Modelling Configuration Knowledge in Heterogeneous Product Families

Matthieu Quéva[1] and **Tomi Männistö**[2] and **Laurent Ricci**[3] and **Christian W. Probst**[1]
[1]DTU Informatics, {mq,probst}@imm.dtu.dk
[2]Aalto University, tomi.mannisto@aalto.fi
[3]Microsoft Development Center Copenhagen, lricci@microsoft.com

Abstract

Product configuration systems play an important role in the development of Mass Customisation. The configuration of complex product families may nowadays involve multiple design disciplines, e.g. hardware, software and services. In this paper, we present a conceptual approach for modelling the variability in such heterogeneous product families. Our approach is based on a framework that aims to cater for the different stakeholders involved in the modelling and management of the product family. The modelling approach is centred around the concepts of views, types and constraints and is illustrated by a motivation example. Furthermore, as a proof of concept, a prototype has been implemented for configuring a non-trivial heterogeneous product family.

1 Introduction

In many companies, there has been an increasing need to reduce the costs while offering highly customised products. Indeed, today's customers demand products with lower prices, higher quality and faster delivery, but they also want products customised to match their unique needs. Product configuration systems (or configurators) have allowed the manufacturers to adapt their business model to Mass Customisation [Pine, 1993] and propose products with hundreds of product features and options for a competitive price.

Model-based configuration is based on a strict separation between the product knowledge (i.e. the data representing the characteristics of the products) and the problem solving knowledge (i.e. the mechanisms used to ensure the consistency of the customised product). As the solving process is independent from the product knowledge, this separation provides a good robustness, compositionality and reusability, making model-based systems the prime choice for configuring large and more complex models [Sabin and Weigel, 1998].

Most of the research on product knowledge modelling has concentrated on manufactured product families [Hvam *et al.*, 2008]. Moreover, configuration techniques have recently been applied to other types of products, such as software variability [Asikainen *et al.*, 2007] or configurable services [Heiskala *et al.*, 2005]. However, many products nowadays are heterogeneous, i.e. different design disciplines are taken into account within the *same* product family. Modelling such products raises two main issues. One must first consider how to structure the different kind of knowledge that needs to be modelled. A second issue concerns the evolution of the product family. The set of features provided by a product family varies according to where and when it is distributed.

In this paper, we present a framework for modelling heterogeneous product families, based on *modelling views*. This framework synthesizes, unifies and extends different approaches to modelling configuration in the different design disciplines, e.g. physical products, software or services. The different views used in the approach are described using UML metamodels, together with different types of constraints that govern the dependencies both within and between views.

Section 2 and Section 3 introduce the necessary background and the research problem behind our approach. Section 4, 5 and Section 6 present concepts and constraints involved in our framework. Section 7 provides a brief overview of the proof of concept for our work, while Section 8 discusses our results and related work. Finally, Section 9 concludes the paper.

2 Background and Previous Work

This section provides a brief overview of different research areas on which this work is based.

2.1 Product Configuration

Product configuration modelling is widely based on concepts such as *components*, *ports*, *resources* and *functions* [Soininen *et al.*, 1998]. A *configurable product* is composed by components that are connected together via ports to form a hierarchical partonomy structure. Specialisation relations also permits to create a taxonomy structure in the model. Resources are balanced entities that can be produced or consumed by components, while functions can be used to define the product from the point of view of what functionalities it provides. The model also contains *constraints* that limit the number of possible variants, e.g. by restricting the combinations of values allowed for the different attributes of the product. The traditional method for modelling products is the type-instance approach: the model defines a number of *types*, that are then instantiated as *individuals* during the configuration process to store the final data.

Several high-level modelling languages tailored for product configuration have been proposed, including PCML [Tiihonen *et al.*, 2002]. Other languages such as UML have also been studied for modelling product configuration [Felfernig *et al.*, 2002; Hvam *et al.*, 2008]. Finally, product configuration has also been successful in industry [Haag, 1998].

2.2 Software Product Lines

Software product lines (SPL), also known as software product families, is a set of software systems sharing a common set of features that "satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements and Northrop, 2001].

Some approaches consider software product lines from an architectural point of view. *Architecture description languages (ADLs)* have been proposed to describe the SPLs in terms of their structure, including their components, interfaces, or communication protocols; but few can handle variability in SPLs. A few exceptions exist: Koalish [Asikainen *et al.*, 2003] for example extends Koala, an ADL based on components and interfaces, by adding variability elements such as attribute *domains* and *constraints*.

A more common method to model SPL is using features. Feature modelling approaches are based on the concept of *features*, that usually represent the visible characteristics of the system, from an end-user point of view. Well-known feature modelling methods, such as FODA (Feature Oriented Domain Analysis) [Kang *et al.*, 1990] or FORM (Feature Oriented Reuse Method) [Kang *et al.*, 1998] use a *feature model*, which represents a feature tree using different relations between features and subfeatures, including *mandatory*, *optional* or *alternative* relations. Feature models have been extended to support shared subfeatures, feature attributes and cardinalities or feature groups [Czarnecki *et al.*, 2005b].

