Timo Asikainen, Tomi Männistö, and Timo Soininen. 2006. A unified conceptual foundation for feature modelling. In: Liam O'Brien (editor). Proceedings of the 10th International Software Product Line Conference (SPLC 2006). Baltimore, Maryland, USA. 21-24 August 2006. IEEE Computer Society, pages 31-40.

A Unified Conceptual Foundation for Feature Modelling

Timo Asikainen, Tomi Männistö, and Timo Soininen
Helsinki University of Technology, Software Business and Engineering Institute (SoberIT)
{timo.asikainen,tomi.mannisto,timo.soininen}@tkk.fi

Abstract

Feature modelling has become perhaps the most popular method for representing variabilities and commonalities in software product families. A large number of feature modelling methods and supporting tools have been reported. The conceptual foundation of feature models remains vague, a fact that severely undermines the usability of feature models. Therefore, we introduce Forfamel, a rigorous conceptual foundation for feature modelling. Forfamel synthesises existing feature modelling methods in the sense that it covers most concepts and constructs found in existing feature modelling methods. In addition, Forfamel includes a few additional constructs that may prove to be useful. We show the novel contribution of Forfamel by reflecting it against previous feature modelling methods and arguing for its underlying design decisions.

1. Introduction

Feature modelling is a method for describing the commonalities and variabilities within a family of systems, such as a software product family. Since its introduction in 1990 [1], feature modelling has attracted significant research interest and it has been applied in a number of application domains. In addition, a large number of tools supporting the feature modelling paradigm have been introduced; see [2] for references concerning both application domains and tool support. However, feature modelling still has not made its break-through into the toolbox of every software architect or requirements engineer.

In our opinion, a factor hindering the propagation of feature modelling is the fact that most feature modelling methods lack a proper conceptual foundation: either the concepts and their interrelations used in a feature modelling method are not defined at all, or in an unsatisfactory manner. In addition, there are significant differences in what concepts are included in each feature modelling method, especially in their semantics.

In this paper, we study the conceptual foundation of feature models. We reify the foundation into a domain ontology called *Forfamel*. Forfamel synthesises existing feature modelling methods in the sense that it covers most concepts and constructs found in existing feature modelling methods. It includes a number of extensions to previous feature modelling methods. Forfamel enables the easy reuse of feature knowledge both within a feature model and between feature models. The modelling concepts are demonstrated using a running example. The semantics of Forfamel is rigorously defined, and the reasons underlying its design are thoroughly motivated by reflecting Forfamel against previous feature modelling methods.

We believe that the rich and well-defined concepts of Forfamel provide an excellent basis for developing tools supporting the creation and management of feature models, and configuring them to yield descriptions of individual systems in the family. Such tools are essential for feature modelling to become a technique truly popular in the software industry. As a proof of this argument, we provide an overview of Kumbang Configurator, a tool enabling the configuration of feature models based on Forfamel.

The remainder of this paper is structured as follows. In Section 2, we reiterate the basic feature modelling concepts. Forfamel is introduced in Section 3. Section 4 discusses a language and a prototype tool supporting Forfamel. Discussion and comparison to previous work follows in Section 5. Conclusions and an outline for further work round up the paper in Section 6.

2. Feature modelling

In this section, we briefly reiterate feature modelling. We begin by discussing FODA [1], the first feature modelling methods reported, thereafter discuss the most important extensions to FODA, and finally provide a brief overview of the research themes related to feature modelling.

In FODA (Feature Oriented Domain Analysis), the notion of feature is limited to attributes of systems that directly affect end users [1]. However, the definition of feature has been extended to "a system property that is

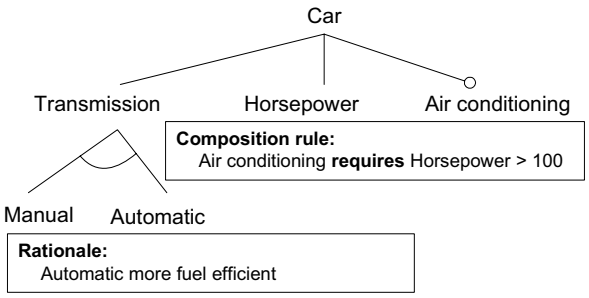

Figure 1 A sample FODA feature model [1]

relevant to some stakeholder" [3]. In this paper, we adopt the latter view of features: we do not assume that features would necessarily affect end users.

In FODA, features are organised in *feature models*. Figure 1 illustrates a sample feature model in FODA. A feature model is a tree where the root is a feature, sometimes referred to as the *concept*. The root feature has features as its subfeatures, and these may in turn have other features as their subfeatures, *et cetera*. A feature model is a description of a system family, e.g., a software product family. A description of an individual system is derived by selecting a subset of features.

There are a number of subfeature kinds: *mandatory* subfeatures must be selected whenever its parent is selected; an *optional* feature may be selected whenever its parent is selected, but needs not be selected; an *alternative* subfeature consists of a set of alternatives of which exactly one must be selected whenever the parent feature is selected.

An individual product of the system family is described by a selection of features that obeys the rules of the feature model. Such a collection of selected features is termed a *feature configuration*. The task of finding a feature configuration matching a specific set of requirements for a product individual at hand is termed *configuration task*.

