and reasoning for product configuration tasks. Doctoral thesis. Helsinki University of Technology. 2000.

Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R., Towards a General Ontology of Configuration, *AI EDAM,* 12(4):357–372, 1998.

Syrjänen, T., A rule-based formal model for software configuration. Master's thesis. Helsinki University of Technology, 2000.

Tiihonen, J., Soininen, T., Männistö, T., Sulonen, R., Configurable products—Lessons learned from the Finnish Industry. In *Proc. of 2nd EDA,* 1998.

van der Hoek, A., Capturing product line architectures. In *Proc. of Intern. Software Architecture Workshop,* 2000.

van der Hoek, A., Heimbigner, D., Wolf, A.L., Capturing architectural configurability: variants, options, and evolution. CU-CS-895–99. 1999.

van Ommering, R., van den Linden, F., Kramer, J., Magee, J., The Koala component model for consumer electronics software, *Computer,* (March):78–85, 2000.

van Veen, E.A., Modelling product structures by generic Bills-of-Material. Doctoral thesis. Eindhoven University of Technology. 1991

Wijnstra, J.G., Supporting diversity with component frameworks as architectural elements. In *Proc. of ICSE00,* pages 50–59, 2000.