Timo Asikainen, Tomi Männistö, and Timo Soininen. 2007. Kumbang: A domain ontology for modelling variability in software product families. Advanced Engineering Informatics, volume 21, number 1, pages 23-40.

Available online at www.sciencedirect.com
ScienceDirect
Advanced Engineering Informatics 21 (2007) 23–40
ADVANCED ENGINEERING INFORMATICS
www.elsevier.com/locate/aei

Kumbang: A domain ontology for modelling variability in software product families

Timo Asikainen *, Tomi Männistö, Timo Soininen

Helsinki University of Technology, Software Business and Engineering Institute (SoberIT), P.O. Box 9210, FIN-02015 TKK, Finland

Received 6 October 2006; accepted 12 November 2006

Abstract

Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context. There is an ever-growing demand for variability of software. Software product families are an important means for implementing software variability. We present a domain ontology called Kumbang for modelling the variability in software product families. Kumbang synthesises previous approaches to modelling variability in software product families. In addition, it incorporates modelling constructs developed in the product configuration domain for modelling the variability in non-software products. The modelling concepts include components and features with compositional structure and attributes, the interfaces of components and connections between them, and constraints. The semantics of Kumbang is rigorously described using natural language and a UML profile. We provide preliminary proof of concept for Kumbang: the domain ontology has been provided with a formal semantics by implementing a translation into a general-purpose knowledge representation language with formal semantics and inference support. A prototype tool for resolving variability has been implemented.

Keywords: Software product family; Variability; Modelling; Feature modelling; Software architecture; Kumbang

1. Introduction

Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context [1]. There is a growing demand for variability of software, and a significant research interest in the topic, as exemplified by the workshops and special issues devoted to it, see, e.g., [2]. Products that incorporate variability can be useful for various purposes: for example, such products can address multiple user segments, allow price categorisation, support various hardware platforms and operating systems, provide different sets of features for different needs, and cover different market areas with different languages, legislation, and market structure. Addressing these concerns without variability would be very difficult, if not impossible.

Software product families, or *software product lines*, as they are also called, have become an important means for implementing variability [3]. A software product family is commonly defined to consist of a *common architecture*, a *set of reusable assets* used in systematically producing individual products, and the *set of products* thus produced [4]. Intuitively, variability in software product families pertains to the fact that different products in the family have many similarities, but also differ from each other in significant ways.

A software product family may contain very large numbers of individual products. Consequently, methods for representing the variability and efficiently reasoning about it are needed. Towards this end, numerous methods for modelling the variability in software product families have been proposed. A practically important class of such

* Corresponding author. Tel.: +358 9 451 5364; fax: +358 9 451 4958.
E-mail address: timo.asikainen@tkk.fi (T. Asikainen).

1474-0346/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2006.11.007

methods is based on modelling the common and variable *features* of a product family. An example of such a method is FODA (feature oriented domain analysis) [5]. In the context of such methods, a feature is defined as "an end-user visible characteristic of a system" [6]. Features are typically organised in a directed tree or other acyclic graph that more or less rigorously defines which features combinations represent valid individuals of the product family. On the other hand, a number of methods for modelling variability in product family architectures have been reported; Koalish [7] and xADL 2.0 [8] are examples of such methods. Product family architectures are typically described in terms of *components*, their connection points called *interfaces*, the compositional structure of components, and *connections* or *connectors* between interfaces. Unfortunately, most, if not all, variability modelling methods lack a well-defined conceptual foundation. They are not rigorous in specifying the modelling concepts, their interrelations and semantics: some authors seem to be more concerned with notation than its meaning. We believe that this condition severely undermines the practical applicability of variability modelling methods.

Variability has been studied in the domain of traditional products, i.e., mechanical and electrical ones. This domain of research is called *product configuration*, or *configuration* for short, and it studies how a general design of a product can be modified, or in other words, *configured*, in prescribed ways to produce product individuals that match the specific needs of customers [9]. Configuration as an activity is highly similar to *routine design* [10]: routine design proceeds within a well-defined state space of potential designs. In contrast to methods for modelling variability in software product families, the results achieved in product configuration domain include a number of conceptualisations of the domain [11,12]. The conceptual work done in the domain has lead to a large number of successful applications [9,13,14]. We believe that much research effort has been wasted when the lessons learnt in the configuration domain have not been transferred to software product family research, even though the transfer of results may not be straightforward due to the differences in the nature of software and physical products.

In this paper, we introduce *Kumbang*, a domain ontology for modelling the variability in software product families. Kumbang unifies the feature and architecture based approaches to modelling variability in software product families. In addition to modelling variability from feature and architectural point of view, Kumbang enables modelling the interrelations of these two views. Unlike its predecessors, Kumbang is rigorously described using both natural language and a UML (Unified Modeling Language) version 2.0 [15] profile. UML is the de facto standard modelling method in the software engineering domain and profiles are its built-in extension mechanism. Kumbang incorporates many lessons learnt in the product configuration domain.

We believe that Kumbang is of great practical value. In addition its rich set of well-defined modelling concepts and constructs, a number of other points speak for its applicability. First, describing Kumbang in terms of UML makes it easier for people already familiar with UML to adopt Kumbang. Further, the fact that Kumbang is described using a UML profile makes it, at least in principle, possible to use existing UML 2.0 compliant tools to provide tool support for Kumbang. The modelling capabilities of Kumbang are demonstrated using an example based on a real-life product.

The remainder of this paper is organised as follows. Next, in Section 2 we discuss previous work relevant to the paper: discussed topics include methods for representing variability in software product families, and an overview of product configuration research, especially from the point of view of representing variability. Thereafter, in Section 3 follows the main contribution of this paper: definition of Kumbang, a domain ontology for modelling variability in software product families. Section 4 provides an overview of the proof of concept existing for Kumbang. Discussion and comparison to previous work follow in Section 5. Conclusions and an outline for further work round up the paper in Section 6.

2. Previous work

In this section, we discuss the relevant previous work that lays the foundation on which this work builds. We begin by discussing software product families. Thereafter, we discuss the notion of variability and existing methods for modelling variability in software product families. Finally, we provide an overview of the product configuration domain.

2.1. Software product families

This subsection introduces the concept of *software product family*, or *line*, an alternative term used. There are a number of definitions for the concept [3,4,16]. Clements and Northrop define the concept as follows [3]:

> A *software product line* is a set of software-intensive systems sharing a common, managed 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.

Jan Bosch defines the concept somewhat differently [4]:

> A *software product line* consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using the mentioned reusable assets.

A comparison between the two definitions indicates that there are both remarkable commonalities and differences between them. Most significantly, the definitions share the notion of a *set of reusable* or *core assets*. Also, both definitions include the set of products or systems developed using these assets.

Despite the similarities, the definitions are far from being equivalent. The Clements and Northrop definition can be seen as market driven: one of the defining characteristics of a software product family is the satisfying of "the specific needs of a particular market segment". Bosch's definition emphasises the role of *software architecture* as a part of a software product family. Thereby, his definition can be said to be technology-driven.

There are still some minor differences between the definitions. First, the Clements and Northrop definition includes the requirement missing from Bosch's definition that the systems belonging to the product family are developed "in a prescribed way". The Bosch definition, on the other hand, postulates that the core assets belonging to the product family are designed to "for incorporation into the product family architecture". However, these differences seem not to imply that the defined concepts were radically different; instead, the definitions can be considered to emphasise different aspects of the same phenomenon.

The activities related to a software product family are typically organised into two phases, namely the *development* and *deployment* [4]; other authors use the terms *domain* and *application engineering* to refer to roughly the same phenomena, respectively [6]. During the development process, the software product family architecture and components implementing the common part, i.e., components present in all individual products in the product family, are implemented. The deployment process involves deriving individual products from the product family, based on a set of specific market or customer requirements. The architecture and components from the above-discussed development process form the basis for the deployment process: they are adapted to match the requirements for the individual product being deployed. However, in a typical case, product-specific code must be developed to implement product-specific requirements that are not covered by the assets from the development process.

A *configurable software product family* [3] is such a software product family for which there is no or little need to develop product-specific software during the deployment process. In other words, the product-specific requirements can be met by configuring the reusable assets developed during the development process. Reported cases from the industry include MarketMaker [3], Securitas [4], Salion [17], and two anonymous cases [18,19].

Variability and its management are an important part of product family engineering. Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context [1]. *Software variability management* and has been the topic of a number of workshops [20–22], and a special issue [2]. A definition for variability management is [23]:

> Variability management encompasses the activities of explicitly representing variability in software artefacts throughout the lifecycle, managing dependencies among different variabilities, and supporting the instantiations of the variabilities.

Variability and its management are key characteristics that distinguish software product family engineering from other approaches to software development [24].

2.2. Modelling variability in software product families

A large number of methods for modelling variability in software product families have been reported. These can be classified into three broad categories: feature-based methods, methods based on modelling variability in software architectures, and methods that do not commit to any specific set of concepts. Below we provide a brief overview of the first two categories; a more detailed account can be found in [25].

2.2.1. Feature modelling

Feature modelling has become a popular method for modelling software variability. A large number of feature modelling methods have been suggested, e.g., [5,6,26,27]. The methods are based on the notion of *feature*. However, different methods use somewhat different definitions of feature: popular definitions of feature include "an end-user visible characteristic of a system", and "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept" [6]. Also, feature has been characterised as capturing a considerable set of requirements and representing a logical unit of behaviour [4].

All feature modelling methods are based on the notion of *feature model*; alternative terms include *feature diagram* and *feature graph*. A feature model describes the common and the variable features of a software product family. Individual products are distinguished from each other through the features they deliver. Feature models of industrial software product families can be very large, consisting of hundreds or even thousands of features [28–30]. A feature model is a description of a system family, e.g., a software product family. A *feature configuration* is a description of an individual product. A feature configuration may consist of a subset of the features in the model, or have a more complex structure.