A number of feature modelling methods that extend FODA have been suggested [3,4,5]. Most popular extensions include feature cardinalities, attributes, and or-features, or more generally, group cardinalities. Some methods explicitly allow feature models to be directed acyclic graphs, a generalisation of trees.

A *feature cardinality* [5] specifies how many times a subfeature must be selected into a feature description. A mandatory feature corresponds to cardinality 1, and an optional subfeature to cardinality 0..1. Other cardinalities can be defined, such as 2..4 (two to four subfeatures must be selected), and 1..* (at least one subfeature must be selected).

In some approaches, features may have attributes [3,5]. An attribute is a value that characterises a feature. There are different variants of attributes: in [5] a feature may be defined multiple attributes, whereas in [3] a feature may have at most one attribute.

An *or*-feature [4] is a subfeature kind similar to an alternative feature, with the difference that at least one of the alternatives must be selected. *Group cardinalities* [2] are a generalisation of the alternative and or-feature concepts, similarly as feature cardinalities generalise mandatory and optional features: a group cardinality specifies the number of subfeatures that must be selected from a group.

In addition to conceptual extensions, a large number of other papers related to feature modelling have been published. Carrying out the configuration task in multiple stages has been studied [3]. Feature models have been used to support a variety of different engineering techniques, such as product line production planning [6], re-engineering legacy systems [7], and other purposes [8,9,10]. Metrics have been defined for feature models [11]. Feature models have been defined semantics using a range of knowledge representation languages, such as propositional logic [12,13,14], constraint programming [15], and grammars [2,13].

3. Forfamel — a conceptual foundation for feature modelling

In this section, we define Forfamel, a domain ontology for feature modelling. First, we define the notion of feature configuration. A feature configuration is a description of the features delivered by an individual system. Thereafter, we discuss the notion of feature models and the conditions a feature configuration must fulfil to be a valid configuration of a feature model.

3.1. Feature configuration

The concept of feature, as defined above, is associated with a system; there is no reference to a software product or other system family in the definition. Therefore, we begin our exploration of features by describing how a feature configuration characterises a system through its features.

A *feature configuration*, or *configuration* for short, is a description of the features delivered by a system; see Figure 2 (a) for a UML metamodel of concepts related to feature configuration.

A feature configuration consists of a *set of features (F)*, the *subfeature relation (s)*, the *attribute relation (a)*, the type function *t*, and the *root feature* (*r*) that is a member of the set of features.

A feature is characterised by its *type*, *subfeatures*, and *attributes*.

The essence of a feature is captured by its type. Two features that do not agree in their types cannot be

