A Koala-Based Ontology for Configurable Software Product Families

Timo Asikainen, Timo Soininen, Tomi Männistö
Helsinki University of Technology, Software Business and Engineering Institute (SoberIT)
PL 9600, FIN-02015 TKK, Finland
{timo.asikainen, timo.soininen, tomi.mannisto}@hut.fi

Abstract

In this paper, we present an approach to applying configuration techniques to managing the variability of configurable software product families. We introduce Koalish, an ontology for modelling configurable software product families, and give the ontology formal semantics by mapping it to weight constraint rules.

1 Introduction

Traditionally, the focus of configuration research has been on traditional products, i.e., mechanical and electronic ones. However, software product families (SPFs) have become an important approach to increasing the efficiency of software development and controlling complexity and variability [Bosch, 2000]. The most systematic of such families closely resemble configurable products in that they are composed of standard reusable assets and have a predefined structure; further, instances of the family can be deployed without customer-specific tailoring or addition of glue code [Bosch, 2000]. Therefore, they can be called *configurable software product families (CSPF)*. Modelling and managing variability of CSPFs can become a complex problem of its own.

In this paper, we study the possibility of utilising techniques developed for modelling and configuring traditional products for configuring CSPFs. There is some evidence that this can be done if the CSPF is described using a simple conceptualisation [Kojo *et al.*, 2003]; however, [Asikainen *et al.*, 2002] suggests that this cannot be done straightforwardly for more advanced conceptualisations. In this paper, we specify an approach to modelling the variability of CSPFs from a technical viewpoint. We further aim at automatically generating a product based on such a model and a set of requirements expressed in the language of the model, or checking the correctness of a product with respect to the model and requirements. We base our approach on a similar one [Tiihonen *et al.*, 2002] in the domain of product configurators. The front-end of our approach is a language used by software engineers for describing the commonalities and variabilities of a CSPF. To be usable, both the underlying concepts and the syntax of the language must correspond to the intuitions of software engineers. The description is automatically translated into a formal model. Based on the model, descriptions of members of the CSPF conforming to a set of user requirements can be generated, similarly as in configuration task. The description of a family member can be mapped into input of realisation tools, such as make, that are used to build the desired software system.

This paper describes how certain parts of the above-described approach can be implemented for a class of CSPFs. We introduce and formalise *Koalish*, an ontology serving as the conceptual basis of a language for describing a static view of architectures of CSPFs; for architectural views, see [Clements *et al.*, 2002]. The concepts of Koalish are based on Koala, a component model and architecture description language (ADL) that is in industrial use at Philips Consumer Electronics. Thus, Koalish only covers such CSPFs that lend themselves to be described using Koala. Koalish extends the concepts of Koala with constructs for modelling variability: we introduce possibilities for selecting the number and type of parts of components. Further, we present a mapping from Koalish to Weight Constraint Rule Language (WCRL) [Simons *et al.*, 2002]. This gives the ontology a sound formal basis, which enables implementing a language based on the ontology using a state-of-the-art inference tool [Tiihonen *et al.*, 2002].

Koala is relevant for CSPFs as the language enables composing multiple products from a single set of reusable assets. We believe that the practical success of Koala gives the ontology a solid foundation. Further, Koala is close to many other ADLs, which suggests that generalising our approach to other ADLs could be possible [Medvidovic and Taylor, 2000]. However, there is currently no support for configuration task in Koala. Thereby, Koalish can serve as the basis for a configuration front-end to Koala.

The remainder of this paper is organised as follows. In Section 2, we give an introduction to Koala and present Koalish. Thereafter, in Section 3 we define a mapping from Koalish to WCRL. Discussion and comparison to previous work follows in Section 4. Conclusions and outline of further work round up the paper in Section 5.

2 An ontology for modelling CSPFs

In this section, we first give an overview of Koala. Second, we present an ontology based on Koala and extended with constructs for representing variability.

2.1 Koala

Koala [van Ommering *et al.*, 2000; van Ommering, 2002] is an architecture description language and a component model (in the sense of Microsoft COM) that has been developed at Philips Consumer Electronics to be used in developing embedded software for consumer electronics devices. Between 100 and 200 developers distributed over 10 sites are involved with Koala, and a number of products developed with Koala are in the market [van Ommering, 2002].