Fig. 1 illustrates a sample feature model represented using FODA (Feature Oriented Domain Analysis) [5], the first feature modelling method reported. A feature model in FODA is a directed tree, or more generally, a rooted, directed, acyclic graph. The root is sometimes referred to as *concept*. The root feature has a number of features as its subfeatures, and these may in turn have other features as their subfeatures, et cetera. 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

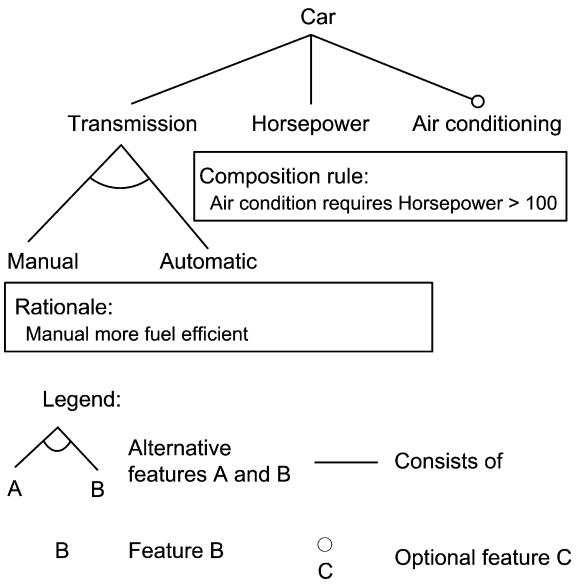

Fig. 1. A sample feature model describing the features of a car, adopted from [5]. The model illustrates the basic modelling concepts of the FODA (Feature Oriented Domain Analysis) method [5], the first feature modelling method reported in the software engineering domain.

the parent feature is selected. A feature configuration, i.e., a description of an individual product, in FODA is such a subset of the features in the model that obeys the rules of the model.

Feature modelling methods reported after FODA are mainly based on the same set of modelling concepts, although a number of additional concepts have been suggested. In FORM (Feature Oriented Reuse Method) [26], features are classified into four layers, namely *capability layer*, *operating system layer*, *domain technology layer*, and *implementation technique layer*. Further, there are three possible relationships between features: *generalisation/specialisation*, *composed of*, and *implemented by*, while there is only one in FODA (*consists-of*). Czarnecki and Eisenecker have introduced *or-features* that are similar to alternative features in FODA, with the exception that instead of selecting exactly one, any number of alternatives greater than equal to one may be selected [6]. They extend their earlier work with *cardinalities*, *attributes*, and *reference attributes* [31]. Cardinalities can be used to specify how many subfeatures of a specific kind a feature must have, e.g., a car could be specified to include exactly four tires. Attributes, in turn, parameterise features. Finally, reference attributes can be used to refer to other features in the feature hierarchy.

The term feature has also been used in other ontological work. However, in the cases we know of, the term is used in a significantly different meaning than in our work. For instance, in [32] the term feature is used to describe the different design conditions that impact construction cost. The difference between this use of the term and ours is that in our work features are used to describe the variabilities and commonalities of a family of systems, whereas in [32] features are design conditions in a single system, a design of a building.

2.2.2. Modelling product family architectures

The level of design concerning the overall structure of software systems is commonly referred to as the *software architecture* level of design. This level includes structural issues such as the organisation of a system as a *composition of components*, the *protocols* for communication, the *assignment of functionality* to design elements, etc. [33]. A definition for software architecture is [34]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relations among them.

Informally, software architecture is used to refer to the structure of a software system on a high level of abstraction. Software architecture does not concern the fine-grained structure or the properties of a software system, or the process used to develop it [35].

A large number of *architecture description languages* (ADLs) have been developed [36]. All ADLs share the notion of *component* as the basic concept for describing software architecture; a component is typically defined as a locus of computation. In addition, most ADLs include concepts for describing the connection points of components, typically termed *ports*; and *connectors*, entities mediating communication between components. A typical ADL is intended for describing the architecture of a single system: they lack concepts and constructs for modelling the variability in product family architectures. Hence, most

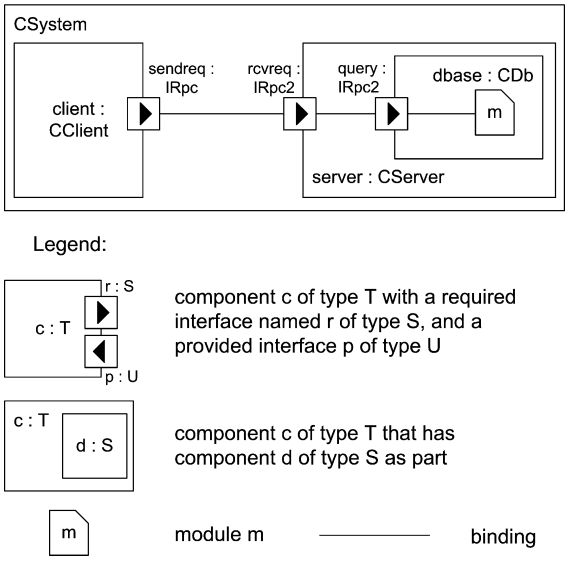

Fig. 2. The architecture of a client–server system described using Koala [37], a component model and architecture description language developed and used at Philips Consumer Electronics. The figure illustrates the main modelling concepts of the Koala method.

of them are not directly applicable to modelling product family architecture. There are exceptions to this rule, including Koala [37] and its derivative Koalish [7] and xADL 2.0 [8].

Koala [37–39] is a component model and an ADL developed at Philips Consumer Electronics. Fig. 2 contains an example of an architecture description in Koala. Koala is one of the few ADLs that have been applied in the industry [39] and is therefore of special interest from a practical point of view.

The main modelling elements of Koala are *components* that have *interfaces* as their explicit connection points. A component is defined as "an encapsulated piece of software with an explicit interface to its environment"; an interface is a "small set of semantically related functions". *Functions* correspond to function signatures in, e.g., the C programming language. Each interface is either a *requires* or a *provides* interface. A requires interface signals that the component containing the interface requires some service from its environment. Similarly, a provides interface signals that the component provides such a service to its environment. There may be *bindings* between interfaces, with the intuitive semantics that a requires interface bound to a provides interface gets its required services from the provides interface.

Components may *contain* other components. The semantics of containment is that of encapsulation: a component that is a part of another component can only be accessed from within the containing component. A *configuration* is a component that is not a part of any other component and has no interfaces. A configuration can be thought of as an independent system that can be deployed in a product.

Koala includes a number of variability mechanisms. A configuration is characterised by a set of *parameters*. Systems with different parameter values are different: the functionality of a system may directly vary based on the parameters. There may be many alternative bindings between interfaces, and different alternatives lead to different systems. An interface may be declared *optional*. Such an interface may but is not required to contained in an instance of the component type defining it.

Koalish [7] extends Koala with explicit mechanisms for modelling variability. In Koalish it is possible to define a *set of possible types* and a *cardinality* for contained components: this enables variability in the type and number of components contained by a given name. *Constraints* can be used to restrict the set of valid individual systems. Koalish is our own earlier work and much of it has been used as a part of the Kumbang domain ontology to be presented in the following section.

xADL 2.0 [8,40] is an infrastructure for rapid development of XML (eXtended Markup Language)-based architecture description languages. Hence, xADL 2.0 is not merely an ADL, but provides facilities for defining customised ADLs. Below we concentrate on the core constructs of xADL 2.0.

The basic modelling elements of xADL 2.0 include *component type*, *connector type*, and *interface type*. Component types may be defined a compositional structure. Interfaces are connection points of components, and they may be bound together using connectors. The variability modelling elements of xADL include *optional elements*, *variant types*, and *optional variant elements*. Optional elements have the intuitive semantics that instances may but are not required to be included in an individual system. Variant types pertain to choosing one out of two or more elements. Finally, optional variant elements are the combination of optional elements and variant types. The control over whether to include an optional element, and which variant should be selected is in *Boolean guards*. They are Boolean expressions on a number of variables. The authors of xADL 2.0 refer to the forms of variability discussed above as *space variabilities* [41]. In addition, xADL 2.0 supports *time variabilities* in the form of *evolution* of design elements. In short, the evolution is described in terms of *versions* and *branches*.

2.2.3. Product configuration

The purpose of this subsection is to provide an overview of product configuration research, a subfield of artificial intelligence [13].

Research in the product configuration domain is based on the notion of *configurable product*: a configurable product is such a product that each product individual is adapted to the requirements of a particular customer order. Historically, the configurable products studied in the domain have been non-software products, often mechanical and electronics products. A modular structure is typical of a configurable product: product individuals consist of pre-designed components, and selecting different components leads to different product variants [9].

The possibilities for adapting the configurable product are predefined in a *configuration model* that explicitly and declaratively describes the set of legal product individuals. A specification of a product individual, *configuration*, is produced in the configuration task based on the configuration model and a set of customer requirements.

There are two widely sited conceptualisations of configuration knowledge [11,12]. The most important concepts in these are: *components*, *ports*, *resources*, and *functions*. Components represent distinguishable entities in a product: a configuration is composed of components, and components may in turn be composed of other components. Ports are connection interfaces, either physical or logical, of components. Ports may be connected with each other. Resources, in turn, are entities that are produced and consumed by components. In a configuration, the production and consumption of each resource kind must be balanced. Finally, functions are abstract characterisations of a product that a customer or sales person would use to describe it.

Efficient knowledge-based information systems, *product configurators*, or *configurators* for short, have become an important and successful application of artificial intelligence techniques for companies selling products adapted

to customer needs [13]. The basic functionality of a configurator is to support a user in finding a configuration of a given configuration model matching his specific needs. Examples of the kinds of support provided are: A configurator represents the available choices in a way that enables the user to easily enter his requirements. The configurator makes deductions based on the requirements the user has entered so far, and prevents or discourages the user from making incompatible choices. Finally, the user can at any point ask the configurator to automatically find a configuration that is valid with respect to the configuration model and satisfies the requirements entered so far. The above-described functionality is based on using declarative configuration models and efficient, sound, and complete inference tools operating on these.

Configurators have reportedly been applied to a number of different kinds of products; perhaps the most challenging cases have been telephone switching systems at Siemens [42], and other kind of telecommunication products at AT&T [43]. At Siemens, the problem instances have been considerably large: typically, a configuration has included tens of thousands of components with hundreds of thousands of attributes, and over 100,000 connection points. Configurators have become parts of ERP (Enterprise Resource Planning) systems, such as SAP [44], and Baan [45].