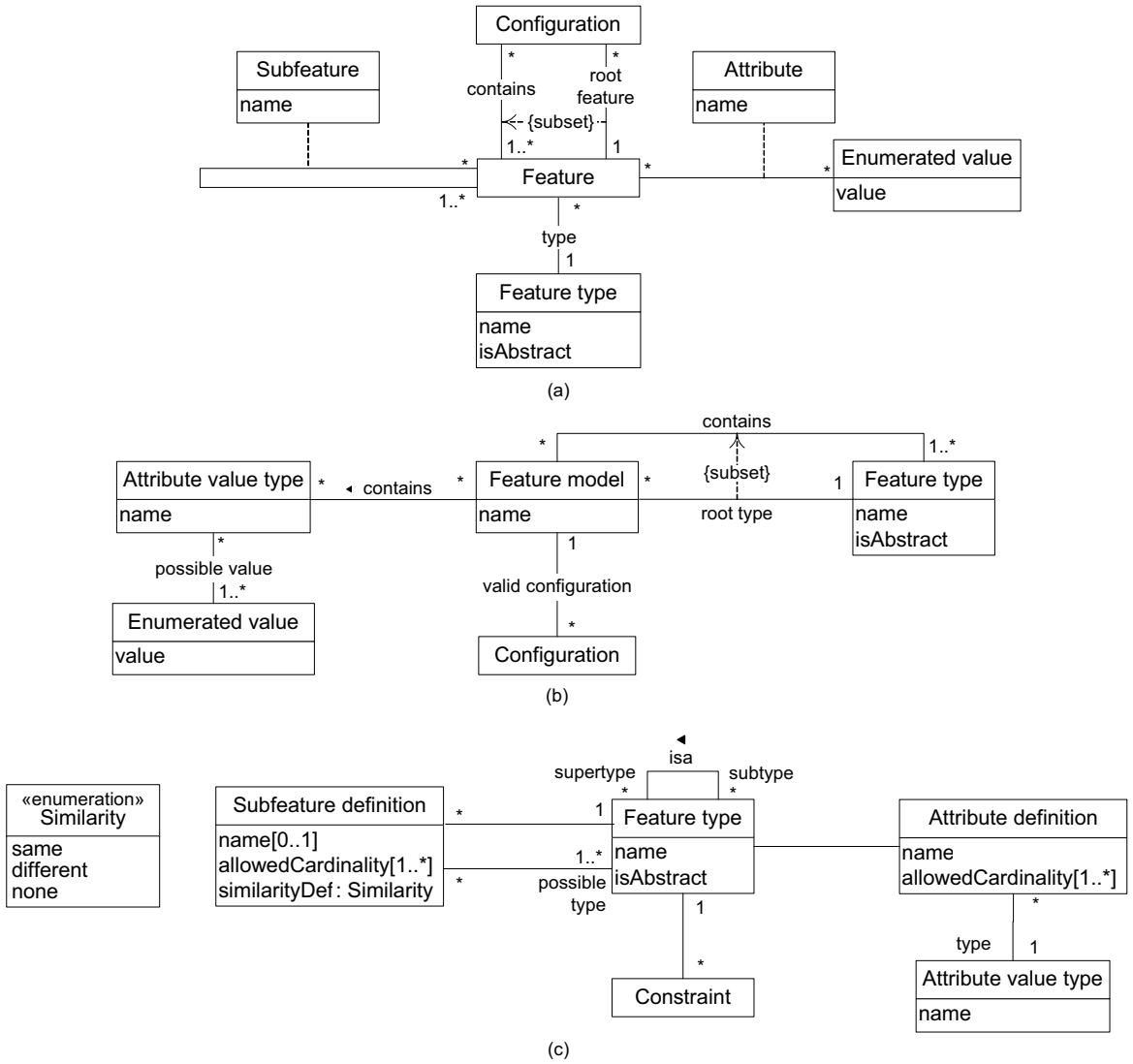

Figure 2 Forfamel metamodel (a) Feature configuration (b) Feature model (c) Feature type

considered to be equal. Two features that do agree in their types may still be distinguished by their subfeatures and attributes. The types of features are represented by the type function t: $F \rightarrow T$, where T is the set of types appearing in the configuration.

Example. Figure 3 (a) contains a sample feature configuration of a text editor; the sample configuration, along with a corresponding feature model, will be used as a running example throughout this paper.

The notation used for features resembles the UML notation for instance specifications: each feature is represented by a box, and the type of the feature is preceded by a colon and underlined. The root feature (of type *Text editor*) is identified by the text '«root feature»', resembling the UML stereotype notation. ■

The subfeatures of a feature are represented by the ternary relation s (short for *subfeature*): the triple (w, p, n) being in s has the semantics that feature w (whole) has feature p (part) as its subfeature in the role identified by n (name). The name may be omitted. We also say that p is a *subfeature* of w, and that w is a *parent* of p.

Example. The running example contains a number of subfeatures. For instance, the root feature has a subfeature of type *Clipboard*, with no explicit role. The root feature has features typed *MathPal* and *EqMaster* in roles *primary* and *secondary* equation editor, respectively. The root feature has a subfeature typed *Spell checking*, that in turn has two features, typed *English* and *Dutch*, as the spell checking languages. ■

The attributes of features are represented by the a (attribute) relation: the triple (f, v, n) being in the a relation has the semantics that feature f has the value v as its attribute with name n.

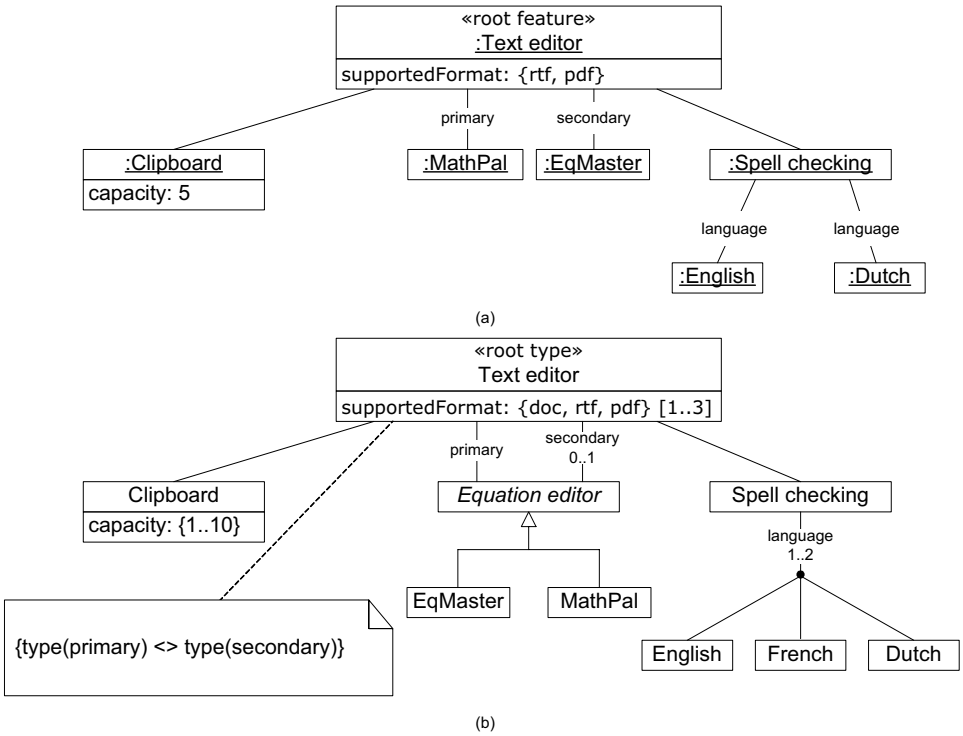

Figure 3 (a) A sample feature configuration (b) A sample feature model

Example. The root feature has two attributes with name *supportedFormat*: *rtf*, and *pdf*. The two attributes denote the fact that the text editor described by the feature configuration supports documents both in RTF and PDF formats. The feature typed *Clipboard* has attribute named *capacity* value 5, implying a clipboard able to hold five items at a time. ■

There are a number of restrictions on the subfeature relation. First, the root feature must not be a part of any other feature. Second, all features, except the root feature, must be subfeatures of some other feature. In other words, the feature configuration must have a single root. Third, the transitive closure of the binary projection

$$s_2 = \{ (w, p) : (w, p, n) \in s \}$$

of s must be anti-symmetric and anti-reflexive: a feature must not be a transitive subfeature of itself, and it must not be true for any two features that a is a subfeature of b, and b is a transitive subfeature of a.