Finally, Asikainen et al. [2007] recently proposed Kumbang. Kumbang combines advanced feature modelling concepts with the approach from Koalish, and adds support for advanced constraint relations compared to traditional feature modelling approaches.

2.3 Service Configuration

Configurable services represent services that can be customised from a set of pre-defined options, in order to fit the needs of individual customers. Research on configurable services and how to model them is a relatively recent topic: several authors have been discussing the configuration of different types of services, e.g. IT services [Böhmann *et al.*, 2003].

Other researchers [Akkermans *et al.*, 2004; Heiskala *et al.*, 2005] propose more detailed conceptualisations for modelling services. The most similar to our work, Heiskala et al. [2005], presents a conceptual model following a type/instance approach using four viewpoints, called *worlds*: the *needs world*, representing the customer's needs; the *service solutions world*, for the service's specifications; the *process world*, related to the service delivery; and finally the *object-of-services world*, that is used to describe the service recipient and the environment in which the service will be supplied.

3 Research Problem

In this section, we define the Research Problem motivating our work, and describe the example that illustrates our approach.

The Research Problem considered in this paper is: *how to uniformly support the configuration and management of heterogeneous product families?* What we refer to as a heterogeneous product family is a family of products integrating separate design disciplines interacting with each others. In the rest of the paper, we refer to these design disciplines as the *dimensions* of such a product family.

Modelling variability in heterogeneous product families yields various issues, due to the diversity of the product knowledge necessary to address these different dimensions. Indeed, the model of a heterogeneous product family can be very complex, and involves several types of users with different skills and objectives, making the need for uniformity of prime importance. The different dimensions in such a product family are rarely independent, and it is primordial to take the interactions between dimensions into account.

With this, we specify the main research problem in more detail with the following Research Questions:

RQ1 *What are the needs of the users of the model to be supported?*

RQ2 *What modelling constructs support addressing the heterogeneity?*

RQ3 *How to integrate together the different dimensions of heterogeneity in the models?*

RQ4 *How to support the management of such product family over time and for different market situations?*

As a motivation example, we present in Figure 1 a simplified version of a product family consisting of netbooks, smartphones and tablet computers. The example represents a configurable family of products consisting of: a set of physical elements (a motherboard with hardware chips, a screen, ...); the configurable software running on the devices (applications, libraries, ...); and the services associated with the devices (subscriptions, synchronisation services, ...).

This running example illustrates how complex the modelling of a heterogeneous product family can be. Indeed, the engineers responsible for modelling the variability in the physical parts of the system often possess a knowledge different from the ones responsible for managing the software configuration model, or creating the service model. As can be seen in the previous section, although the modelling approaches often use the same basis (types, partonomy, etc.), the high-level concepts behind each type of modelling are different, and thus require different mindsets. One can then assume that the task of managing the variability of the hardware, software and service parts for large products is delegated to separate groups of knowledge engineers.

Configuring such a product family can be quite complex, due to the amount of technical details represented in each different aspects of the products. Those details are often not very accessible to salespersons and end-customers, who prefer viewing the features (or functions) of the product families, as described in [Soininen *et al.*, 1998]. Defining the feature set of the product family may be enough in some cases.

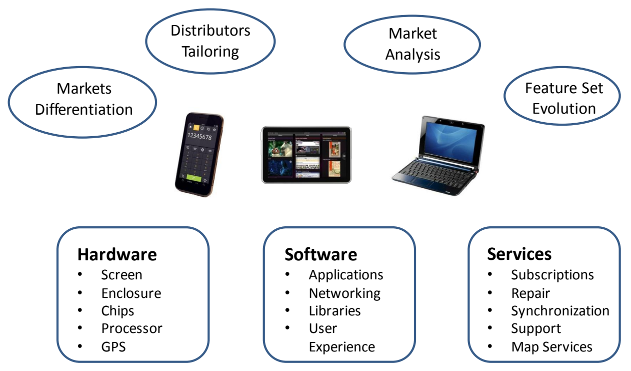

Figure 1: Running example. The product family represents mobile devices and is configured according to three dimensions, hardware, software and services, and need to be adapted to different scenarios, e.g. tailored to distributors and markets.