3. Kumbang

In this section, we define *Kumbang*, a domain ontology for representing variability in software product families. Kumbang includes concepts for modelling variability both from the feature and architecture point of views and the interrelations between these two views. Unlike most, if not all, existing methods for modelling the variability in software product families, Kumbang is rigorously described using both natural language and a UML (Unified Modeling Language) 2.0 [15] profile.

3.1. Foundation: Kumbang models and configurations

Kumbang is based on three layers of abstraction. At the highest level of abstraction is the *metalayer* that contains the modelling concepts, or *metaclasses*. The next layer is the *model layer* that contains Kumbang models. The entities that appear in Kumbang models are termed *classes* and are instances of metaclasses. Finally, the third layer, *instance layer*, contains the instances of the classes appearing at the model layer.

We describe Kumbang predominantly by characterising Kumbang models, their constituent entities, and the relations between entities in the model and instance layers. We use expressions such as "a Kumbang model" when we mean "an instance of the metaclass Kumbang model", or "Kumbang instances" when meaning "instances of the metaclass Kumbang instance".

A *Kumbang model*, or *model* for short, is a representation of the variability in a software product family. A *Kumbang configuration* (*configuration*) is a representation of an individual software product. A Kumbang model defines a (possible empty) set of Kumbang configurations that are *valid* with respect to the model. Intuitively, a valid configuration is a configuration that conforms to the model and is hence a representation of a valid individual of the product family. More specific requirements for a valid configuration will be defined in the remaining subsections.

UML. We give the UML 2.0 presentation of Kumbang concepts under separate headings. We assume the reader is familiar with UML. The language is defined in an OMG (Object Management Group) standard [15]. We define the Kumbang ontology as a profile extending the UML metamodel, see Fig. 3 for an illustration of the profile. The stereotypes appearing in the profile correspond to metaclasses in the terminology introduced in the beginning of this subsection. We do not explicate the correspondences between metaclasses and the corresponding UML stereotypes as these are obvious from their names. Fig. 3a illustrates that *KumbangModel* is defined as a stereotype extending the metaclass *Model (from Models)*. *KumbangConfiguration* is defined as a stereotype extending the same metaclass.

A Kumbang model consists of a set of *Kumbang types*. A Kumbang type serves as a description of its instances, *Kumbang instances*, which are entities appearing in Kumbang configurations. Kumbang type is an abstract metaclass that has three subclasses, namely *composable type*, *interface type*, and *attribute type*, see Fig. 3b. Of these, the latter two are concrete metaclasses, whereas composable type is an abstract metaclass with two concrete subclasses, *component type* and *feature type*. Each of these metaclasses will be discussed in more detail in the remaining subsections.

A *Kumbang instance* is a description of an entity in a product individual. Kumbang instance is an abstract metaclass. Similarly as for Kumbang type above, Kumbang instance is the root of a taxonomy of metaclasses, see Fig. 3c for the UML representation.

UML. *KumbangType* is represented as an abstract stereotype extending the metaclass *Classifier (from Kernel)*. The subclasses of Kumbang type are specialisations of the *KumbangType* stereotype. *KumbangInstance* is an abstract metaclass extending the metaclass *InstanceSpecification (from Kernel)*. The subclasses of Kumbang instance are represented as specialisations of *KumbangInstance*. The properties *rootComponent* and *rootFeature* corresponding to the root component and root feature types are defined for *KumbangModel*.

3.2. Types and taxonomy

Kumbang types are organised into taxonomies defined by the *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, 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

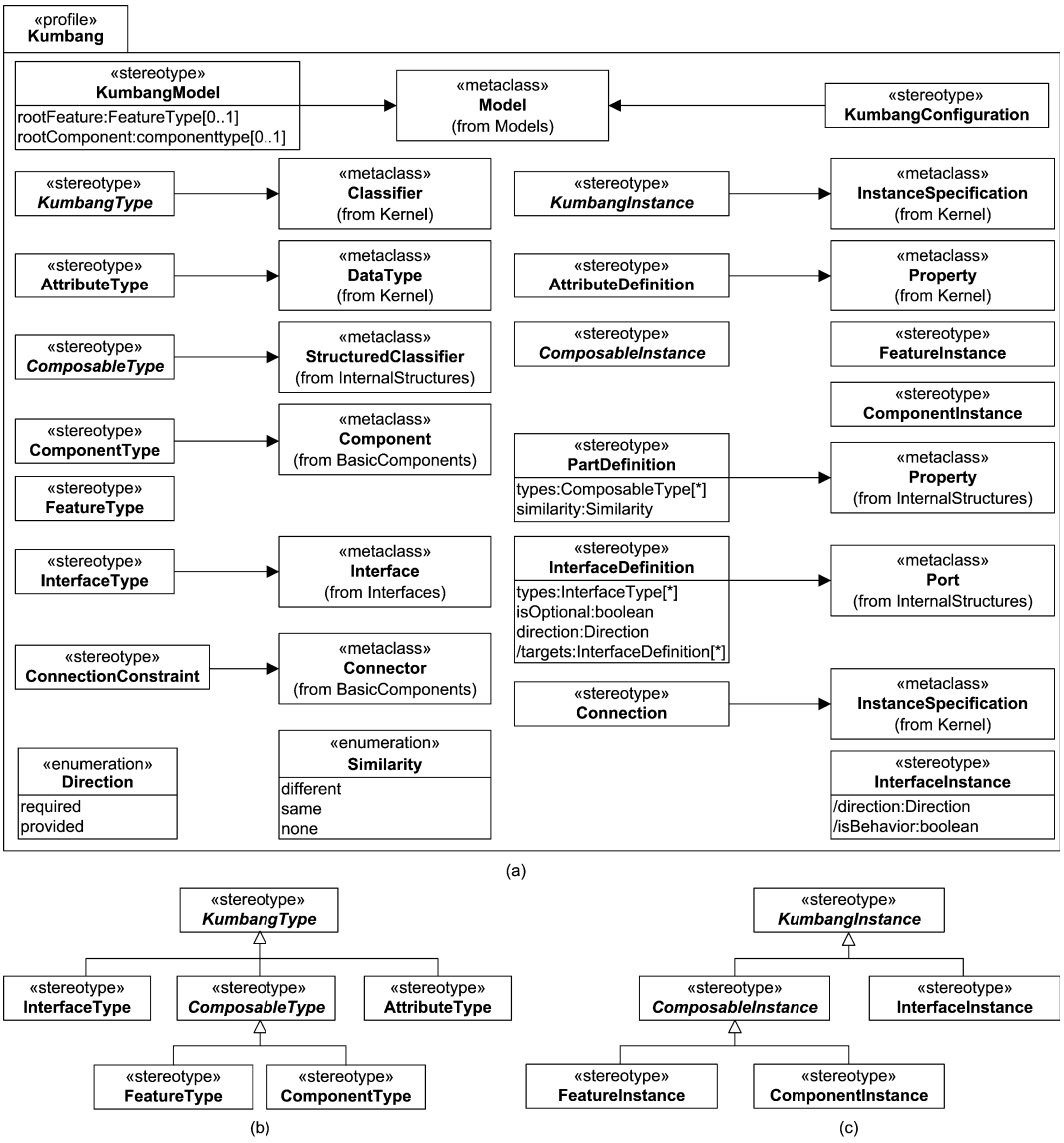

Fig. 3. The Kumbang profile used to represent the Kumbang domain ontology using UML concepts. (a) Stereotypes are used to represent individual concepts in Kumbang. Each stereotype extends a UML metaclass. (b) Stereotypes may be organised in a taxonomy. Here, the taxonomy of stereotypes representing Kumbang types is presented. (c) Taxonomy of Kumbang instances.

implies that T_2 is a *supertype* of T_1. We require that in a well-formed model the transitive closure of the *isa* relation is asymmetric. In addition, we require that only pairs of types (T_1, T_2) where T_1 and T_2 are instances of the same concrete metaclass may belong to the *isa* relation. This implies, for example, that component types may only have other component types as their supertypes. Each Kumbang type is either *abstract* or *concrete*.

A subtype inherits the properties of its supertypes. The properties that are inherited include *attribute definitions*, *part definitions*, *interface definitions*, and *constraints*. Each of these classes of properties will be discussed in more detail in the remaining subsections.

Each Kumbang instance is (directly) of exactly one type; this type is termed the *type* of the instance, and the instance is said to be an *instance* of that type. In addition, each instance is an *indirect instance* of all the supertypes of its type.

Semantics. We describe the semantics of Kumbang under separate headings. Only concrete types may have instances in a valid configuration. All the instances in a valid configuration must be *valid*. Intuitively, a valid instance conforms to its type; more specific requirements for a valid instance are defined in subsequent subsections.

UML. In UML, the base type for Kumbang types, i.e., *KumbangType*, is a stereotype extending the metaclass *Classifier (from Kernel)*. Further, the taxonomy between types is represented using the *Generalisation* relation between the types.