The main modelling elements of Koala are *components* having explicit connection points called *interfaces*. 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 e.g. in C. Each interface is either a *provided* or a *required* interface; a provided interface signals that the component having the interface provides some service for its environment, and a required interface that such a service is required from the environment.

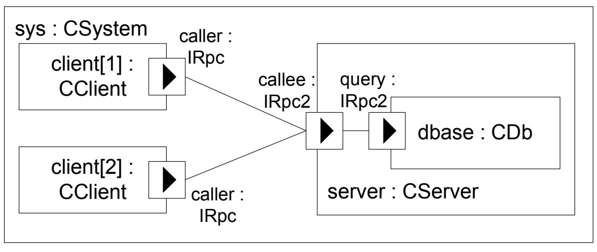
Components can have other components as their *parts*. Further, there can exist *bindings* between interfaces. A binding signals that one of the interfaces uses the service provided by the other. Bindings are related to function calls occurring in a software system. There are three different kinds of bindings. In a sense the fundamental kinds of bindings occur between a required and a provided interface: the binding translates calls to the functions of the required interface into calls to functions realising the provided interface. The other two types of bindings occur between interfaces in components that are in part-whole relationship with each other: calls to a provided interface in the whole component can be delegated to an interface in the part component; calls to required interfaces in the part component can be forwarded to a required interface of the whole component.

Example. Figure 1 (a) depicts a Koala system that will be used as a running example throughout the paper. See Figure 1 (b) for a legend of the notation. Components *client[1]* and *client[2]* are both of type **CClient**, *server* of type **CServer**, and *dbase* of type **CDb**. Further, above-mentioned components have interfaces: e.g., *client[1]* and *client[2]* have interface *caller* of type **IRpc**. Finally, there are bindings between interfaces: e.g., the interface *caller* in both *client[1]* and *client[2]* is bound into *callee* of *server*. ■

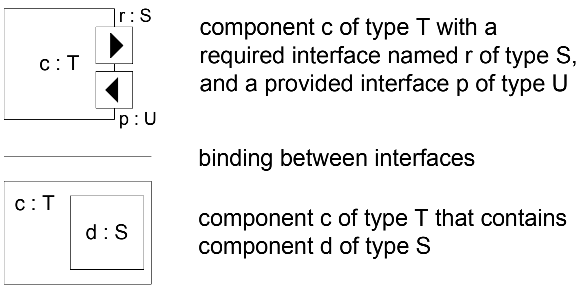
2.2 Ontology

In this section, we present Koalish, an ontology for modelling CSPFs. The ontology is based on combining Koala and an existing ontology of configurable products [Soininen *et al.*, 1998]. The basic concepts are the same as in Koala, but they are extended with constructs for modelling variation possibilities similarly as in [Soininen *et al.*, 1998].

We distinguish between *component* and *interface* types and *instances* of these. Occasionally, when there is no risk of ambiguity, we use the terms *component* and *interface* to refer to component and interface instances, respectively.



(a)



r : S
c : T
p : U
component c of type T with a required interface named r of type S, and a provided interface p of type U
binding between interfaces
c : T
d : S
component c of type T that contains component d of type S

(b)

Figure 1 (a) A sample Koala system (b) Legend of notation used

A *Koalish model* represents the reusable components and the possibilities for combining them. A Koalish model consists of a set of *component* and *interface types*, and a set of *constraints*. Such a model represents the properties of a software product family and contains information about component and interface types that can occur in configurations. A *configuration* consists of a set of instances of these types. Further, a *valid configuration* is a configuration that satisfies some conditions that will be explicated below.

In the following, we present different aspects of Koalish models and configurations based on them, and discuss the related properties of component and interface types.

Components and compositional structure

A component type is a description of an encapsulated piece of software with an explicit interface. We employ two main classes of information for representing the compositional structure of component instances. First, a component type defines the compositional structure of its instances through *part definitions*. Second, a component type defines the possibilities for its instances to interact with other components through a set of *interface definitions*.

A part definition consists of a *part name*, a *set of possible part types*, and a *cardinality definition*. The component type containing the part definition is termed *whole type*.

The part name distinguishes the role in which different component instances are parts of a component instance of the whole type. The set of possible part types consists of the types the instances of which may occur as parts of instances of the whole type with the associated part name. The cardinality is an integer range and specifies how many component instances must occur as parts with the part name.