We do not require that no feature is a subfeature of more than one feature. That is, we allow feature diagrams that are not tree-formed but directed acyclic graphs (DAG).

3.2. Feature model

In this subsection, we describe feature models and show their relation to feature configurations. In more detail, we describe the constituent elements of feature models and define when a feature configuration is a valid configuration of a feature model.

A *feature model* is a characterisation of a system family, such as a software product family. A feature model intentionally defines which feature configurations are valid descriptions of systems in the software product family. We will call such a feature configuration a *valid configuration* of the model. Below, we will detail what is required of a feature configuration to be valid.

A feature model consists of a set of feature types Φ, a set of attribute value types A, and a root feature type ρ; see Figure 2 (b) for a UML-metamodel of the concepts related to feature models.

A feature type is a description of its instances, i.e., features. We will discuss how feature types characterise their instances in more detail below.

Feature types are organised into taxonomies defined by the binary *isa*-relation. For types t_1 and t_2, $isa(t_1, t_2)$ implies that t_1 is a *direct subtype* of t_2 and t_2 is a *direct supertype* of t_1. Further, that t_1 is a *subtype* of t_2 is implied by the pair (t_1, t_2) being in the transitive closure of *isa*; this also implies that t_2 is a *supertype* of t_1.

Each feature is a *direct instance* of exactly one feature type that is the type given by the t relation of a feature configuration. In addition, a feature is an (indirect) instance of all the supertypes of t.

A feature type is either *abstract* or *concrete*. The intuitive semantics is that only concrete feature types may have direct instances in a valid configuration.

Example. Figure 3 (b) illustrates a feature model. Feature type *Equation editor* is an abstract feature type. It has two subtypes, *EqMaster* and *MathPal*, both of which are concrete. ■

A feature model must specify a *root feature type* that is one of the feature types in the model

A feature type, see Figure 2 (c), consists of a *set subfeature definitions*, a *set of attribute definitions*, and a *set of constraints*. A feature type inherits these constituent elements from all of its supertypes.

A subfeature definition consists of an optional *subfeature name* (for short, *name*), a *set of possible subfeature types* (*types*), a *similarity definition*, and a non-empty, finite *set of allowed cardinalities* (*card*). The similarity definition takes one of three values: *same*, *different*, and *none*.

Intuitively, the subfeature definitions of a feature type specify what subfeatures the features of the type must and may have. The semantics of a subfeature definition are captured by the following definition. We will use the dot (.) notation to refer to the properties of subfeature definitions, and later attribute definitions. For instance, *d.card* refers to the set of allowed cardinalities of subfeature definition *d*.

Definition (Subfeature definition conformance). Given a feature type t and d a part definition of t, and feature f of type t, let $C = \{ c : (f, c, d.name) \in s\}$, i.e., C is the set of features that are subfeatures of f by the name $d.name$.

Feature *f conforms to subfeature definition d* if (and only if) $|C| \in d.card$ and every member of C is a valid instance of a member of *d.types*; and, if the similarity definition has value *same*, all members of C must be direct features of the same type, and if the value is *different*, all the members of C must be of different direct type. ■

Intuitively, it is required that the number of features must agree with the cardinality definition, be of a type contained in the set of possible subfeature types, and additionally adhere to the similarity definition.

Example. The root feature type is *Text editor*. It contains four subfeature definitions: unnamed definitions with *Clipboard* and *Spell checking* as their sole possible subfeature types, and definitions named *primary* and *secondary* with *Equation editor* as their sole possible subfeature type. All these definitions but *secondary* are mandatory subfeatures: the set of allowed cardinalities is 1 (default value, not shown in figure). The definition named *secondary* has the allowed set of cardinalities of 0..1, hence the subfeature is optional.

Feature type *Spell checking* defines a subfeature by the name *language*. The cardinality is 1..2, implying that at least one and at most two languages from *English*, *French*, and *Dutch*, must be selected. The similarity definition is *different* (the default value, not shown), implying that each possible type may be selected at most once. ■

An attribute definition consists of an *attribute name* (*name*), an *attribute value type* (*type*), and a non-empty *set of allowed cardinalities* (*card*). An attribute value type is a finite set of values, such as integers or strings.

Intuitively, the attribute definitions of a feature type specify which attributes the features of the type may and must have.

Definition (Attribute definition conformance). Given a feature type t, an attribute definition d of t, and feature f of type t, let $V = \{ v : (f, v, d.name) \in a\}$, i.e., the set of values that are attributes of f by the name $d.name$.

Feature f conforms to attribute definition d if $|V| \in d.card$ and every member of V is a member of $d.type$. ■

The intuition behind attribute definition conformance is similar to that behind subfeature definitions: the cardinality must match, and the values that are attributes of the instance must be in the attribute value type of the definition.