Example. Fig. 4 introduces a running example that will be used to demonstrate Kumbang concepts throughout this paper. The example is based on an industrial product family, a car periphery system (CPS) by Robert Bosch GmbH

```
Kumbang model CarPeripherySystem
  root component CPSArchitecture,  root feature CPSFeatures

feature type CPSFeatures
  subfeatures
    (ParkingAssistance, PreCrash) application[1-2] {diff};
  constraints
    value(application:ParkingAssistance, symmetry) =
      PassengerSideOnly => not present(application:PreCrash)

feature type ParkingAssistance
  subfeatures
    Zone imminent, veryNear, near, inProximity[0-1];
    (Alarm, Display) userinterface[1-2] {diff};
  attributes Symmetry symmetry;
  implementation
    present($.software.application:ParkingSW);
    value(symmetry) = PassengerSideOnly =>
      instanceOf($.software.sensorSW, SingleSensorSW);

feature type Zone
  attributes Depth triggerDepth;
  implementation
    value(triggerDepth) <= value($.hardware.sensorHW,
      supervisionDepth);

feature type Alarm
  attributes AlarmType alarm;
  implementation
    value(alarm) = Buzzing <=> hasInstances(Buzzer);
    value(alarm) = Speaker<=> hasInstances(Loudspeaker);

feature type Display
  attributes DisplayType display;
  implementation
    value(display) = MediumTFT <=> hasInstances(TFT1);

feature type PreCrash
  attributes Probability wrongActionProbability;
  implementation
    present($.software. precrashSW);
    instanceOf($.hardware.sensorHW, SSRSensorSet);
    value(wrongActionProbability) = VeryUnlikely =>
      hasinstances(COI2D);
    value(wrongActionProbability) = ExtremelyUnlikely =>
      cardinality($.software.precrashSW.coi) = 2;

component type CPSArchitecture
  contains Software software; Hardware hardware;
  constraints
    instanceOf(software.sensorSW, SingleSensorSW) <=>
      instanceOf(hardware.sensorHW, SingleSensorSet);

component type Software
  contains
    SensorSoftware sensorSW;
    PreCrashSW precrashSW[0-1];
    (ParkingSW, PreSetSW, PreFireSW) application[1-3] {diff};
  connects
    application:PreSetSW->crashinput = precrashSW->precrash;
    application:PreFireSW->crashinput = precrashSW->precrash
    application:ParkingSW->sensorInput = sensorSW->sdata;

abstract component type SensorSoftware
  provides ISensor sdata {grounded};

component type SingleSensorSW extends SensorSoftware;
component type MultiSensorSW extends SensorSoftware;

component type ParkingSW
  requires ISensor sensorInput;

component type PreSetSW
  requires IPrecrash crashinput;

component type PreFireSW
  requires IPrecrash crashinput;

component type PreCrashSW
  contains (COI1D, COI2D) coi[1-2] {diff};
  requires ISensor sensor;
  provides IPrecrash precrash { grounded };

component type COI1D; component type COI2D;

component type Hardware
  contains
    SensorSet sensorHW;
    (Buzzer, Loudspeaker, RadioAdapter) audioHW[0-1];
    (TFT1, TFT2) displayHW[0-1];

abstract component type SensorSet
  attributes Depth supervisionDepth;

abstract component type MultiSensorSet extends SensorSet;
abstract component type SingleSensorSet extends SensorSet;
abstract component type SSRSensorSet extends SensorSet;
abstract component type USSensorSet extends SensorSet;

component type 1xUSa extends SingleSensorSet, USSensorSet
  constraints value(supervisionDepth) = 100cm;

component type 1xUSb extends SingleSensorSet, USSensorSet
  constraints value(supervisionDepth) = 250cm;

component type 1xSSRa extends SingleSensorSet, SSRSensorSet
  constraints value(supervisionDepth) = 300cm;

component type 2xSSRa extends MultiSensorSet, SSRSensorSet
  constraints value(supervisionDepth) = 1000cm;

component type 2xSSR extends MultiSensorSet, SSRSensorSet
  constraints value(supervisionDepth) = 2000cm;

component type Buzzer; component type Loudspeaker;
component type RadioAdapter;

component type TFT1; component type TFT2;

attribute type Depth ={100cm, 250cm, 300cm, 1000cm, 2000cm}

attribute type Symmetry = { PassengerSideOnly, LeftAndRight }

attribute type AlarmType = { Buzzing, Speakern }

attribute type DisplayType = { BigTFT, MediumTFT }

attribute type Probability = { Unlikely, VeryUnlikely,
  ExtremelyUnlikely }

interface type ISensor { getSensorData() }

interface type IPrecrash { getPrecrashData() }
```

Fig. 4. A Kumbang model of a car periphery system (CPS) product family used as a running example in this paper. The model represents that variability of a car periphery system both in terms of the features individual systems in the family deliver, and their architecture, both hardware and software.

[46,47]. The model of the product family used as the basis for the running example is available as [48]. Car periphery systems are embedded systems based on sensors installed around the vehicle that monitor its environment. The data obtained from the sensors is used as input for different applications. Examples of such applications include parking assistance and pre-crash detection.

The example is expressed using *Kumbang language*, a prototype language that defines concrete syntax for Kumbang. The language is structured according to types, i.e., the properties of a type are collected under its definition.

The example includes several cases of subtyping. Fig. 5 illustrates the taxonomies of component and feature types as a UML class diagram. Fig. 5a shows that the abstract component type *SensorSoftware* has two concrete subtypes, *SingleSensorSW* and *MultiSensorSW*. Intuitively, different software is required for systems including only a single sensor or multiple sensors. Fig. 5b shows that con-

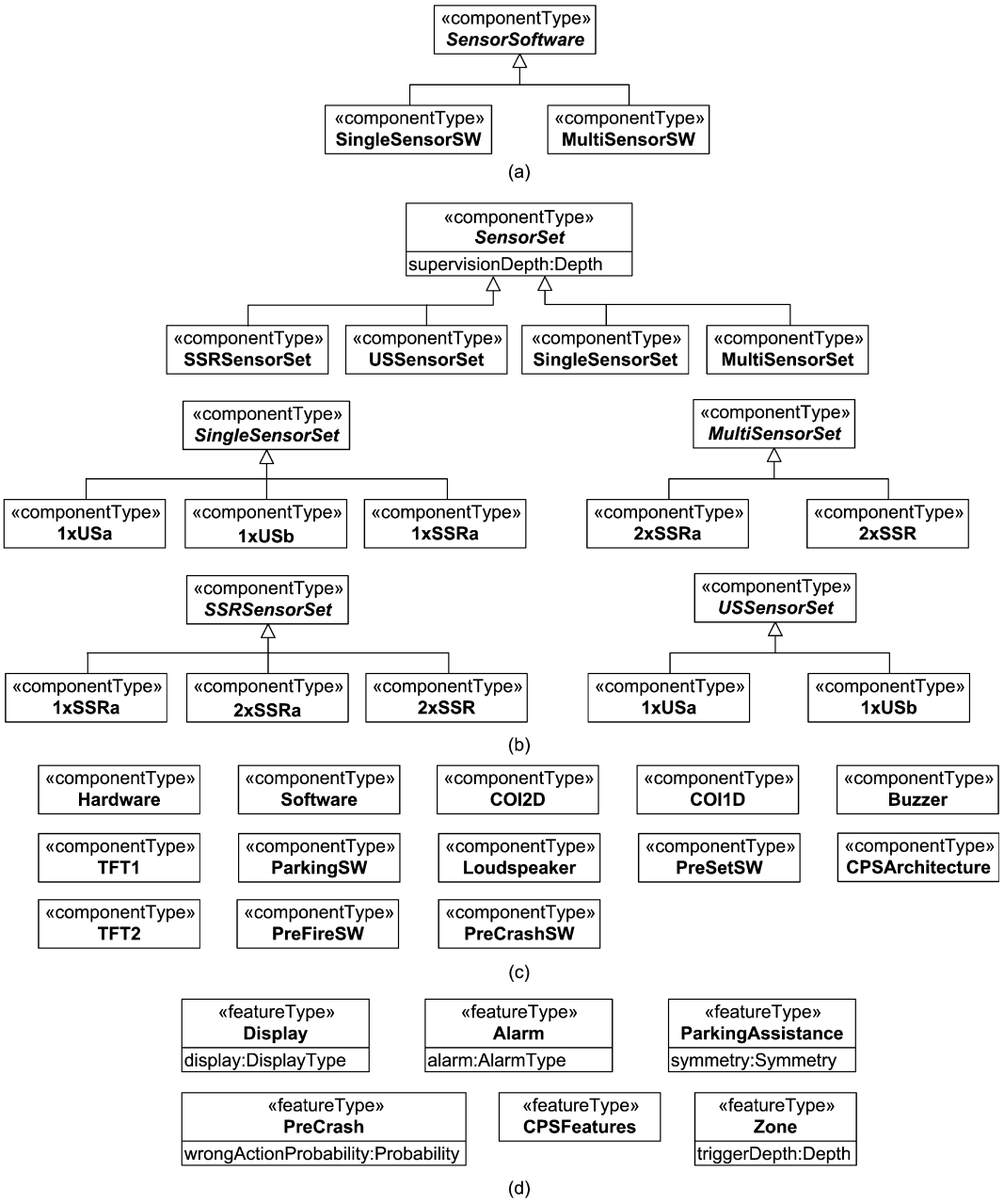

Fig. 5. The Kumbang types in the sample model (Fig. 4) represented using UML. (a) The taxonomy of types representing sensor software. (b) The taxonomy of types representing sensor sets. (c) Other component types, i.e., types not participating in taxonomic relations. (d) Feature types.

crete component types representing physical sensor sets may have multiple supertypes: each of the concrete sensor sets is a subtype of either the component type *SingleSensorSet* representing a sensor set consisting of a single sensor, or of *MultiSensorSet*, consisting of multiple sensors, and of either component type *SSRSensorSet* or *USSensorSet*. The remaining component types are illustrated in Fig. 5c and feature types in Fig. 5d.

3.3. Attributes

A Kumbang instance is characterised by its *attributes*. An attribute is a name–value pair. In a Kumbang type, *attribute definitions* are used to specify the attributes their instances may and must have. An attribute definition includes a *name*, and an *attribute type*. An attribute type specifies a non-empty set of possible attribute values. Possible attribute value type kinds include *enumerations*, *strings*, *integers*, *floating point numbers*, and structured types built from these.

Semantics. A valid instance of a Kumbang type has the attributes of its type. In more detail, a valid instance i of type t has, for each attribute definition a of t, an attribute with name *a.name* and value that is one of the possible values defined by *a.type*. The attribute definitions of t include those defined in the type and those inherited from its supertypes; more generally, the properties of a type include those defined in its supertypes. A valid

Kumbang instance has no other attributes than those of its type.

UML. In UML, attribute definitions are represented as properties. In more detail, attribute definitions are represented using the stereotype *AttributeDefinition* that extends the metaclass *Property (from Kernel)*. A number of constraints apply: each property must have a type. Further, the type must have the stereotype *Attribute type* applied to it; hence, as *AttributeType* is an extension of *DataType* in the UML metamodel, only types the instances of which are identified by their values may act as attribute types in Kumbang [15].