However, we identified several scenarios that illustrate different situations where this feature set may need refinement:

- *Market differentiation*: The company selling the products proposes different feature sets for different markets. In our example, different markets, e.g. Europe and United States, means different data signals to be handled by the phones, as well as different regulations. The possible combinations of features may just be restricted on those different markets.
- *Feature set evolution*: The product family's feature set is evolving with time. Devices may not arrive fully featured on the market, due to time constraints or strategic decisions. A refined feature set may be needed for a specific time, with additional constraints that may disappear (or be modified) in future evolution of the product family.
- *Distributors tailoring*: The producing company is distributing the products to different intermediary vendors. Products as our example may not be distributed directly by the manufacturer. This producer may propose a feature set to vendors that can adapt it in order to forbid specific combinations, or to create a more simple feature set for the end-customer. For example, the example products may be sold by distributors by letting the customer choose between different feature packages, limiting the choices in configuration.
- *Market analysis*: The final customers can also be considered first (instead of the product family). A market study identifies the different needs of the final customers (or needs that the company wants to introduce in the market) and build different feature sets to satisfy these needs, aiming at creating a product family to fit those. On the contrary to the first three scenarios, this scenario considers the market needs as the basis for designing the product family.

These scenarios provide a more concrete characterisation of how the functionalities of the product family may need to evolve depending on its use and distribution, as introduced in Research Question 4.

4 Modelling Framework

Our approach is based on the concept of *modelling views*. Those views are used to model different aspects of the product family, according to the different roles of the modellers. The main assumption is that each product family considered consists of different dimensions, and that all those parts need (and benefit from) configuration. Models are created and maintained by knowledge engineers from various informations given by domain experts. However, heterogeneous product families with multiple dimensions may require different kind of domain experts with different roles and sets of skills, according to the degree of technicality or the dimension considered.

In this section, we thus define three different types of views: the feature views, the structure views and the realisation views, depending on their intended audience and how they contribute to the model of the product family through different levels of abstraction. The views are characterised by a set of concepts with a specific organisation. Most of the concepts presented here are not new in themselves, but how they interact between each others within and between views is of importance.

4.1 Feature Views

Feature views provide a view of a product family from a high level of abstraction. These views are targeted at sales persons or end-customers that need to have an understanding of what the product individuals can do, instead of how they can do it. In our conceptual approach, feature views are not separated according to the different dimensions of the product family. The relations between the concepts described in feature views are related to the product individuals as a whole, and as such should not be dimension-specific. Product individuals can indeed be characterised by the features (or functions) they provide, independently from the way they are structured.

A feature view is composed of *feature types*, organise in partonomy (*subfeatures*) and taxonomy (*subtypes*) structures, as shown in the UML metamodel in Figure 2(a). Variability is defined in each feature type using *attributes* that can take different values. A feature subtype inherits all the properties of its supertype, i.e. its attributes, subfeatures, and constraints. Two types of constraints can be added to each type. *Compatibility constraints* model dependencies between the feature view, i.e. it specifies conditions that must hold in a valid configuration. *Implementation constraints* model the dependencies between different types of views, and will be detailed in Section 5.

Example: Consider our motivation scenario (Figure 3). Feature types such as *Input* or *Localisation* can be used to define the input type (touch input, keyboard features) or if GPS localisation should be available on the device.

4.2 Structure Views

Feature views are implemented by *structure views*, which define the different design components that realise the described features of the product family, and the relations between them. Structure-based approaches for configuration are widely used [Soininen *et al.*, 1998], as the compositional structure of the product families is often used to represent the product data knowledge. The structure views communicate the aspects of the architecture of interest to those involved in designing the