Example. The sample feature model contains two attribute definitions: feature type *Clipboard* defines an attribute named *capacity* with cardinality 1 of type integer in the range 1 to 10. In addition, the root feature type defines an attribute named *supportedFormat* and value type consisting of values *rtf*, *pdf*, and *doc.* ■

A *constraint* is a Boolean condition that can be evaluated in the context of a feature, that is, the feature, its attributes, transitive subfeatures and their attributes. Constraints are expressed using a constraint language. Specifying a constraint language, along with its semantics, is out of the scope of this paper; in practice, a language resembling the Object Constraint Language (OCL) [16] would likely satisfy the requirements for a constraint language.

Example. The running example contains one constraint, represented using the UML constraint notation. The intuition of the constraint is that the primary and secondary equation editors must be of different types: it would not make sense for a text editor to have the same equation editor, say, *MathPal*, both as the primary and secondary editor. ■

At this point, we are ready to define what is required of a valid instance of a type.

Definition (Valid instance). Feature *f* is a valid instance of feature type *t* if:

1) *f* is an instance of *t*;
2) *f* conforms to each subfeature definition of *t* and has no other subfeatures;
3) *f* conforms to each attribute definition of *t* and has no other attributes; and
4) each constraint of *t* evaluates to *true* for *f*. ■

The intuition behind the above definition is as follows. An instance of a type must conform to the subfeature and attribute definitions and constraints of the type. There may be not other subfeatures or attributes than those defined for the type.

Definition (Valid configuration). Feature configuration *c* is a valid configuration of feature model *M* if the root feature *c.r* is a valid instance of the root feature type $M.\rho$ and each feature in the configuration is a direct instance of a concrete feature type. ■

Intuitively, the root feature must be an instance of the root feature type. The definition also requires that all the features in the configuration be of concrete feature types. This captures the intuition between abstract and concrete features types: only concrete feature types may have instances in a valid configuration.

Example. All the instances in Figure 3 (a) are valid instances of their respective direct types that are all concrete. Hence the configuration is a valid configuration of the feature model in Figure 3 (b). ■

We require that in a well-formed feature model the transitive closure of the isa relation is asymmetric: no feature type is a supertype of itself. We do not restrict the number of direct supertypes a type may have: feature types may inherit properties from multiple types.

It is required that the compositional structure of feature types is such that a in a valid configuration, a feature instance may not contain a transitive subfeature that is an instance of the type of containing feature.

4. Validation

In this section we describe how we have validated the feasibility of Forfamel.

We have provided Forfamel with formal semantics by defining a translation from *Kumbang* to Weight Constraint Rule Language (WCRL) [17], a general-purpose knowledge representation language. Kumbang is a domain ontology and a modelling language we have developed. It includes Forfamel as a subset and additionally contains concepts and constructs for describing software architecture in terms of components, interfaces, *etc*.

Although general-purpose, WCRL has been designed to allow the easy representation of configuration knowledge about non-software products and shown to suit this purpose [18]. This suggests that WCRL is a reasonable choice for the knowledge representation formalism of our approach as well. Further, an inference system *smodels*[1] operating on WCRL has been shown to have a very competitive performance compared to other problem solvers, especially in the case of problems including structure [17].

We have defined a language that enables expressing feature models using Forfamel. The language has been defined machine-readable syntax using javaCC (Java Compiler Constructor)[2], a tool that generates Java code for both lexical and syntax analysis based on a grammar description. Further, we have implemented a piece of software that translates a Forfamel model expressed in the language into WCRL, thus providing Forfamel with formal semantics. A configuration tool supporting Forfamel has been implemented in our research group. The tool is called Kumbang Configurator [19]. Kumbang Configurator supports a user in the configuration task as follows. The tool reads in a Kumbang model and represents the model in a graphical user interface, see Figure 4. The user can enter her requirements for the individual product by resolving the variation points in the model: e.g., the user may decide whether to include an optional element in the configuration or not, or to select attribute values or the type of a of a given part. After each requirement entered by the user, the tool checks the consistency of the configuration, i.e., are the requirements entered so far mutually compatible, and deduces the consequences of the requirements entered so far; the consequences are reflected in the user interface. The consistency checks and deductions are done using *smodels* and the WCRL program translated from the model.

Once all the variation points have been resolved and a valid configuration thus found, the tool is able to export the configuration, which can be entered as an input for tools used to implement the software, or used for other purposes. Further details about Kumbang Configurator can be found in [19].

A number of software product families have been modelled using Kumbang. First, a model based on a family of car periphery systems by Robert Bosch GmbH has been constructed using Kumbang; see [20] for the original model of the family. This example includes both a feature and an architectural point of view. The total number of types (feature and component) in the example is about 30. In addition, a weather prediction network loosely based on a real product has been modelled. The model contains about 20 types.