Example. The running example includes several attribute definitions. For example, the component type *SensorSet* includes an attribute named *supervisionDepth* of type *Depth*; the type is an enumerated set of distances given in centimetres. Intuitively, the attribute represents the supervision depth of the sensor set, that is, what is the maximum distance for objects that can be supervised by the sensor set. In addition, the model contains binary attribute types *Symmetry*, *AlarmType*, and *DisplayType*, and attribute type *Probability* with three possible values. Each of these is used in a feature type to represent information some information. For instance, the attribute definition *symmetry* of type *Symmetry* in feature type *ParkingAssistance* is used to represent whether parking assistance is provided for the passenger side only (value *PassengerSideOnly*) or for both sides of the vehicle (*LeftAndRight*).

3.4. Compositional structure

In this subsection, we discuss concepts for describing the structure of software product families and their individuals. We use the term structure to refer to how instances are composed of other instances of the same kind.

A *composable type* can be defined a compositional structure. Composable type is an abstract metaclass that is a subclass of Kumbang type. Composable type has two concrete subclasses, *component type* and *feature type*, see Fig. 3b. A composable type intentionally specifies the properties of its instances.

A component type represents a distinguishable entity in a software product family. Such an entity may be, for instance, a software component, or, in the case of an embedded system, a hardware component. *Components*, the instances of component types, represent software components constituting individual software systems in the software product family.

A feature type defines the properties of its instances, i.e., *features*. A feature is an end-user visible characteristic of a system [5]. Hence, the features a product individual delivers characterise the individual from the user's or sales person's point of view.

At most one component type can be defined to be the *root component type*, and at most one feature type the *root feature type*. A Kumbang model must define at least one of the root types. This implies that a Kumbang model may describe a software product family from either an architectural or a feature point of view, or from both.

The compositional structure of a composable type specifies how components and features are either physically or logically composed of other components and features, respectively. We say that composable instances may have other composable instances as their *parts*. When discussing features, we also use the term *subfeature* to refer to their parts.

A composable type specifies its compositional structure through a set of *part definitions*. A part definition consists of an optional *part name*, a non-empty *set of possible part types*, a *cardinality*, and *similarity definition*. We use the term *whole type* to refer to the composable type containing the part definition.

The part name identifies the role in which composable instances are parts of the individuals of the whole type. The set of possible part types contains the composable types the instances of which may occur as parts of the instances of the whole type with the part name. The set may only contain component types if the whole type is a component, and feature types if the whole type is a feature type. The cardinality consists of a lower and upper bound. The lower bound is a non-negative integer, and the upper bound an integer greater than or equal to one. The upper bound must be greater than or equal to the lower bound. Finally, the similarity definition takes one of the values *same*, *different*, and *none*; the definition specifies whether the parts must be of the same type (*same*), must be of different types (*different*), or there are no restrictions on their respective types (*none*).

Semantics. In a valid configuration, there must exist exactly one instance of the root component type, if one is defined. We call this instance the *root component instance*. Similarly for the root feature type, if one is defined, there must be exactly one instance of it (*root feature instance*) in a valid configuration. All other composable instances must be a part of some other composable instance. Hence, all the component instances in a valid configuration are transitive parts of the root component instance, and all feature instances of the root feature instance. The types of all instances in a valid configuration must be concrete.

A valid instance of a composable type has the parts specified by its part definitions. In more detail, an instance i is valid if (and only if) it has, for each part definition p of its type t, n composable instances as its parts with the part name $p.name$. The number n must be between the lower and upper bounds of the cardinality. Each part must be of one of the possible part types, $p.types$. Further, if the similarity definition has value *same*, all the part instances must be of same type; if the similarity definition has value *different*, the part instances must all be instances of different types. A valid composable instance has no other parts than those implied by the part definitions of its type.

UML. In the Kumbang profile, *ComposableType* is an abstract stereotype that specialises *KumbangType*; see

Fig. 3b. *ComponentType* and *FeatureType* are concrete stereotypes that specialise *ComposableType*. The stereotype *ComposableType* extends *StructuredClassifier (from InternalStructures)*. In addition, *ComponentType* extends *Component (from components)*. No explicit extension is given for *FeatureType*: the stereotype inherits the extension from its superclass *ComposableType*.

Part definitions are represented by the *PartDefinition* stereotype. The stereotype is defined two properties: *types* of type *ComposableType* contains the set of possible part types and the enumerated value *similarity* encodes the similarity definition. The cardinality of the *types* property is 1..*, implying that the set of possible part types must be non-empty.

Example. The structure of the example software product family is illustrated in Fig. 6. In more detail, Fig. 6a illustrates the featural structure: a car periphery system (CPS), represented by the root feature type, *CPSFeatures*, is composed of two applications: pre-crash detection and parking assistance, represented by the feature type *PreCrash* and *ParkingAssistance*. The two applications are represented by the part definition *application* with cardinality 1..2 and the two above-mentioned types as the possible part types. The fact that the similarity definition has the value *different* captures that fact that the same application cannot be meaningfully delivered twice by the same system.

There are further part definitions in the model. The feature type *ParkingAssistance* includes five part definitions: *imminent*, *veryNear*, *near*, *inProximity*, and *userInterface*. The first four represent the different zones that the parking assistance system monitors. The cardinality 0..1 of *inproximity* implies that this zone is optional (may or may not be included in a parking assistance application), whereas the other zones are mandatory (must be included). The fifth part definition, *userInterface*, represents the user interfaces of the application: the interface may be either an alarming device, a display, or both.

On the component side, the root component type Fig. 6b defines two parts, *hardware* of type *Hardware* and *software* of type *Software*. As the names imply, these represent the hardware and software, respectively, constituting the system. Both the software and hardware are further decomposed. The decomposition of hardware is

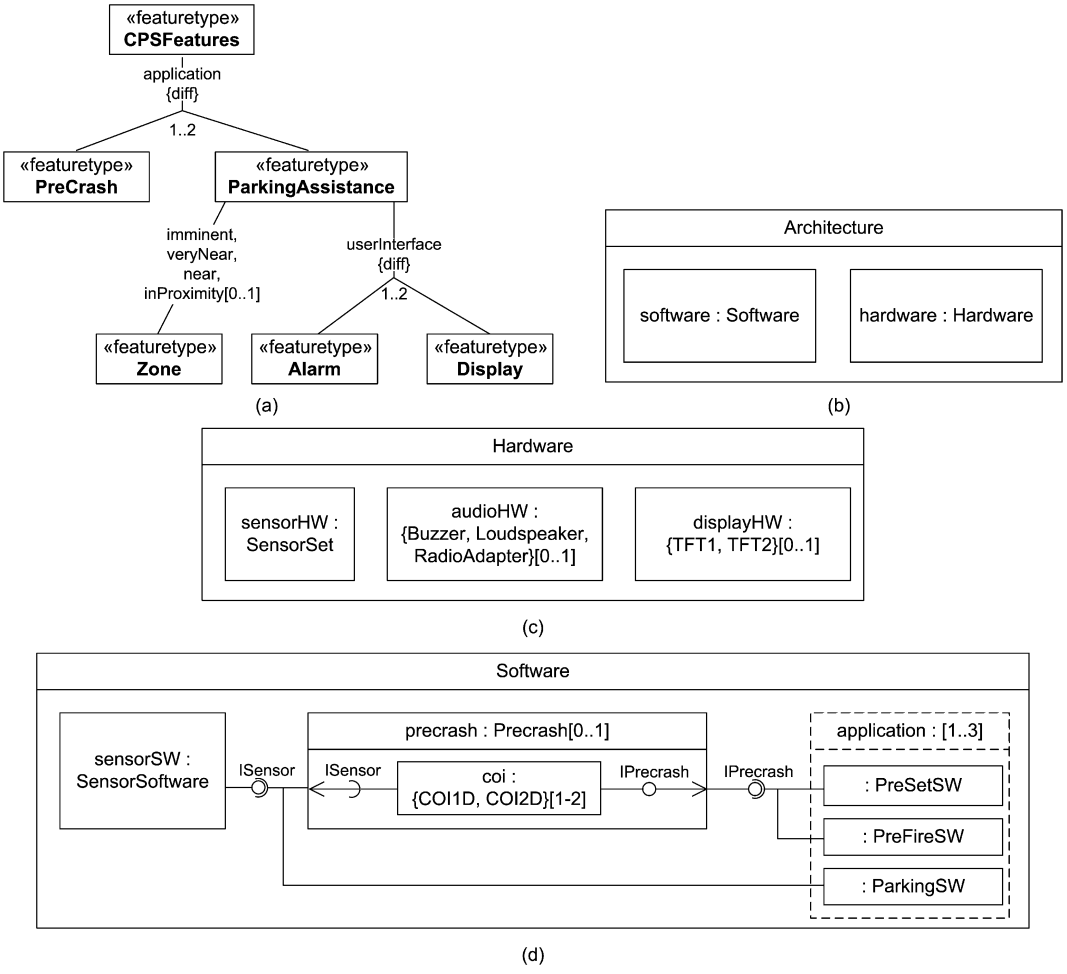

Fig. 6. The compositional structure of the running example. (a) The compositional structure of feature types. (b) The architecture consists of the software and hardware architectures. (c) The hardware architecture illustrated in more detail. (d) The architecture of the software components, i.e., their compositional structure, interfaces and connections between interfaces.

represented in Fig. 6c: the hardware consists of a sensor set (part definition named *sensorHW*) of type *SensorSet*, optional audio hardware (*audioHW*) of type *Buzzer*, *Loudspeaker*, or *RadioAdapter*, and an optional display (*displayHW*) either of type *TFT1* or *TFT2*. The software, illustrated in Fig. 6c, consists of sensor software (*sensors*), optional pre-crash detection software (*precrash*), and one to three different applications (*application*).

3.5. Interfaces and connections

Components, especially software ones, are often characterised by the services they require from the environment in which they are deployed, and the services they provide themselves to their environment. *Interfaces* are a means to describe such services. In this subsection, we discuss the concepts and constructs in Kumbang for describing the interfaces of components, and the connections between such interfaces.

An *interface type* is a description of a set of *functions*. A function corresponds to a function signature in a programming language, such as the C programming language. Intuitively, an interface type describes a set of services. Interface type is a subclass of the metaclass Kumbang type. The instances of interface types, *interface instances*, are connection points of component instances.