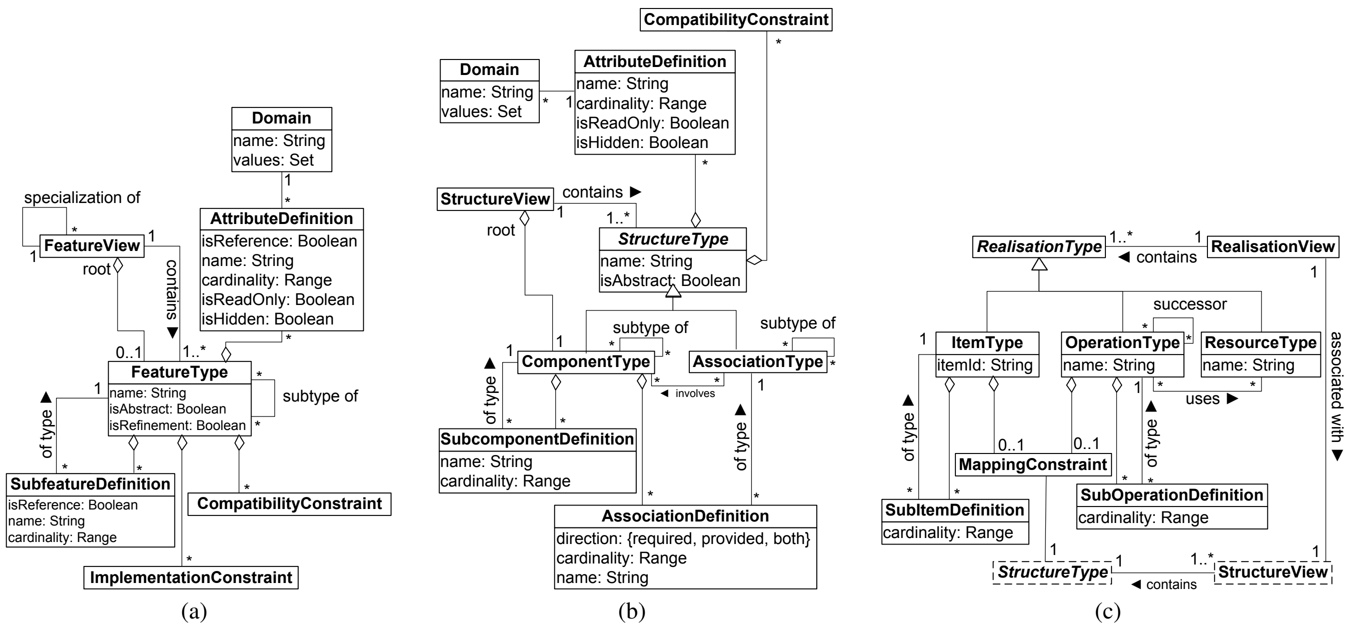

Figure 2: UML metamodels for: (a) Feature views (b) Structure views (c) Realisation views

system. They provide more concrete models of a product family, as it represents the specifications of the components of the system. Structure views are thus mainly aimed at design or maintenance and are for example targeted at product design engineers, software architects or service contractors.

A structure view is composed of *structure types*, that can be either *component types* or *association types*. As for feature views, structure views are organised in partonomy and taxonomy structures (Figure 2(b)). Types can have attributes and compatibility constraints as well.

The different concepts used in structure views have a specific meaning according to which dimension each view refers to. For example, a *physical structure view* represents the physical structure of the product family. Component types are entities whose individuals are physical components involved in the physical design, while association types are used to model non-directional physical links between two components. A *software structure view* describes the architecture of the software system involved in the product family. Instances of component types represent software components, and association can be defined to model interfaces, whether they provide software functions or require some. Also, a *service structure view* describes the specifications of the service to be delivered. Component types are service element types, and describes contractual agreements of what to be delivered, similar to what is modelled in the service solutions world of Heiskala et al. [2005].

Example: In our motivation scenario (Figure 3), the physical structure view contains a *Screen* and a *TouchScreen* component types with a *size* attribute, while the software view handles the *User Experience (UX)* framework and software interfaces to the *Middleware* libraries. The service structure view declares *RepairCoverage* or *PhoneSubscription* as types.

4.3 Realisation Views

Realisation views offer a detailed technical view of how the product individuals are realised. Compared to structure views, whose purpose is to represent the design of a specific dimension of the product family, realisation views are aimed at describing the elements necessary for the concrete realisation of the system for that dimension. They are thus targeted at highly specialised engineers, e.g. product engineers, software developers or service deliverers, and represent the lowest abstraction level in our conceptual modelling framework. Each realisation view is associated with a dimension, which defines its proper meaning: physical products use this view to represent manufacturing data, while software involve the solution deployment, and services the delivery process.

The building blocks of a realisation view are *realisation types*. There are three possible realisation types: *item types*, *operation types* and *resource types*. Item types represent the production components used to realise the products. It can be a BOM item for manufactured parts, a software package when dealing with software, or an object to be produced when delivering a service (e.g. a contract or a bill). Operation types are used to specify a set of operations needed during the production of individuals (e.g. manufacturing operations, software deployment, service processes). Resource types may describe a machine, an operator, an information or anything that may be necessary to complete the operations.

Contrary to structure views, realisation views are not starting with a single root type. Instead, each realisation view is associated with a structure view (from the same dimension), and each item or operation type may be associated to a relevant structure type via a *mapping constraint*. Types mapped to a structure type defines their own tree of *subitems*, providing a more detailed breakdown of the production components.