A part definition is reflected in a valid configuration as follows: an instance of the whole type has the number of component instances specified by the cardinality as its parts. Each part must be an instance of one of the possible part types. The fact that a component instance has a component instance as a part with a given name is conceptualised as the two instances being in *has-part* relation with the part name.

The *has-part* relation organises component instances in composition hierarchies. Koala requires that in a valid configuration, all the components be contained in a single composition hierarchy. Each Koalish model defines a single *root component type*, and in a valid configuration, the root of the component hierarchy, termed *root component*, must be an instance of the root component type. The root component type may not contain interface definitions, neither can it appear as a possible part type in a part definition.

Example. Figure 1 (a) depicts a valid configuration. Component instance *sys* is the root component. Its parts are *client[1]*, *client[2]*, and *server*, which has *dbase* as part. ■

Koala excludes self-containment from the composition hierarchy, i.e., the composition hierarchy must indeed be a hierarchy. Thus, we require that no component type may, even transitively, contain a part definition where the component type occurs as a possible part type.

The above restriction implies that there is no self-containment for instances, either. In addition, we require that no component is a part of two or more components.

Interfaces and bindings

The possibilities for interaction between components are another important aspect of configurations. Interfaces are the points in components where all the interaction between the component and its environment occur.

An interface type represents a small set of semantically related functions. In a valid configuration, the instances of the interface type have the functions defined by their types.

An interface definition consists of an *interface name*, an *interface type*, a *direction definition*, and an *optionality definition*. The interface name specifies the role of the interface instance in the component instance, and the interface type the interface type an instance of which the interface instance must be. The direction definition has two possible values, *required* and *provided*: value required represents a *required interface*, and the value *provided* a *provided interface*. The optionality definition has two possible values, *optional* and *mandatory*: we will use the terms optional and mandatory interface in their obvious meanings.

An interface definition is reflected in a configuration as follows: for a mandatory interface, an instance of the component type must have an instance of the interface type as its interface; for an optional interface, the component type may have such an interface. The interface instance must have the name and direction specified in the interface definition, respectively. The fact that a component has an interface is conceptualised as the component and the interface being in the *has-interface* relation with the interface name.

Example. The interfaces *caller* in *client[1]* and *client[2]* are required ones, and *callee* and *query* provided ones. ■

Bindings are conceptualised with the binary relation *bound-to$_i$*. The pair (*a*, *b*) being in *bound-to$_i$* denotes the fact that *a* is bound to *b*. The reader should observe that bindings are not symmetric, but, on the contrary, asymmetric.

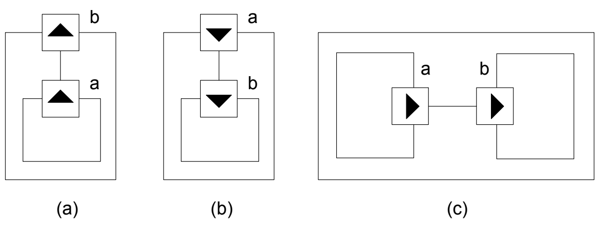


Figure 2 Different possibilities for place-compatibility

Example. In Figure 1 (a), the *caller* interfaces are bound to *callee*, which is, in turn, is bound to *query*. ■

In Koala, there are some restrictions on which bindings may occur in valid configurations. These restrictions are associated with the types of the bound interfaces, along with their directions and relative positions. Next, we will detail these restrictions.

First, if the pair of interfaces (*a*, *b*) is in the *bound-to$_i$* relation, then the type of *b* must be a subtype of *a*. In Koala, the binary relation *subtype* is defined as follows: interface type **A** is a subtype of interface type **B** if and only if **A** contains at least all the functions in **B**. It should be noticed that every interface type is a subtype of itself. Pair (*a*, *b*) of interface instances fulfilling this requirement is said to be *type-compatible*.

Second, *a* and *b* must have certain directions and be in an appropriate relative position. There are three possible combinations that are illustrated in Figure 2. If a pair of interfaces fulfils any of the following requirements, the pair is said to be *place-compatible*. First (Figure 2 (a)), both *a* and *b* are required interfaces, and the component having *b* as its interface has the component having *a* as its interface as its part. Second (b), both *a* and *b* are provided interfaces, and the component having *a* as its interface has the component having *b* as its interface as its part. Third (c), *a* is a required interface, *b* is a provided interface, *a* and *b* are interfaces of different components, and both of these components are parts of the same component.