A component type specifies the required and provided services of its instances via *interface definitions*. An interface definition consists of an optional *interface name*, a non-empty *set of possible interface types*, a *direction definition*, an *optionality definition*, and a *groundedness definition*. The interface name specifies the role in which the interfaces instances are interfaces of the component type. The set of possible interface types contains the interface types the instances of which may occur as interfaces of instances of the component type by the interface name.

The direction definition has two possible values: *provided* and *required*, with the intuitive semantics that interface instances denote services either provided or required by the component, based on the value in the definition corresponding to the interface.

The optionality definition has two possible values: *optional* and *mandatory*: the value mandatory implies that the instances of the component type must have an interface with the interface name, whereas the value optional implies that the instances may but need not have such an interface. Finally, the groundedness definition is a Boolean value that specifies whether an instance of the component implements the functions and attributes defined by the interface type (*true*), or whether the responsibility is delegated to its parts (*false*). It should be noted that the groundedness definition is meaningful only for provided interfaces.

There may be a *connection* between a pair of interface instances. The fact that the pair of interface instances (a, b) is connected is conceptualised by the pair being in the *connected* relation. The interface instance is the first position is termed the *source* interface, and the interface in the second position the *target* interface. The *connected* relation is not, by default, symmetric: *connected*(a, b) does not imply *connected*(b, a). The intuitive semantics of connections is that requests, such as function calls, flow from the source interface to the target interface. In other words, the target interface provides the services described by its type to the source interface.

Not all pairs of interface instances may be reasonably connected. Which interfaces instances may be connected, is determined by their directions, types, and the relative locations of the component instances containing them. A valid configuration may be required to contain certain connections between pairs of interface instances. The specific rules that determine which connections are possible and required in a valid configuration depend on the component model assumed. One set of such rules is based on the ones used in Koala [37]; other rules are likewise possible. In the following, we assume that such a set of rules has been defined, but do not commit to any specific set of such rules.

A component type may include a number of *connection constraints*. A connection constraint is a condition requiring that a connection between a pair of interfaces exists. A connection constraint is a special case of the *constraint* concepts, and will be discussed in more detail in the following subsection.

Semantics. A valid component instance has the interfaces defined by its type. In more detail, a valid component instance c has, for each interface definition d where the optionality definition has value *mandatory*, an interface instance i as its interface with name *d.name*. If the optionality definition has value *optional*, the component instance may, but is does not need to have such an interface. The interface instance is either a required or a provided interface, according to the direction definition. The interface is either grounded or not according to the groundedness definition. A valid component instance has no other interfaces than those described by its interface definitions.

The connections between pairs of interfaces must obey the set of rules for connections.

UML. Interface type is represented by the stereotype *InterfaceType* that extends the metaclass *Interface (from Interfaces)*. An instance of the stereotype, i.e., an interface type, may only own operations. Similarly, an interface definition is represented by the stereotype *InterfaceDefinition* that extends the metaclass *Port* (*from Ports*). *InterfaceDefiniton* defines three properties: *types* of type *InterfaceType* and cardinality 1..* is the set of possible interface types; if there is only one possible interface type, this is stored in the *type* property of the metaclass *Property*. The Boolean value *isOptional* corresponds to the optionality definition; and *direction* of the enumerated value type *Direction* that has two possible values, *required* and *provided*. The groundedness definition corresponds to the *isBehavior* attribute of the metaclass *Port (from ports)*.

An interface instance is represented by the stereotype *InterfaceInstance* that extends the metaclass *InstanceSpeci-*

fication (from Kernel). The stereotype defines two properties: *direction* and *isBehavior*. Both of these are derived attributes: the values are derived from the values of the interface definitions of the interface instances.

A connection is represented as a stereotype extending the metaclass *InstanceSpecification (from Kernel)*. In more detail, connections are represented by links between instance specifications corresponding to interface instances. The links are instances of the *targets* association that is derived based on the rules for connecting interfaces.

Example. The running example includes two interface types, *ISensor* and *IPrecrash*. The sensor software (*sensors*) provides an interface named of type *ISensor* that represents raw data from sensors, Fig. 6d. This data is refined by the pre-crash software (*precrash*): the component collects the sensor data through a required interface of type *ISensor* and refined provides the refined data for other components to use through a provided interface of the refined type, *IPrecrash*. Applications represented by component types *PreSetSW* and *PreFireSW* use the refined data and the parking software, represented by component type *ParkingSW* the raw data; the usage of data is represented by required interfaces.

3.6. Constraints

The concepts and constructs discussed so far form the basis for modelling variability in software product families. However, there may exist interdependencies between entities that cannot be adequately captured using the concepts and constructs discussed so far. Hence, mechanisms for specifying such dependencies are needed. In this subsection, we describe how constraints fulfil this need.

A *constraint* is a Boolean condition. It must be possible to evaluate a constraint with respect to a configuration. Further, each constraint must be true in a valid configuration. Kumbang does not confine to any specific constraint language: any constraint language can be used as long as it can evaluated with respect to a configuration. However, we have specified the *Kumbang Constraint Language* as an example of a constraint language. We discuss this language in the remainder of this subsection.

In the Kumbang Constraint Language, it is possible to make references in the compositional hierarchy. Such references are termed *part references* and consist of a sequence of part names, optionally qualified by type names. A part reference is evaluated in the context of a composable instance, and the semantics of a part reference is the set of composable instances that match the sequence of part names and type qualifiers (if present): a part matches a part name if the name equals its role, and the type if the part is an instance of the type. It is also possible to make references to interfaces by adding an interface name to the end of a part reference. Also interface names may be qualified with interface types.

Example. The constraint in component type *CPSArchitecture* includes two part references, spelled out as *software.sensorSW* and *hardware.sensorHW*, see Fig. 4. In feature type *CPSFeatures*, the part reference *application:ParkingAssistance* refers to those parts of *CPS* instances named *application* that are of type *ParkingAssistance*.

The *connects* section of component type *Software* contains interface references. As an example, the reference *application:PreSetSW->crashinput* refers to the interface named *crashinput* of a part named *application* that is of type *PreSetSW*.

The Kumbang constraint language defines a number of predicates and functions on part and interface references and Kumbang types. Table 1 contains a summary of these.

Part references are set-valued, i.e., they refer to sets of instances. This is due to the fact that a composable instance may have multiple parts in the same role. When predicates are applied to set-valued referenced, the semantics is existentially quantified in the sense that the predicate evaluates to true if it is true for at least one of the instances in the set, or any pair of instances in the case of *connectedTo* predicate, see Table 1 for details. It is also possible to explicitly quantify over part references using the standard universal ($\forall$) and existential ($\exists$) quantifiers.

Standard arithmetic operators ($+$, $-$, $*$, /, mod) can be applied to arithmetic values, i.e., real-valued attributes and cardinalities, to yield new values. Further, such values can be compared using the customary comparison

Table 1
The predicates and functions in Kumbang Constraint Language and their semantics. The Kumbang Constraint Language is a constraint language that can be used to capture complex dependencies between entities in Kumbang models that cannot be captured using other construct

Predicate	Semantics
cardinality(*ref*)	The number of instances referenced by *ref*
value(*ref*, *attr*)	The set of values of attribute named *attr* of instances referenced by *ref*
present(*ref*)	True if an instance referenced by *ref* is in the configuration
instanceOf(*ref*, *T*)	True if an instance referenced by *ref* is of type *T*
hasInstances(*T*)	True if the type *T* has instances (at least one) in the configuration
hasPartOfType(*T*)	True if the instance has an instance of type *T* as its transitive part
hasPartOfType(*T*, *ref*)	True if an instance referenced by *ref* has a transitive part of type *T*
connected(*ref*)	True if an interface instance referenced by *ref* is connected
connectedTo(ref_1, ref_2)	True if an interface instance referenced by ref_1 is connected to an interface instance referenced by ref_2

operators ($=, \neq, <, \leqslant, >, \geqslant$). Other attribute values can be compared for equality and inequality ($=, \neq$). The results of the comparisons are Boolean values. These can be further used as operands for the standard logical operators, the unary not ($\neg$), and the binary operators: or ($\vee$), and ($\wedge$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$).

An important class of constraints are *connection constraints*. Connection constraints are a special case of constraints that can be captured using the *connectedTo*-predicate. However, a special syntax for them is used due to their importance.

Example. In the running example, connection constraints are defined in component type *Software*, in the *connects* section. For example, the constraint *connectedTo(application: ParkingSW->sensorInput, sensorSW->sdata)* is written as *application:ParkingSW->sensorInput = sensorSW->sdata*. The intuitive semantics is that whenever the parking assistance software is included in the system, its required interface (*sensorInput*) must be connected to the sensor date source (interface *sdata* of *sensors*).

Constraints can also be used to specify the dependencies between features and components. We say that features instances are *implemented by* entities in the component hierarchy. Constraints used to specify such dependencies are termed *implementation constraints*. In such constraints, the dollar symbol ($) can be used to refer to the root component instance.

Example. There are a number of implementation constraints in the example model. As an example, feature type *ParkingAssistance* includes the implementation constraint *present($.software.parkingSW)* that has the semantics that to implement parking assistance, the root software component must have a part referenced by *software.parkingSW* present.

Semantics. A valid Kumbang instance satisfies the constraints of its type.

4. Proof of concept

In this section, we discuss the proof of concept existing for Kumbang. This includes a translation from Kumbang to Weight Constraint Rule language, a general-purpose knowledge representation language; *Kumbang Configurator*, a prototype tool that can be used to resolve variability represented using Kumbang models; and a number of case studies inspired by real-world software product families in which Kumbang has been used.

We have provided Kumbang with formal semantics by defining a translation from Kumbang models to Weight Constraint Rule Language (WCRL) [49], a general-purpose knowledge representation language. Although general-purpose, WCRL has been designed to allow the easy representation of configuration knowledge about non-software products. Furthermore, it has been shown to be suitable for representing configuration modelling concepts [50]. This suggests that WCRL is a reasonable choice for the knowledge representation formalism of our approach as well. Further, an inference system *smodels* (see http://www.tcs.tkk.fi/smodels/) 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 [49].