---

[1] See `http://www.tcs.hut.fi/Software/smodels/`

[2] See `https://javacc.dev.java.net/`

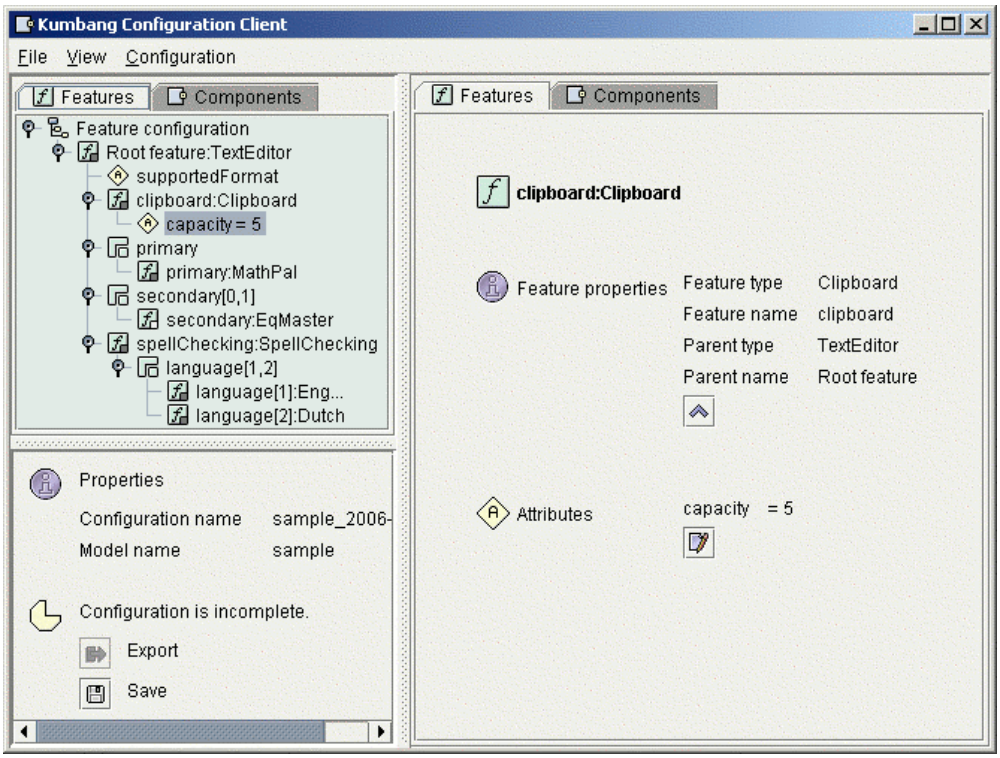

Figure 4 A screenshot from Kumbang Configurator

In both cases, Kumbang has provided a sufficient level of support to capture the intent of the product families. Specifically, Forfamel has provided an adequate level of support for modelling the feature aspects of the product families. The cognitive effort required to create the models has been reasonable. Translating the models into the formal representation in WCRL could be done within a couple of seconds. When configuring the models using Kumbang configurator, the time required for the reasoning tasks has been unnoticeable and the results have been in accordance with the semantics described in previous section.

5. Comparison to previous work

In this section, we reflect our work against previous work on feature modelling. As this paper studies feature and feature modelling from a conceptual point of view, the comparison will likewise be mostly made at the conceptual level: comparison to papers focusing on applications of feature models and defining semantics for feature models are only discussed when they extend the conceptual basis of feature models.

5.1. Models and configurations

Unlike most previous papers on feature modelling, we have emphasised the distinction between a feature model, i.e., a description of a system family, and a feature configuration, i.e., a description of an individual system. We have rigorously defined the notions of a feature configuration, a feature model, and valid configuration. The relationship between a feature model and its configurations is the very reason for pursuing feature modelling. Therefore, we consider this to be a major contribution of this paper.

Czarnecki *et al.* have introduced the notion of *staged configuration* [3]. The idea underlying the notion is that configuring a feature model is not an atomic action, but involves different stakeholder groups during multiple stages. Within each stage, a feature model, or diagram, as Czarnecki *et al.* call them, is specialised in such a way that the set of configurations of the specialised model is a subset of the original model. A fully specialised feature diagram denotes only one configuration. It may even be the case that the artefact of interest is not a fully specialised feature model, or a configuration, but a feature model still containing variability.

Arguably, the notion of staged configuration undermines the distinction between a feature model and its valid configuration. However, both the definition of a specialisation step and fully specialised configuration given above are dependent on the notion of a feature model and its configuration. Hence, it is still necessary to specify what are the valid configurations of a feature model.

0-7695-2599-7/06 $20.00

5.2. Types and instances

Traditionally, feature modelling methods have made no clear distinction between feature types and their instances, features [1,4,13]. That is, both entities appearing in a feature model and in a feature configuration, i.e., a description of an individual system, have been termed features. However, we believe there are several reasons for distinguishing 'features' appearing in feature models and configurations.

First, entities appearing in feature models include variability, e.g., alternative and optional subfeatures. Entities appearing in configurations are free of such variabilities. This is a fundamental difference, and therefore we believe it necessary to consider entities appearing in feature models and configurations as instances of different meta-entities.

Further, especially in the presence of feature cardinalities, it may happen that "a configuration may include several different variants of the same feature" [2]. We believe that the notion of types and instances is a concise and easily understood way of characterising the relationship between feature in a model and its variants in a configuration.