In addition to bindings between interfaces, there can exist bindings between individual functions contained in interface instances. Such bindings are termed *function bindings*. A function binding is conceptualised as a pair of functions (*a*, *b*) being in the *bound-to$_f$* relation; we say that *a* is bound to *b*. The semantics of a function binding are similar to those of a binding between a pair of interfaces: function *a* being bound to function *b* implies that calls to *a* are translated into calls to *b*.

Bindings (between interfaces) imply certain function bindings: if interface *a* is bound to interface *b*, then every function of *a* is bound to a function in *b* with the same name. However, there can exist function bindings beyond the above-mentioned ones. Further, as functions have different semantics, not all bindings between functions are possible. Finally, only function bindings, where the interfaces containing the functions are place-compatible (recall Figure 2), can occur in valid configurations.

In order for a configuration to be valid, certain bindings must exist between interfaces: every function in every required interface must be bound to exactly one function.

Binding constraints are used for specifying bindings between interfaces or functions in a configuration. A binding constraint is a logical condition that refers to interface definitions in components; in a configuration, a binding constraint is reflected as bindings between interface instances or function bindings.

Constraints

A *constraint* is a formal rule that specifies a condition that must hold in a valid configuration. Constraints are expressed using a constraint language. However, due to space limitations, we do not specify one here. In short, the constraint can refer to the parts and interfaces of components and bindings between interfaces and functions, and such references can be combined using Boolean connectives.

3 Formalisation

In this section, we give formal semantics to Koalish. We accomplish this by first giving a short introduction to weight constraint rules (WCRs), and thereafter define the mapping from Koalish to WCRs.

3.1 Weight constraint rules

WCRL is a general-purpose knowledge representation formalism similar to logic programs. WCRL has been shown to be suitable for representing configuration modelling concepts, see [Soininen *et al.*, 2001]. Due to space limitations, giving a thorough account of WCRs is not possible. Instead, we use an example to demonstrate the aspects of weight constraints rules that are most relevant for our purposes. For a full account of weight constraint rules, the reader should refer to [Simons *et al.*, 2002].

The structure of a company board can be expressed using WCRs as follows. We model the possible members of the board using *object constants*, such as *alice*, *bob*, and *carol*. Further, we use *domain predicates* for classifying object constants in a similar manner as classes or types are used in programming languages. The domain predicate *person* denotes the fact that certain object constants are persons:

$$person(alice) \leftarrow \qquad person(bob) \leftarrow \qquad person(carol) \leftarrow$$

The above kind of *rules* are termed *facts*, and the *person*(X) are *predicates*. The semantics of a rule are roughly that if all the predicates on the right side (termed *body*) are true, then the left side (*head*) must be true as well.

The unary *predicate member* represents the fact that a person is a member of the board.

The board must have at least five and at most seven persons as its members. This is represented as follows:

$$5 \; \{ \; member(X) : person(X) \; \} \; 7 \leftarrow$$

In the above rule, the part on the left of the arrow is a *cardinality constraint*. They can be used in place of a predicate anywhere in a rule.

The board has a chairman that is also a member of the board. We use the domain predicate *chair*(X) to denote the fact that X is the chairman of the board. The fact that the a chairman is a member is formalised as follows:

$$member(X) \leftarrow chair(X)$$

Similarly, the board has a vice-chairman, which is also a member of the board. We employ the following *integrity constraint* to prevent the chairman and the vice-chairman being the same person:

$$\leftarrow chair(X),\ vice(X)$$

3.2 Formalisation of the ontology

In the following, we distinguish between two classes of rules, ontological definitions and rules representing a configuration model. Ontological definitions are the same for every Koalish model. Rules representing a configuration model are Koalish model, i.e. product, specific. Ontological definitions will be enclosed in boxes; the rest of the rules formalise the example presented above.

Throughout the translation, we assume that the input is valid with respect to the criteria mentioned in the previous section. Hence, there is no need for constraints stating that components are singly typed, components may not be self-contained, or that all components must be contained within a single composition hierarchy etc. Consequently, these and similar constraints are omitted from the translation.

Compositional structure

Component types are represented as domain predicates, component instances as object constants, and their compositional structure with the predicate $hp(c_1, c_2, pn)$, which is interpreted as component instance c_1 having component instance c_2 as its part with part name pn. Part names are represented as object constants, and the set of part names is represented using the unary domain predicate *pan*.