We have defined a language based on the Kumbang domain ontology. The language is likewise called Kumbang. The language has been provided with a machine-readable syntax for the language using javaCC (Java Compiler Constructor) [51], a tool that generates Java code for both lexical and syntax analysis based on an input file describing the tokens and syntax of a language. The machine-readable syntax closely resembles that used in Fig. 4, although minor changes have been made in this paper to save space in the figure. In short, the language consists of a number of type definitions, each of which contains the properties defined by the type. Hence, the syntax resembles that of an object-oriented programming language. We have implemented the translation from Kumbang models expressed in the language to WCRL.

A configuration tool supporting Kumbang has been implemented in our research group; Fig. 7 shows a screenshot. The tool is called Kumbang Configurator [52]. Kumbang Configurator supports the user in the configuration task as follows. The tool reads in a Kumbang model and represents the variability in the model in a graphical user interface. 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, to select attribute values or the type of a given part, create a connection between interfaces, etc. 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 performed using *smodels* based on 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. The current implementation has the limitation that only enumerated attribute value types are supported. Further details about Kumbang Configurator can be found in [52].

Our research group has modelled a number of example products using Kumbang: in addition to the running example used in this paper, see Section 3 above, a weather prediction network loosely based on a real product has been modelled from the architectural point of view. The models are roughly equal in size, containing dozens of model elements (types and their properties). In both cases, Kumbang has provided a sufficient level of support to capture the intent of the product families, i.e., the models capture

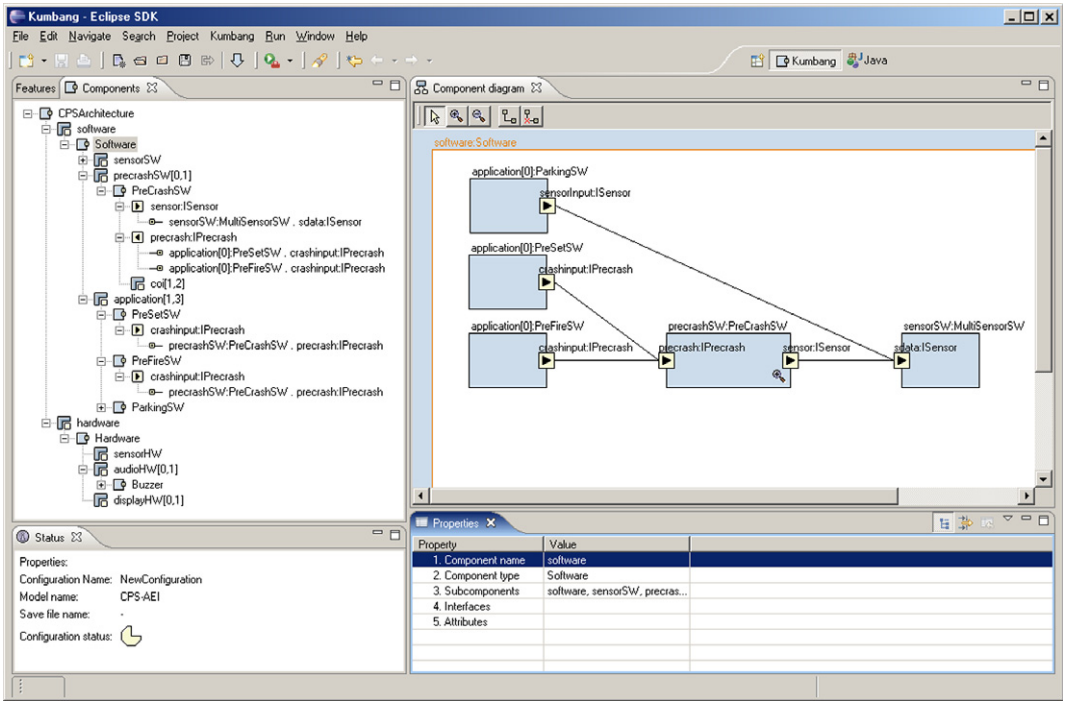

Fig. 7. A screenshot from Kumbang Configurator, a tool supporting the configuration task. The configurator takes a Kumbang model as an input and allows the user to enter his requirements for a specific product using a graphical interface. The screenshot illustrates the running example being configured.

correctly the intuition of what are the valid configurations in the families and rules out configurations that are intuitively invalid. The cognitive effort required to create the models has been reasonable. The translation from Kumbang language to WCRL could be done within a couple of seconds.

5. Comparison to related work

In this section, we compare Kumbang to related work. The comparison is organised according to the main classes modelling concepts of Kumbang. The main related work includes feature modelling methods, methods for modelling product family architectures, the Unified Modeling Language (UML), and configuration ontologies.

The comparison is made to UML version 2.0. There are several reasons for selecting this language version. An obvious reason is that it is the current official version of the language. In addition, UML 2.0 contains a number of concepts and constructs useful for representing compositional structures and architectures that are not found in previous language version. These include improved facilities for specifying the structure of components and other classifiers, and the concept of connector used to represent connections between interfaces in components.

5.1. Types and taxonomy

Not all architecture description languages (ADLs) distinguish between types and instances. However, the distinction is made in Koala [37] and in xADL 2.0 [40]. Due to the practical success of Koala [39], we believe that distinguishing between component instances is useful. Also, in UML components may have instances. The same distinction is also made in the product configuration domain [11,12].

Traditionally, feature modelling methods have included only the notion of features: no distinction between feature types and instances has been made. However, at least the authors of [31] have identified a phenomenon in which "a configuration may include several different variants of the same feature". We believe this is an important phenomenon and therefore needs to be explicitly modelled. Further, we believe it is natural to model this phenomenon using feature types and instances: this enables a distinction between feature types appearing in feature models and feature instances appearing in configurations.

Another benefit of using types is that they facilitate the reuse of knowledge. Once a type has been defined, it is possible to refer to it in a simple manner, instead of repeating the entire definition. The possibility of organising types into taxonomies and inheriting property definitions further promote reuse of feature knowledge.

Example. In the running example, the classification of sensor sets based on the number of sensors (single vs. multiple), and type (US vs. SSR) is modelled using subtyping.

5.2. Attributes

Attributes are one of the fundamental modelling constructs found in almost all software modelling methods. For instance, they are found in almost all ADLs [36], and in UML. Hence, it is natural that component types can

define attributes in Kumbang. Attributes are also used in feature modelling methods: although not included in FODA [5] and argued against in [6], they have more recently been found to be useful concept for modelling features [31].

Some kinds of attribute types are problematic from the point of view of formalising Kumbang models and reasoning about them using WCRL: enumerated values is the easiest kind of attribute types, and these are the kind of attributes that are supported in the current proof of concept. Comparisons between attributes of string type for equality and inequality can be implemented using WCRL in a straightforward manner. The same applies to structured attribute value types containing enumerated values and strings as fields. However, arithmetic operations including floating point values would require adopting a new knowledge representation language and inference tool.

5.3. Compositional structure

Part definitions in Kumbang distinguish between the role (defined by the part name) and type (defined by the set of possible part types). The distinction is also reflected in the instance world.

Many methods for modelling components distinguish between roles and types. The distinction is made in UML, where parts may be specified both a type and a name. In Koala, the contained components are defined both a name and a type. The same holds also for configuration ontologies [11,12].

However, in feature modelling methods there is typically no distinction between a feature and the role in which the feature is a subfeature of a whole feature. A number of arguments speak for introducing the distinction. First, given the distinction between feature types and their instances, it is natural that a feature type can be used in multiple locations of the compositional hierarchy of features, possible in different roles. Second, instances of a single feature type may appear as subfeatures of a single feature type in different roles.

Example. In feature type *ParkingAssistance*, the feature type *Zone* is the possible type of four part definitions, namely *imminent*, *veryNear*, *near*, and *inProximity*.

The possibility of defining multiple possible part types roughly corresponds to alternative and or-features, as defined in feature modelling methods, such as [31]: an alternative feature corresponds to a part definition with a possible part types corresponding to the alternative features and cardinality of one. Further, an or-feature can be represented similarly, with the exception that cardinality is now 1..*n*, where *n* is the number of possible part types. However, in UML 2.0 or in Koala there is no construct similar to multiple possible types. Hence, the *PartDefinition* stereotype in the Kumbang profile had to be extended with the property *types* in order to represent multiple possible part types. In the product configuration domain, the construct of multiple possible part types is in wide use.

Cardinality of parts or subfeatures is a construct present in UML 2.0, some feature modelling methods [31], and configuration ontologies. Hence, its inclusion to Kumbang is well motivated.

5.4. Interfaces and connections

Traditionally, points of interactions and connections between them are an important concept in ADLs, such as Koala and xADL 2.0. The term most commonly used for points of interaction in ADLs is *port*, and for connections between them *connector*. Ports have also been considered in configuration ontologies as connection points of components. Such points of interaction have not been defined for features. Consequently, Kumbang defines interaction points only for components, not for features.

The advocates of ADLs have traditionally emphasised the importance of connectors in architecture description [36]: it has been argued that they should be "first class entities", equal in importance to components and ports. However, in Koala connectors are not given much emphasis: unlike in many ADLs, connectors are typically merely connections between connections points with no internal structure; there are no connector types in either of these languages, nor are there connector instances in UML. The practical success of Koala and the fact that UML is the de facto standard modelling method in software engineering domain motivated the connection concept in Kumbang to be similar to these two languages.

5.5. Constraints

Constraints have been found an important modelling concept in the product configuration domain [11]. Also, feature modelling methods typically include the possibility to state at least two forms of constraints: for features *A* and *B*, it is possible to state that *A* and *B* are *incompatible*, or that *A requires B*. However, it has been argued that this form of constraints is not sufficient [53]. As an example, there is no straightforward way to state that *A* and *B* together require *C*. Further, there may exist even more complex interactions between features [54]. Also, with models describing variability both from a feature and an architecture point of view, it is not feasible to assume that there would be no interactions between entities in these views. Hence, we believe a comprehensive constraint language is needed to capture the possibly complex interactions between different entities.

6. Conclusions and further work

We have presented Kumbang, a domain ontology for modelling the variability in software product families.

The ontology synthesises existing variability modelling methods, predominantly feature and architecture based ones. It incorporates modelling constructs developed in the product configuration domain. The proof of concept shows that the semantics is rigorous enough to serve as the basis for representing Kumbang models using Weight Constraint Rule Language (WCRL) [49], a general purpose knowledge-representation language with formal semantics. Therefore, Kumbang serves as a solid basis for developing modelling languages and tool support for the tasks of managing variability in software product families and configuring them to meet specific customer requirements.