The notion of feature types facilitates the reuse of feature knowledge. In Forfamel, a feature type may naturally appear as a possible type in multiple subfeature definitions. In previous work on feature modelling, additional concepts such as "feature-diagram reference" [3] and "macro" [21] have been required to enable reusing features within a model. The reuse possibilities are further enhanced by the possibility of defining subtypes that inherit the properties of their supertypes.

5.3. Subfeature definitions

The notion of subfeatures is quintessential in any feature modelling method. This is also the case in Forfamel. However, the mechanism used in Forfamel for representing subfeatures in feature models is somewhat different from those found in previous methods.

As shown in Figure 2 (c), a subfeature definition has three attributes: an optional name, a set of possible cardinalities, and a similarity definition.

The name of a subfeature is new in feature modelling. The intuition is that if a name is given, it specifies the role that the features specified by the definition have in parent feature; in many cases the role is apparent from the feature type, and no separate name is needed.

There are a number of factors motivating the inclusion of a subfeature name. First, constituent elements are identified by their names, not types, in many widely applied modelling methods, such as classes and their members in object-oriented languages and properties in UML. Second, a role name enables features of the same type to have multiple roles as subfeatures of the same feature; the subfeature definitions *primary* and *secondary* of the running example (Figure 3) demonstrates this point.

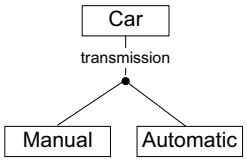

Figure 5 An example of subfeature names

Further, subfeature names can be used to simplify feature models: e.g., Figure 5 illustrates how a fragment of the feature model of Figure 1 can be represented fewer levels of subfeatures, with the aid of subfeature names. Of course, there are still further ways to represent the same fragment, e.g., using subtyping. However, we do believe that subfeature names may be a useful modelling construct in many cases. We emphasise that subfeature names are optional: the example of Figure 3 shows multiple cases in which the subfeature names have been omitted.

A subfeature definition is a natural place to store knowledge about cardinality. This is in contrast to most previous papers on feature modelling, in which the terms 'mandatory feature' and 'optional feature' have been extensively used. These terms suggest that being mandatory or optional is a property of a feature. This point is demonstrated by the running example (Figure 3): the feature type *Equation editor* could be characterised to be both a mandatory and an optional subfeature of *Text editor*. Recently, this point has been acknowledged by Czarnecki *et al*., but still left implicit in their metamodel [3].

In previous work it has been suggested that features can be classified based on their role in a feature model as grouped features, solitary features, and root features [3], see Figure 6 for a fragment of the metamodel. In Forfamel feature types contain no knowledge about their role as subfeatures of other features; such knowledge is contained in subfeature definitions instead. A practical argument to support this view is that including such knowledge in a feature would conceptually prohibit reusing the feature in different context. A similar argument applies to classifying a feature as a root feature, see Figure 6: arguably, a feature that is a root feature in a feature model should be reusable as a non-root feature in another model. In Forfamel, on the other hand, the root feature is encoded as a property of a feature model, not of any feature type.

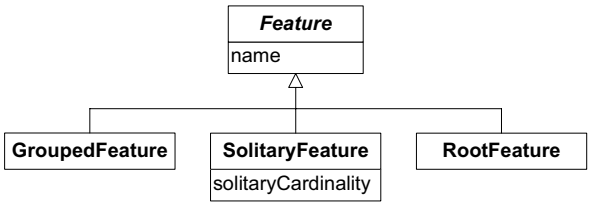

Figure 6 A fragment of a UML metamodel for cardinality based feature modelling [3]

Cardinalities and similarity definitions are closely related through the questions: Given a subfeature definition that allows a cardinality greater than one and includes multiple possible part types, can the same type be selected multiple times? Must all the subfeatures be of the same type? The similarity definition can be used to specify the desired semantics.

In previous work on cardinality-based feature modelling [2], this issue has been resolved differently: a feature in a group of features may only be selected once. Under this semantics, it is difficult to represent certain practically relevant situations. Consider the fragment of feature model in Figure 7 (a): a car has four tires, and all of them must be either slicks or rain tires; mixing is not an alternative. We cannot think of a way expressing this fragment without resorting to constraints using the feature modelling method of [2].

The notion of cardinality in Forfamel is different from that suggested in [2] additional ways. First, in [2] cardinalities are expressed as a sequence of integer intervals, e.g., [0..1][2..4], whereas in Forfamel cardinalities are represented as a set of integers. It is not difficult to see that these two representations are equivalent.

A more fundamental difference is that [2] allows specifying an unbounded cardinality, denoted by the symbol *. As the example in Figure 7 (b) shows, there are natural cases in which there is no natural upper bound for the cardinality of a subfeature: a text editor may have any number of plug-ins.

Unfortunately, an unbounded cardinality leads into cases in which some or all the valid configurations of a model are infinite, i.e., contain an infinite number of features: consider a constraint on subfeature a of unbounded cardinality with integer attribute i stating that for each subfeature a_1, there exists a subfeature a_2 such that $a_2.i > a_1.i$. It is easy to see that there is no finite configuration that satisfies this constraint, but there are an infinite number of infinite configurations that do satisfy it. Dealing with infinite configurations may not be impossible, but surely more difficult than with finite configurations. This is why we have excluded unbounded cardinalities from Forfamel.