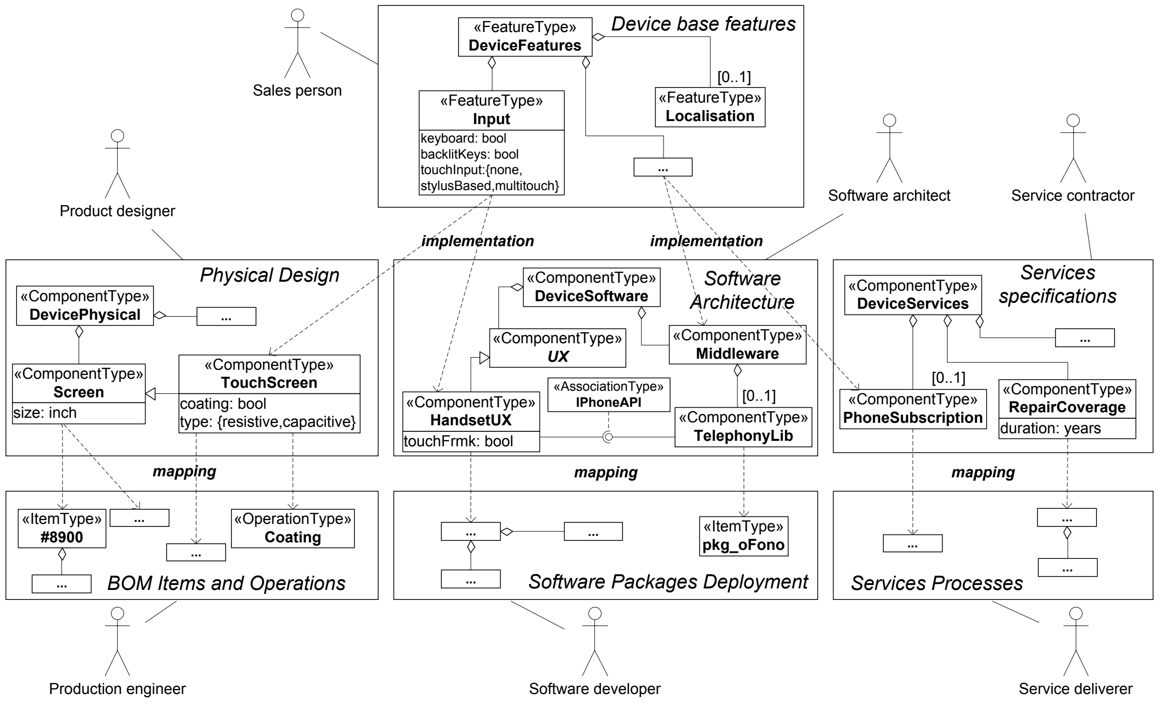

Figure 3: Overview of the motivation example model and the different modelling views, depending on the three dimensions (physical, software and services). Partonomy relations are shown using UML Aggregations, taxonomy relations using UML Generalizations. For the sake of brevity, only a subset of the types and the base feature view in the hierarchy are shown.

Example: The *Screen* component type in the physical structure view of our motivation example can be mapped to different manufacturing items. A specific mapping can be made if *TouchScreen* is chosen instead, or when a specific configuration is made (e.g. changing the value of the *size* attribute).

5 Dependencies and Constraints

Constraints can be used to specify dependencies within or between views when other modelling mechanisms are not sufficient to capture them. Constraints are written in a constraint language, and involve types attributes and predicates using pre-defined functions, such as *Count(...)* that returns the actual cardinality in a partonomy relation. The full description of the constraint language is out of the scope of this paper.

5.1 Compatibility Constraints

A compatibility constraint is specific to a particular view, and can only involve properties of this view. The evaluation of a constraint occurs during configuration, when types are instantiated to individuals. Each instance of the context type in which the constraint is declared must satisfy it.

Example: The following constraint, declared in the *Input* component type, specifies that the physical keyboard feature must be selected if one wants a backlit keyboard:

$$backlitKeys \Rightarrow keyboard$$

Constraints may also contain references to properties that are not always present in the product individual being configured, e.g. if a constraint accesses a subpart whose cardinality is not fixed, or an attribute from a subtype that may not be chosen (the property is said *inactive*). Each compatibility constraint containing at least one inactive term is evaluated to *true* during configuration.

5.2 Implementation Constraints

Implementation constraints are essential to our framework, as they model the interaction between the base feature view and the structure views (and in the feature views hierarchy, see Section 6). They are composed of a left-hand side expression *L* and a right-hand side expression *R*, related by an implication or an equivalence operator. The expression *L* represents the features to be implemented by the constraint, in a similar way as in the compatibility constraints. On the other hand, the expression *R* represents what is needed in the stucture view(s) to implement the features specified by *L*.

Example: Consider the following constraint in the *Input* feature type from the base feature view:

$$\begin{aligned}&touchInput = \text{“}multitouch\text{''} \Leftrightarrow\\ &\quad (Physical :: TouchScreen.type = \text{“}capacitive\text{''}\\ &\quad \wedge Software :: HandsetUX.touchFrmk = true)\end{aligned}$$

This implementation constraint specifies that a device has a multitouch input if *there exists* a capacitive touchscreen and a touch framework is implemented in the software. Existential quantifiers are implicitly used in the semantics of the expression *R*, as the feature may exist if there is at least one combination of structural elements implementing it. Universal quantifiers can also be explicitly used in some specific cases.