The set of component instances in a configuration is represented with the predicate $in(c)$. Further, the domain predicate *cmp* is used to represent the set of all component instances. The domain predicate *root* represents the root component instance.

A part definition is represented as a rule that employs a cardinality constraint. The domain predicate $ppa(c_1, c_2, pn)$ denotes the fact that component c_1 may have component c_2 as its part with part name pn.

Example. As our formalisation defines a mapping from a Koalish model to a set of rules, using the configuration of Figure 1 (a) as an example is not possible. Instead, we use a Koalish model as our example; the configuration we have used so far is a valid configuration of the model, but the model has other valid configurations as well.

We assume that there is a sufficient amount of object constants representing instances of each component type. For brevity, we omit the discussion about what is a sufficient amount of instances; see [Tiihonen *et al.*, 2002] for a solution to a similar problem. We introduce the instances using facts like:

$$system(c_0) \leftarrow \qquad server(c_1) \leftarrow \qquad client(c_2) \leftarrow$$

The root component type in our example is **CSystem**. This is represented as follows:

$$root(C) \leftarrow system(C)$$

The following kind of rules conceptualise the fact that instances of every component type are components.

$$cmp(C) \;\leftarrow\; system(C) \qquad cmp(C) \;\leftarrow\; server(C)$$

The part names occurring in the configuration model are represented as facts as follows:

$pan(client) \leftarrow \quad pan(server) \leftarrow \quad pan(dbase) \leftarrow$

The sets of possible instances for each part definition are defined using rules like:

$ppa(C_1, C_2, client) \leftarrow system(C_1), client(C_2)$

The part definitions themselves are formalised using the following kind of rules. The following rule says that there are one, two or three clients in a system:

$1 \{ hp(C_1, C_2, client) : ppa(C_1, C_2, client) \} 3 \leftarrow in(C_1), system(C_1)$ ■

There must exist a single root component:

$\boxed{1 \{ in(C) : root(C) \} 1 \leftarrow}$

The part name pn in predicate $hp(c_1, c_2, pn)$ will in some rules be of no interest. Thus, we introduce a binary predicate $hp(c_1, c_2)$, with the semantics that c_1 has c_2 as a part.

$\boxed{hp(C_1, C_2) \leftarrow hp(C_1, C_2, N), ppa(C_1, C_2, N)}$

A component is in the configuration if it is a part of some other component.

$\boxed{in(C_2) \leftarrow hp(C_1, C_2), ppa(C_1, C_2, N)}$

The fact that no component instance may be a part of two or more component instances is represented as follows:

$\boxed{\leftarrow cmp(C_2), 2 \{ hp(C_1, C_2) : cmp(C_1) \}}$

Interfaces and bindings

Interfaces types are represented as domain predicates, and interface instances as object constants. The predicates *req* and *prov* represent the sets of required and provided interface instances, respectively. The interfaces of component instances are modelled with the predicate $hi(c, i, iname)$, which is interpreted as component c having interface i with the interface name *iname*. The set of interface names is represented using a domain predicate, in this case $intn(iname)$.

The functions defined in an interface type, and thus contained in the instances of the type, are captured using the predicate $cf(i, f)$, where i is an interface instance and f is a function signature, and the semantics are that i contains f.

The predicate $tc(i_1, i_2)$ is used to capture the idea of type-compatible interfaces. The semantics are that the pair (i_1, i_2) is type-compatible.

Similarly as for components, the predicate *in* is used to represent the fact that an interface instance is in the configuration. Further, the predicate *int* is used to represent the set of all interface instances.

An interface definition is represented as a cardinality constraint. The predicate $pint(c, i, intn)$ signals that component c may have interface i as its interface by the name *intn*.

Example. As for components, we must assure that there is a sufficient amount of interface instances of each type and direction available. This is achieved by rules like:

$rpc(i_0) \leftarrow \quad prov(i_0) \leftarrow$

The example did not define the functions contained in each interface type. Let *rpc* contain f; *rpc2* contains f and g:

$$\begin{array}{lll} int(I) \leftarrow rpc(I) & & int(I) \leftarrow rpc2(I) \\ cf(I, f) \leftarrow rpc(I) & & cf(I, f) \leftarrow rpc2(I) \\ & & cf(I, g) \leftarrow rpc2(I) \end{array}$$