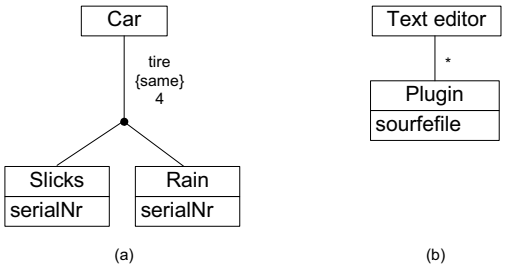

Figure 7 Examples of similarities

Similarly as above, infinite valid configurations would also result if feature types were allowed to define subfeatures with themselves as possible subfeature types.

5.4. Constraints

All feature modelling methods include a form of constraints. In most methods, these have been of the form *A requires B*, or *A* excludes *B*. More recently, it has been argued that more comprehensive forms of constraints are required [13]. A construct similar to constraints called product sets has been suggested to support complex product decisions. [22].

We did not specify a constraint language as part of Forfamel in this paper. However, we do believe that a comprehensive constraints language is essential for expressing the potentially complex dependencies between features in industrial software product families.

6. Conclusion and further work

We have presented Forfamel, a feature modelling method that synthesises modelling concepts and constructs found in existing feature modelling methods. It does this by rigorously defining the relationship between a feature model and its valid configurations. In addition, Forfamel includes a number of extensions not found in existing feature modelling methods. Most importantly, subfeature names can be used to specify the role of a subfeature. Features may be defined arbitrarily many attributes, each of which may be set-valued.

To be practically applicable, the work presented in this paper must be augmented in several ways. Although we have evaluated the language and initial tool support briefly described in this paper, additional studies are needed and further development may be required based on the outcomes. Forfamel could be extended to support multi-staged configuration. Also, as many of the computational problems related to Forfamel and feature models in general are computation-

ally hard, the feasibility of solving them in practice must be verified.

7. Acknowledgements

We gratefully acknowledge the financial support from HeCSE (Helsinki Graduate School on Computer Science and Engineering) and Tekes (Finnish Funding Agency for Technology and Innovation). We thank Ms. Varvana Myllärniemi for implementing Kumbang Configurator.

8. References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S. A. Peterson, *Feature-Oriented Domain Analysis (FODA) - Feasibility Study*, Report number CMU/SEI-90-TR-21, 1990.

[2] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing Cardinality-based Feature Models and Their Specialization", *Software Process: Improvement and Practices*, 10, 2005, pp. 7-29

[3] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged Configuration through Specialization and Multilevel Configuration of Feature Models", *Software Process: Improvement and Practices*, 10, 2005, pp. 143-169

[4] K.Czarnecki and U.W.Eisenecker, *Generative Programming*, Addison-Wesley, 2000.

[5] K. Czarnecki, T. Bednasch, P. Unger, and U.W. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", *in: Proceedings of GPCE'02*, 2002.

[6] J. Lee, K.C. Kang, and S. Kim, "A Feature-Based Approach to Product Line Production Planning", *in: Proceedings of SPLC 2004*, 2004.

[7] K.C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets - a Case Study", *in: Proc. of SPLC 2005*, 2005.

[8] P. Dolog and W. Nejdl, "Using UML-based Feature Models and UML Collaboration Diagrams to Information Modelling for Web-Based Applications", *in: Proceedings of UML 2004*, 2004.

[9] M. Eriksson, J. Börstler, and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations", *in: Proc. of SPLC 2005*, 2005.

[10] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants", *in: Proceedings of GPCE'05*, 2005.

[11] T. von der Maßen and H. Lichter, "Determining the Variation Degree of Feature Models", *in: Proceedings of SPLC 2005*, 2005.

[12] M. Mannion, "Using First-Order Logic for Product Line Model Validation", *in: Proceedings of SPLC2*, 2002.

[13] D. Batory, "Feature Models, Grammars, and Propositional Formulas", *in: Proceedings of SPLC 2005*, 2005.

[14] W. Zhang, H. Zhao, and H. Mei, "A Propositional Logic-Based Method for Verification of Feature Models", *in: Proceedings of ICFEM 2004*, 2006.

[15] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models", *in: Proceedings of CAiSE'05*, 2005.

[16] Object Management Group, *OCL 2.0 Specification*, Report number ptc/2005-06-06, 2005.

[17] P. Simons, I. Niemelä, and T. Soininen, "Extending and Implementing the Stable Model Semantics", *Artificial Intelligence*, 138, 2002, pp. 181-234

[18] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen, "Representing Configuration Knowledge with Weight Constraint Rules", *in: Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming*, 2001.

[19] V. Myllärniemi, T. Asikainen, T. Männistö, and T. Soininen, "Kumbang Configurator - A Configuration Tool for Software Product Families", *in: Configuration workshop of IJCAI-05*, 2005.

[20] L. Hotz, T. Krebs, and K. Wolter, Combining Software Product Lines and Structure-Based Configuration - Methods and Experiences. Available as: *http://www.soberit.hut.fi/SPLC%2DWS/Presentations/MacGregorPresentation%20ConIPF_At_SPLC.pdf*.

[21] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger, "XML-Based Feature Modelling", *in: Proceedings of ICSR-8*, 2004.

[22] M.-O. Reiser and M. Weber, "Using Product Sets to Define Complex Product Decisions", *in: Proceedings of SPLC 2005*, 2005.