5.3 Mapping Constraints

Mapping constraints are defined in realisation views to specify under which conditions a realisation type should be included in the configuration results. There exists indeed a mapping between structure types and item and operation types, and the latters should only be part of the final configuration if certain conditions are met. Mapping constraints are declared in item and operation types, and refers to attributes from the structural type defined as context.

Example: The following mapping constraint is declared in the Coating operation type and takes as context the TouchScreen component type from the physical structure view:

$$c_{map}(TouchScreen, Coating) : oleophobicCoating = true$$

A valid configuration thus ensures that the latter constraint is true for each instance of the *TouchScreen* type, i.e. an instance of the *Coating* operation type is present for each instance of *TouchScreen* where the attribute *oleophobicCoating* is *true*.

6 Feature View Hierarchy

To address the management and evolution of the product family (Research Question 4) and the scenarios discussed in the Section 3, several feature views can be defined and organised in a *feature view hierarchy*. A model defines a *base feature view*, which will contain all the features available for the modelled product family, and should be implemented by the structural views. This base feature view may then be specialised, as different versions or evolutions of the product family may require special restrictions to the set of available features (*Market differentiation* and *Feature set evolution* scenarios), or even more abstract feature views in order to be presented to final customers (*Distributors tailoring* and *Market analysis*).

The feature view hierarchy defines a specialisation tree, rooted by the base feature view. A feature view $\mathcal{F}'$ is the child of another feature view $\mathcal{F}$ if $\mathcal{F}'$ is a *specialisation* of $\mathcal{F}$. This specialisation is done through different concepts:

- **Implementation**: A feature view $\mathcal{F}'$ can declare new feature types and attributes, for example to define more abstract feature groups and properties. As for the base feature view, the types in $\mathcal{F}'$ must use *implementation constraints* to associate their properties to the feature view $\mathcal{F}$, parent of $\mathcal{F}'$.
- **Refinement and reference**: Apart from defining new feature types, feature views can refine feature types from their parent view. A *refined feature type* can transform the original type by: defining new attributes or subfeatures; refining *referenced* attribute or subfeature definitions by restricting its cardinality, its domain or visibility (attribute) or change it to one of its subtypes (subfeature); changing the type from concrete to abstract to force the use of its subtypes; or by adding compatibility constraints to constrain the model even more.

Figure 4 shows the mechanism of feature types refinement. Type $F1$ is refined: the attribute $a1$ in $F1$ is declared as hidden, and a new attribute $a5$ is declared. The $feat3$ subfeature cardinality is also refined to $[1..2]$. Finally, even though $F1'$ is not directly modified, the type $F4$ is also refined: the domain of $a4$ is reduced and a new subfeature is defined.

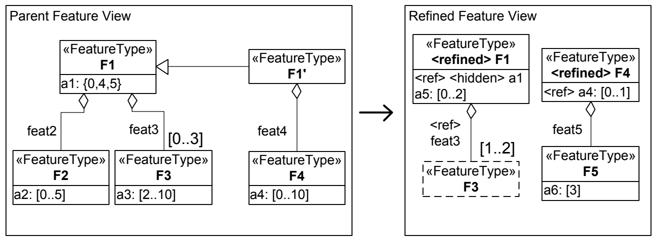

Figure 4: Feature types refinement. Refined types are characterised by the **<refined>** tag, while referenced definitions are tagged with **<ref>**. The type $F3$ in the Refined Feature View is shown with a dashed outline, as only the $feat3$ definition is part of the refined view, while the feature type is not and is just shown here for illustrative purpose.

7 Proof of Concept

A textual modelling language named ProCoLa has been defined to support our modelling framework. The language syntax is closely mapped to the different concepts described in this paper. A language service has been implemented in Visual Studio to support the ProCoLa language, providing an important number of features for tool support, such as syntax highlighting, syntax and semantic checks, automatic word completion among others. The language is supported by a C# compiler, and resembles that of an object-oriented programming language. A formalism of the framework and model analyses are currently being worked on in order to provide additional tool support, such as the ability to see model-wide dependencies of any change that may happen in a view, e.g. the deletion of an attribute or type.

The semantics behind our modelling approach (and ProCoLa) have been defined by implementing the translation of models to Dynamic CSPs (DCSPs) [Bartak and Surynek, 2005] and Conditional CSPs (CondCSPs) [Mittal and Falkenhainer, 1990] formalisms. DCSPs are used to handle the dynamic addition and removal of value assignments to attributes during interactive configuration, while CondCSPs are used to handle the notion of activity involved when dealing with dynamic cardinality or taxonomy structures for example, or the existential (or universal) quantifiers implied by implementation constraints. During the configuration process, an end-user first chooses which feature view he wants to use (if multiple feature views exist in the hierarchy), and then can enter his requirements through a user-interface by assigning values to attributes, or connecting associations. A single CSP model is usually used for all views, allowing a full propagation of the choices to the other views. However, the user may also consider configuring only a single view (using only compatibility constraints). More details can be found in [Quéva, 2011].