CRpc2 is a subtype of **CRpc**, and similarly **CRpc** and **CRpc2** of themselves. This implies type-compatibilities:

$$tc(I_1, I_2) \leftarrow rpc(I_1), rpc2(I_2)$$

$$tc(I_1, I_2) \leftarrow rpc(I_1), rpc(I_2) \quad tc(I_1, I_2) \leftarrow rpc2(I_1), rpc2(I_2)$$

The interface names are captured by the facts:

$intn(callee) \leftarrow \quad intn(caller) \leftarrow \quad intn(query) \leftarrow$

The possible interfaces are represented using rules like:

$pint(C, I, caller) \leftarrow client(C), rpc(I), req(I)$

The rule corresponding to the definition would be:

$0 \{ hi(C, I, caller) : pint(C, I, caller) \} 1 \leftarrow in(C), client(C)$ ■

The predicate $bn(i_1, i_2)$ represents the fact that interface i_1 is bound to i_2. Further, the predicate *pc* is used to represent the pairs of interfaces that are place-compatible. Each of the three following rules capture one of the ways in which a pair of interfaces can be place-compatible. The rules use a binary version of the predicate *hi* that is defined analogously to the binary version of *hp*; we omit the definition for brevity.

$$\boxed{\begin{array}{rl} pc(I_1, I_2) \leftarrow & req(I_1), req(I_2), hi(C_1, I_1), hi(C_2, I_2), \\ & hp(C_1, C_2), cmp(C_1), cmp(C_2) \\ pc(I_1, I_2) \leftarrow & prov(I_1), prov(I_2), hi(C_1, I_1), hi(C_2, I_2), \\ & hp(C_1, C_2), cmp(C_1), cmp(C_2) \\ pc(I_1, I_2) \leftarrow & req(I_1), prov(I_2), hi(C_1, I_1), hi(C_2, I_2), \\ & C_1 \neq C_2, hp(C_3, C_1), hp(C_3, C_2), cmp(C_1), \\ & cmp(C_2), cmp(C_3) \end{array}}$$

A binding between a pair of interfaces can exist if the pair is both type- and place-compatible:

$\boxed{0 \{ bn(I_1, I_2) \} 1 \leftarrow tc(I_1, I_2), pc(I_1, I_2)}$

The fact that a binding between two interfaces implies bindings between functions is captured as an ontological definition in which the predicate $bnf(i_1, f_1, i_2, f_2)$ has the semantics that the function with signature f_1 contained in interface i_1 is bound to function f_2 contained in i_2.

$\boxed{bnf(I_1, F, I_2, F) \leftarrow bn(I_1, I_2), int(I_1), int(I_2), cf(I_1, F)}$

The predicate $cmbf(f_1, f_2)$ captures the idea of compatible functions between which there can exist a function binding.

$$\boxed{\begin{array}{l} 0 \{bnf(I_1, F_1, I_2, F_2) \} 1 \leftarrow cmbf(F_1, F_2), pc(I_1, I_2), \\ \qquad cf(I_1, F_1), cf(I_2, F_2) \end{array}}$$

An interface instance is in the configuration when it is contained in a component.

$\boxed{in(I) \leftarrow hi(C, I), pint(C, I, N)}$

We define a binary version of the predicate *cnf*:

$\boxed{cnf(I_1, F_1) \leftarrow cnf(I_1, F_1, I_2, F_2), cf(I_1, F_1), cf(I_2, F_2)}$

The requirement that each function in every required interface must be bound to exactly once is formalised follows:

$$\boxed{\begin{array}{c} \leftarrow \{ cnf(I_1, F_1) \} 0, cf(I_1, F_1), req(I_1), \mathrm{in}(I_1) \\ \leftarrow 2 \{ cnf(I_1, F_1) \}, cf(I_1, F_1), req(I_1), \mathrm{in}(I_1) \end{array}}$$

4 Discussion and previous work

Research closely related to this paper has been conducted earlier, both in the configuration and software architecture domain. However, we do not know of earlier attempts of conceptualising and formalising concepts of ADLs extended with variability. This is our main contribution.

As stated above, Koalish intentionally borrows a number of concepts originally introduced in the configuration domain: for instance, compositional structure and connections between explicit connection points are established modelling primitives in the domain, see, e.g. [Soininen *et al.*, 1998]. However, there is hardly any research in the configuration domain that would take the special characteristics of software into account.