There are a number of possible ways to extend Kumbang. One such extension would be to extend Kumbang to support modelling the evolution of software product families: many software product families have long life-spans during which new component types and versions of types are introduced [55]. Also, in some cases it is desirably to configure a software product family at multiple stages [56]: at each stage, some decisions are made while others are deferred till later stages. Extending Kumbang to support such a configuration process is an interesting research problem.

In addition to conceptual extensions, Kumbang can be extended in other directions. One of the directions is modelling languages based on Kumbang. In this paper we have used one such language to present the example; other languages, both textual and graphical, would be likewise feasible. In designing such a language, minimising the cognitive effort required to create Kumbang models should be an important design criteria.

Still further towards applications, the tool support for Kumbang should be further developed. A prototype of a configuration tool exists and needs to be further developed. The fact that Kumbang is described as a UML profile suggests that existing UML tools could be used to create Kumbang models. However, this is not quite straightforward, as most such tools seem to support the standard less than perfectly and also the standard itself is not quite unambiguous.

Finally, demonstrating the practical applicability of Kumbang requires modelling real software product families in real software development contexts. Both the expressive power and usability of Kumbang and the language built on it should be evaluated. Similarly, applying Kumbang to a sufficiently wide range of different kinds of configurable software product families can be used to analyse its scope of applicability.

Acknowledgements

We gratefully acknowledge the financial support from Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Finnish Funding Agency for Technology and Innovation (Tekes). We thank Varvana Myllärniemi, Pyry Lahti, Janne Lemmetti, and Markus Pitkäranta for implementing Kumbang Configurator and participating in creating the example model used in the paper.

References

[1] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software - Practice and Experience 35 (8) (2006) 705–754.
[2] J. Bosch, Software variability management (introduction to special issue on software variability management), Science of Computer Programming 53 (5) (2004) 255–258.
[3] P.C. Clements, L. Northrop, Software Product Lines – Practices and Patterns, Addison-Wesley, Boston, MA, 2001.
[4] J. Bosch, Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach, Addison-Wesley, Boston, MA, 2000.
[5] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, S.A. Peterson. Feature-oriented domain analysis (FODA) – feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
[6] K. Czarnecki, U.W. Eisenecker, Generative Programming, Addison-Wesley, Boston, MA, 2000.
[7] T. Asikainen, T. Soininen, T. Männistö, A Koala-based approach for modelling and deploying configurable software product families, in: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5), 2003, pp. 225–249.
[8] E. Dashofy, A. van der Hoek, R.M. Taylor, A comprehensive approach for the development of modular software architecture description languages, ACM Transactions on Software Engineering and Methodology 14 (2) (2005) 199–245.
[9] T. Soininen, M. Stumptner, Introduction to special issue on configuration, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17 (1–2) (2003) 1–2.
[10] J.S. Gero, Design prototype: a knowledge representation schema for design, AI Magazine 11 (4) (1990) 26–36.
[11] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, Towards a general ontology of configuration, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (4) (1998) 357–372.
[12] A. Felfernig, G. Friedrich, D. Jannach, Conceptual modeling for configuration of mass-customizable products, Artificial Intelligence in Engineering 15 (2) (2001) 165–176.
[13] B. Faltings, E.C. Freuder, Special issue on configuration, IEEE Intelligent Systems 14 (4) (1998) 29–85.
[14] T. Darr, M. Klein, D.L. McGuinness, Special issue on configuration design, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (4) (1998) 293–397.
[15] Unified Modeling Language: Superstructure, version 2.0. Technical Report Formal/05-07-04, Object Management Group (OMG), 2005.
[16] D. Weiss, C.T.R. Lai, Software Product Line Engineering: A Family Based Software Development Process, Addison-Wesley, Boston, MA, 1999.
[17] P.C. Clements, Northrop L. Salion, Inc.: A Software Product Line Case Study. Technical Report CMU/SEI-2002-TR-038, Carnegie Mellon University, 2002.
[18] M. Raatikainen, T. Soininen, T. Männistö, A. Mattila, Characterizing configurable software product families and their derivation, Software Process: Improvement and Practices 10 (1) (2005) 41–60.
[19] M. Raatikainen, T. Soininen, T. Männistö, A. Mattila, A case study of two configurable software product families, in: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5), 2004, pp. 403–421.
[20] J. van Gurp, J. Bosch, Proceedings of Software Variability Management Workshop, Technical Report IWI Preprint 2003-7-01, University of Groningen, 2003.

[21] J. Bosch, P. Knauber, Proceedings of International Workshop on Software Variability Management (SVM) (held in conjunction with ICSE 2003), Portland, Oregon, USA, 2003.
[22] T. Männistö, J. Bosch, Proceedings of software variability management for product derivation – towards tool support, a workshop in SPLC 2004, Technical Report HUT-SoberIT-C6. 2004.
[23] K. Schmid, I. John, A customizable approach to full lifecycle variability management, Science of Computer Programming 53 (3) (2004) 259–284.
[24] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J.H. Obbink, K. Pohl, Variability issues in software product lines. in: Proceeding of the 4th International Workshop on Product Family Engineering (PFE-4), 2001.
[25] T. Asikainen, Modelling methods for managing variability of configurable software product families, Licentiate thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2004.
[26] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, FORM: a feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering 5 (1998) 143–168.
[27] M. Griss, J. Favaro, M. d'Alessandro, Integrating feature modelling with the RSEB. in: Proceedings of the Fifth International Conference on Software Reuse, 1998, pp. 76–85.
[28] K. Lee, K.C. Kang, J. Lee, Concepts and guidelines of feature modeling for product line software engineering, in: Proceedings of the 7th International Conference on Software Reuse, 2002, pp. 62–77.
[29] T. von der Maßen, H. Lichter, RequiLine: a requirements engineering tool for software product lines, in: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5), 2004, pp. 168–180.
[30] D. Fey, R. Fajta, A. Boros, Feature modeling: a meta-model to enhance usability and usefulness, in: Proceedings of Second International Software Product Line Conference (SPLC2), 2002, pp. 198–216.
[31] K. Czarnecki, T. Bednasch, P. Unger, U.W. Eisenecker, Generative programming for embedded software: an industrial experience report, ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering 2002, pp. 156–172.
[32] S. Staub-French, M. Fischer, J. Kunz, K. Ishii, A feature ontology to support construction cost estimation, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17 (2) (2003) 133–154.
[33] D. Garlan, Software architecture, in: J.J. Marciniak (Ed.), Encyclopedia of Software Engineering, John Wiley & Sons, New York, 2001.
[34] L. Bass, P.C. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, Boston, MA, 1999.
[35] N. Medvidovic, R.M. Taylor, Separating fact from fiction in software architecture, Third International Software Architecture Workshop (ISAW-3) in conjunction with SIFSOFT'98 (FSE-6), 1998.
[36] N. Medvidovic, R.M. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.
[37] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The Koala component model for consumer electronics software, IEEE Computer 33 (3) (2000) 78–85.
[38] R. van Ommering, Building Product Populations with Software Components. Doctoral dissertation, University of Groningen, The Netherlands, 2004.
[39] R. van Ommering, Building product populations with software components. in: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), 2002, pp. 255–265.
[40] E. Dashofy, A. van der Hoek, R.M. Taylor, An infrastructure for the rapid development of XML-based architecture description languages, in: Proceedings of the ICSE 2002 International Conference on Software Engineering (ICSE 2002), 2002.
[41] A. van der Hoek, Design-time product line architectures for any-time variability, Science of Computer Programming 53 (3) (2004) 285–304.
[42] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, M. Stumptner, Configuring large systems using generative constraint satisfaction, IEEE Intelligent Systems 13 (4) (1998) 59–68.
[43] D.L. McGuinness, J.R. Wright, An industrial-strength description logic-based configurator platform, IEEE Intelligent Systems 14 (4) (1998) 69–77.
[44] A. Haag, Sales configuration in business processes, IEEE Intelligent Systems 13 (4) (1998) 78–85.
[45] B. Yu, J. Skovgaard, A configuration tool to increase product competitiveness, IEEE Intelligent Systems 13 (4) (1998) 34–41.
[46] S. Thiel, S. Ferber, T. Fischer, A. Hein, M. Schlick, A case study in applying a product line approach for car periphery supervision systems, in: Proceedings of In-Vehicle Software 2001 (SP-1587), 2001, pp. 43–55.
[47] S. Thiel, A. Hein, Modeling and using product line variability in automotive systems, IEEE Software 19 (4) (2002) 66–72.
[48] L. Hotz, T. Krebs, K. Wolter, Combining Software Product Lines and Structure-Based Configuration – Methods and Experiences, http://www.soberit.tkk.fi/SPLC%2DWS/Presentations/MacGregor-Presentation%20ConIPF_At_SPLC.pdf (last accessed 2006).
[49] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (1–2) (2002) 181–234.
[50] T. Soininen, I. Niemelä, J. Tiihonen, R. Sulonen, Representing configuration knowledge with weight constraint rules, in: Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, 2001.
[51] javaCC home page, https://javacc.dev.java.net/ (last accessed 2004).
[52] V. Myllärniemi, Kumbang Configurator – A Tool for Configuring Software Product Families, Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2004.
[53] D. Batory, Feature models, grammars, and propositional formulas. in: Proceedings of the 9th international Software Product Line Conference (SPLC 2005), 2005, pp. 7–20.
[54] S. Ferber, J. Haag, J. Savolainen, Feature interaction and dependencies: modeling features for reengineering a legacy product line, in: Proceedings of the Second Software Product Line Conference (SPLC2), 2002, pp. 235–256.
[55] T. Kojo, T. Männistö, T. Soininen, Towards intelligent support for managing evolution of configurable software product families, in: Proceedings of 11th International Workshop on Software Configuration Management (SCM-11), 2003, pp. 86–101.
[56] K. Czarnecki, S. Helsen, U.W. Eisenecker, Staged configuration through specialization and multilevel configuration of feature models, Software Process: Improvement and Practices 10 (2) (2005) 143–169.