A larger mobile device product family based on the motivation example presented in this paper has been modelled using our conceptual framework and ProCoLa. The model is split into 13 views, including three realisation views and three structure views (one for each dimension), and a feature views hierarchy of 7 views. It contains around 250 types, 200 attributes and over 300 constraints. During the modelling of the product family, ProCoLa has provided a sufficient level of support to capture the different part of the family and their dependencies, in a reasonable amount of time. The translation into the CSP formalisms is very fast, while the consistency checks at runtime are done within a few seconds at most.

8 Discussion and Comparison with Related Work

The different views provides a modelling framework as a contribution to address the Research Questions (RQs) exposed in Section 3. The clear separation of concerns in the structural and realisation data for each dimension is motivated by RQ1 (*What are the needs of the users of the model to be supported?*) and previous work on modelling each dimension (Section 2). Each view is targeted at a different audience: the structural model of the software is handled by a software architect, while a production engineer may be more adequate to handle bill-of-materials and manufacturing operations. Moreover, we argue that structural and realisation views from each dimension should be considered independently from each others, and unified in the feature models they contribute to implement, defined in feature views. In Figure 3, the sales persons working on the device features model the types of input that the end-user may be interested in. How this feature is implemented is dependent on several structural elements from different parts of the system: the touch screen hardware and a touch framework component in the user experience software. Those two elements can however be chosen independently from each others, but will only provide the feature if they are both present in the final product.

The UML metamodels (Figure 2) provide a good basis in order to address the problem of modelling the different dimensions of an heterogeneous product family, as raised in RQ2 (*What modelling constructs support addressing the heterogeneity?*). Uniform modelling constructs and the different types of inter-views constraints defined in the framework also contribute to the issue posed in RQ3 (*How to integrate together the different dimensions of heterogeneity in the models?*): the implementation and mapping constraints permit to model the interdependencies between the views, allowing a tight integration of the different dimensions of the product family. Modelling these constraints requires communication between the different stakeholders. The sales person responsible for the touch input feature inquires the product designer in order to assess what hardware components are needed for the requested feature. On the other hand, product designers and production engineers need to confer on which items are available to realise the structural design of the hardware.

Our modelling approach also extends the concept of feature model to a feature view hierarchy, as a contribution to RQ4 (*How to support the management of such product family over time and for different market situations?*). The refinement of feature type's attributes can be used to model scenarios such as *Market differentiation* (by adding constraints for specific markets), *Feature set evolution* (by creating multiple feature views depending on the current capabilities of the product) or *Distributors tailoring* (by allowing them to create their own specialised views). In a *Market analysis* scenario, several specific feature views are created in order to match the product feature sets to introduce in the market. These views may then be joined into one base feature view, by gathering common elements or creating more abstract features that can be specialised to fit the original views, via refinement or implementation. The feature view hierarchy thus enables a unification of the product family management and evolution at the feature level, independently from the heterogeneity of the family, while each dimension may have its own separate mechanism for coping with this issue (e.g. product data management, ...).

Modelling concepts from our approach are based on previous work, mainly in product configuration [Soininen *et al.*, 1998]. The four worlds from Heiskala et al. [2005] can also be compared with the modelling views of our framework: the *needs world* concerns the customer's needs (in an abstract way), and is thus close to our feature views, which describes the abstract features that the customer may require; the *service solution world* denotes the set of elements used to establish the service's specifications, as the structure views; the *process world* describes how the service will be delivered, or realised, as in our realisation views. Note that there is nothing in our conceptual approach that is similar to the *object-of-services world* from [Heiskala *et al.*, 2005], which specifies the service recipients or the environment relevant to those recipient. From a modelling point of view, all these worlds are based on the same metamodel, using different types and attributes, as well as taxonomy and partonomy structures, as in our approach. However, dependencies between types of different worlds are simply modelled using classical constraints, while we use implementation and mapping constraints. Also, our framework is centered on the configured product, and thus the services described in the services dimension are seen from the configured product's point of view, while the external environment is not considered. Another type of view may thus be necessary in our framework to define externally controlled elements (such as, in the running example, access to company specific services or credentials, data transfer from an old device, etc.).

Feature modelling approaches such as cardinality-based models [Czarnecki *et al.*, 2005a] also have similarities to our feature views, although the richness of constraints and the partonomy/taxonomy structures used in our models is somewhat more complex than with the classic feature-oriented relations. Multi-view models in feature modelling have also been studied. Czarnecki et al. [2006] sketches a model where different levels of customisation are modelled (including feature and design view). Reiser and Weber [2006] and work from Zaid et al. [2010] propose feature models with different perspectives, although they are all centered on software variability and feature modelling techniques only, and the lack of specialisation hierarchy may make the task of implementing the unification with different structured views difficult.