The most significant distinguishing factor between Koalish style CSPFs and traditional products seems to be the notion of interface in CSPFs. First, interfaces are different from ports, as defined in [Soininen *et al.*, 1998]. Most importantly, the connections between interfaces are directed, between ports undirected. Further, a port instance can generally accept only one connection, but there can be any number of connections to an interface. Second, interfaces are different from resources, see, e.g., [Soininen *et al.*, 1998]. The fact that resources are produced and used creates a degree of similarity between resources and interfaces. However, there are significant differences: resources are typically balanced without explicit connections between the consumers and users, whereas interfaces must always be explicitly connected. Further, there is no notion within the resource concept corresponding to the internal structure of interface, i.e., functions. Due to these important semantic differences, interfaces cannot be adequately captured with the existing configuration modelling concepts.

On the other hand, the concepts of Koalish are practically the same as Koala, and Koalish shares many of its concepts with other ADLs. This is also the intention: to define a conceptualisation that can be used as the basis for developing tools for managing CSPFs. In order to achieve this task, our conceptualisation incorporates explicit variability mechanisms and formal semantics. There are some ADLs that have variability mechanisms or formal semantics, but, to the best of our knowledge, there is no ADL having both. An example of an ADL is the conceptualisation introduced by van der Hoek et al. [van der Hoek *et al.*, 1998]: the conceptualisation includes some variability mechanism, but is not defined formal semantics.

5 Conclusions and further work

We presented and formalised Koalish, an ontology for configurable software product families based on Koala. The formalism used allows the compact representation of software architectures and using intelligent tool support in modelling and deploying a variety of software systems from a configurable software product family.

However, there still remain a large number of unresolved issues. First, for practical purposes, a language committing to the software ontology presented in this paper and an exact mapping from the language to the weight constraint rules should be defined. Second, software tools supporting modelling, configuring, and checking the correctness of configurations should be developed and integrated with tools currently used in software development: a configuration could serve as an input for tools such as make. Currently, we are implementing a prototype tool that takes as an input a language implementing the concepts of Koalish. Further, the tool is able to translate the description into weight constraint rules, which serve as the input for *smodels*, an inference tool operating on weight constraint rule language.

The ontology presented in this paper should be validated and possibly improved by modelling actual software product families with it. Likewise, the computational feasibility of the configuration task should be analysed by conducting empirical experiments with real software products, and theoretical complexity analysis. Finally, in order to be widely applicable, the ontology should be extended to cover new aspects and kinds of software product families.

Acknowledgements

This work has been supported by Academy of Finland (project 51394) and National Technology Agency of Finland.

References

[Asikainen et al., 2002] Asikainen, T., Soininen, T. and Männistö, T. Representing Software Product Family Architecture Using a Configuration Ontology, in *Configuration Workshop of ECAI 2002*, 2002.

[Bosch, 2000] Bosch, J. *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Addison-Wesley, Boston (MA), 2000.

[Clements *et al.*, 2002] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. *Documenting Software Architecture*. Addison Wesley, 2002

[van der Hoek *et al.*, 1999] van der Hoek, A., Heimbigner, D. and Wolf, A. *Capturing Architectural Configurability: Variants, Options, and Evolution*. Technical report CU-CS-895-99, University of Colorado, 1999.

[Kojo et al., 2003] Kojo, T., Männistö, T. and Soininen, T. Towards Intelligent Support for Managing Evolution of Configurable Software Product Families, in *Proc. of 11th Int. Workshop on Software Configuration Management (SCM-11)*, LNCS 2649, 2003.

[van Ommering, 2002] van Ommering, R. Building Product Populations with Software Components, in *Proc. of the 24th Int. Conf. on Software Engineering*, 2002

[van Ommering *et al.*, 2000] van Ommering, R., van der Linden, F., Kramer, J. and Magee, J. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3): 78-85. 2000.

[Simons *et al.*, 2002] Simons, P., Niemelä, I. and Soininen, T. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1-2): 181-234. 2002.

[Soininen *et al.*, 1998] Soininen, T., Tiihonen, J., Männistö, T. and Sulonen, R. Towards a General Ontology of Configuration. *AI EDAM*, 12(4): 357-372. 1998.

[Tiihonen *et al.*, 2002] Tiihonen, J., Soininen, T., Niemelä, I. and Sulonen, R. Empirical Testing of a Weight Constraint Rule Based Configurator, in *Configuration workshop of ECAI 2002*, Lyon, France, 2002