Kumbang [Asikainen *et al.*, 2007] is the closest to our work on the software variability side, including their type-instance approach. We consider our work to be an extension of Kumbang, as we use implementation constraints to unify structure views from the different dimensions (including manufactured products and services), as well as we model realisation data. Thus the main contribution of our work is to provide conceptual and practical mechanisms to bring the different dimensions together and unify them under feature models.

9 Conclusion

In this paper, we present an approach to help with the issue of modelling a product family consisting of different design disciplines (or dimensions). The presented framework has been motivated by a four research questions and illustrated by several scenarios.

Our framework is based on modelling views and synthesizes the concepts from different approaches from product configuration, software variability and service configuration, and unify them around feature views using implementation mechanisms. We also describe a feature view hierarchy and refinement mechanisms to cope with the evolution and adaptation of the product family, which remains an important issue [Krebs, 2008].

The approach has been motivated by the use case of a mobile devices product family, and has been implemented in a language prototype as a proof of concept. However, we have yet to perform an in-depth case study with industrial data in order to test the feasibility of implementing a real-life product family with our framework, as well as completing the formalism and tool support for the language, which is planned as future work.

References

[Akkermans *et al.*, 2004] H. Akkermans, Z. Baida, J. Gordijn, N. Peña, A. Altuna, and I. Laresgoiti. Value webs: Using ontologies to bundle real-world services. *IEEE Intelligent Systems*, 19(4):57–66, 2004.

[Asikainen *et al.*, 2003] Timo Asikainen, Timo Soininen, and Tomi Mannisto. A koala-based approach for modelling and deploying configurable software product families. *Proc. International Workshop on Product Family Engineering*, pages 225–249, Jan 2003.

[Asikainen *et al.*, 2007] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.

[Bartak and Surynek, 2005] R. Bartak and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. *Proc. FLAIRS'05*, pages 161–166, 2005.

[Böhmann *et al.*, 2003] T. Böhmann, M. Junginger, and H. Kremar. Modular service architectures: a concept and method for engineering it services. In *Proc. International Conference on System Sciences*, 2003.

[Clements and Northrop, 2001] P. Clements and L. Northrop. *Software Product Lines — Practices and Patterns*. Addison-Wesley, 2001.

[Czarnecki *et al.*, 2005a] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practices*, 10(1):7–29, 2005.

[Czarnecki *et al.*, 2005b] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practices*, 10(2):143–169, 2005.

[Czarnecki *et al.*, 2006] K. Czarnecki, M. Antkiewicz, Chang Hwan, and Peter Kim. Multi-level customization in application engineering. *Communications of the ACM - Software product line*, 49(12):61–65, 2006.

[Felfernig *et al.*, 2002] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Configuration knowledge representation using uml/ocl. *Lecture notes in computer science*, pages 49–62, Jan 2002.

[Haag, 1998] A. Haag. Sales configuration in business processes. *IEEE Intelligent Systems*, 13(4):78–85, 1998.

[Heiskala *et al.*, 2005] M. Heiskala, J. Tiihonen, and T. Soininen. A conceptual model for configurable services. In *IJCAI Workshop on Configuration*, Scotland, 2005.

[Hvam *et al.*, 2008] L. Hvam, N. H. Mortensen, and J. Riis. *Product customization*, volume XII. Springer, 2008.

[Kang *et al.*, 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S .A. Peterson. Feature-oriented domain analysis (foda) — feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Kang *et al.*, 1998] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[Krebs, 2008] Thorsten Krebs. Knowledge management for evolving products. In Max Bramer, Frans Coenen, and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXIV*, pages 307–320. 2008.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. 1990.

[Pine, 1993] B. J. Pine. *Mass Customization - The New Frontier in Business Competition*. Harvard Business School Press, 1993.

[Quéva, 2011] Matthieu Quéva. *A Framework for Constraint-Programming based Configuration*. PhD thesis, DTU Informatics, 2011. Submitted in May 2011.

[Reiser and Weber, 2006] Mark-Oliver Reiser and Matthias Weber. Managing highly-complex product families with multi-level feature trees. In *Proc. of the 14th IEEE International Requirements Engineering Conference*, 2006.

[Sabin and Weigel, 1998] Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13:42–49, July 1998.

[Soininen *et al.*, 1998] T Soininen, J Tiihonen, T Männistö, and R Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[Tiihonen *et al.*, 2002] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen. Empirical testing of a weight constraint rule based configurator. In *Workshop on Configuration, ECAI*, pages 17–22, 2002.

[Zaid *et al.*, 2010] L.A. Zaid, F. Kleinermann, and O. De Troyer. Feature assembly: a new feature modeling technique. In J. Parsons et al., editor, *ER 2010, LNCS*, volume 6412, pages 233–246, 2010.