Representing Software Product Family Architectures Using a Configuration Ontology

Timo Asikainen[1], Timo Soininen[1] and Tomi Männistö[1]

Abstract. In this paper, we study the possibility of applying techniques developed for configuring mechanical and electronics products to configuring software. We analyze and compare at the conceptual level software architecture description languages and configuration modelling concepts. Based on the analysis we are able to define a way of representing much of the architectural knowledge using the configuration modelling concepts. This indicates that it is relatively easy to provide software configuration support using the existing techniques if the software is represented through architectural descriptions. However, there are also some differences that require extending the current conceptualizations of configuration knowledge to capture software products adequately.

1 INTRODUCTION

In the recent years there has been an increasing research effort dedicated to providing better configuration modelling languages and tools. However, the research on configuration has mainly dealt with mechanical and electronics products. At the same time, software product lines or families have become increasingly important in the software industry [1]. The most systematic of such families closely resemble configurable products in that they are composed of standard re-usable assets and have a predefined architecture [1]. A major effort in the software product family and architecture research has been spent on developing architecture description languages (ADLs) for representing these re-usable assets and software architectures. Thus product families and ADLs are natural counterparts in the software domain for configurable products and configuration modelling languages. There are many ADLs and large differences between them [2,3].

In this paper, we study the possibility of applying techniques developed for modelling and configuring mechanical and electronics products to configuring software. A prerequisite for coming up with a general solution to this problem is to define a mapping from the conceptualization of software systems to a conceptualization of configuration knowledge. Towards this end, we analyze three prominent ADLs at the conceptual level and compare them with the major concepts used for modelling configuration knowledge. Based on the analysis and comparison, we show how to represent main concepts of ADLs using the configuration modelling concepts. In addition, we identify several potential needs for extending the configuration modelling concepts with ADL derived concepts.

For the purposes of this paper we concentrate on three important ADLs: Acme [4,5,6], Wright [7,8] and Koala [9,10]. Out of these, Acme has been designed to include features of other ADLs that its designers considered central. The relevance of Acme is further promoted by the fact that one of the goals of Acme is to serve as an interchange language for other ADLs. Wright is a widely cited ADL that has a rigorous semantics and describes behavioural aspects of software. Both the use of formal methods and description of behaviour make Wright important among ADLs. Koala is in commercial use at Philips Consumer Electronics. Being one of the few ADLs used in commercial applications, it is an important example of the practical aspects of ADLs.

As the reference point in the comparison we employ a configuration ontology presented by Soininen et al. [11]. This ontology synthesizes prior conceptualizations of configuration knowledge. Moreover, it is very similar to another recognized configuration ontology presented by Felfernig et al. [12]. Thus, as it seems to cover most approaches to configuration modelling, it is a natural reference point for conceptual level analysis.

The remainder of this paper is organized as follows: An overview of software architecture and ADLs will be given in Section 2. Section 3 introduces our framework for analyzing and comparing ADLs along with the most important characteristics of three ADLs. In Section 4, a comparison between the ADLs and the concepts of the configuration ontology is presented. A mapping from the most important concepts of ADLs to the concepts of the configuration ontology is given in Section 5 and potential extensions of the ontology are discussed in Section 6. We discuss our findings and previous work in Section 7 and finally give our conclusions and topics for further research in Section 8.

2 SOFTWARE ARCHITECTURES AND ARCHITECTURE DESCRIPTION LANGUAGES

Software architecture of a system purports to describe the high-level structure of a software system. The significance of considering architecture when designing software systems is well understood. There is, however, no single, generally accepted method for describing software architecture. Simple methods, such as referring to an existing architectural style or using box-and-line diagrams with no or vague semantics, have been recognized to be inadequate for the task [13]. Hence, there is a need for better methods.

Architecture description languages (ADLs) are a promising candidate solution for the architecture description problem. Loosely defined, ADLs are formal notations with well-defined semantics, whose primary purpose is to represent the architecture of software systems. A large number of ADLs have been proposed. ADLs have in common the concept of component, although different ADLs have different names for the same concept [3]. But in their other characteristics, ADLs differ from each other radically. Some of them address a special application domain and others are dedicated to a specific architectural style [3]. ADLs also employ different

---

[1] Helsinki University of Technology, TAI Research Centre and Software Business and Engineering Institute, P.O. Box 9600, FIN-02015 HUT, Finland. Email: {Timo.Asikainen, Timo.Soininen, Tomi.Mannisto}@hut.fi

formalisms for specifying semantics, and there is variety in how rigorously the syntax and semantics are defined.

The most fundamental elements of architectural descriptions include *components*, *connectors* and their *configurations* [3,4,13].

Components represent the main computational elements and data stores of the system. Intuitively, they correspond to the boxes in the box-and-line diagrams. Clients, servers and filters are examples of components. In a working system, a component might manifest itself as an executable file or a dynamic link library. [4]

Unlike components, connectors are not loci of application specific computation in software systems. Instead, they represent interactions between components. In a box-and-line diagram, connectors are depicted as lines between the boxes. Examples of connectors include method invocation, pipes and event broadcast. [4]

Components can be connected to each other to form configurations. They are sometimes referred to as systems [4] or architectural configurations [3]. In many ADLs, components can only be connected through connectors; explicit use of connectors has even been proposed a defining characteristic of an ADL [3]. Typically, components are connected to each other through *connection points*. Different ADLs call these connection points with different names, e.g. port, role or interface.

In some ADLs, components can also have an inner structure. Such components are called *compound components* and they represent a subsystem that has an architecture of its own. With composite components it is important to be able to specify how the inner parts of the component are linked to the component itself. Usually, the linkage is defined by binding connection points of the compound component with connection points of its parts. Intuitively, binding means that the connection point of the compound component is in fact a connection point of some other component inside the compound component.

A practical concern with ADLs is the tool support available for them. Tool support is out of the scope of this paper, since the goal is to analyze the modelling languages. However, it should be noted that support for generating executable systems out of architectural descriptions is one of the goals of research on ADLs [3]. This is a goal shared by research on configuration modelling.

3 ANALYSIS OF THREE ARCHITECTURE DESCRIPTION LANGUAGES

In this section, we first define a framework for analyzing and comparing the concepts of ADLs with those of configuration. Thereafter, we use the framework to study three ADLs: Acme [4,5,6], Wright [7,8] and Koala [9,10].

3.1 Framework for analysis and comparison

The fundamental phenomena described by the configuration ontology and that presented in [12] are: taxonomies, structure, topology, resources, functions and constraints. Underlying all the abovementioned phenomena is the division of configuration knowledge into three classes, *configuration model knowledge*, *configuration solution knowledge* and *requirements knowledge*. *Types* and *instances* are entities occurring in the configuration model knowledge and configuration solution knowledge, respectively.

In the following three subsections, we will analyse the abovementioned ADLs using a comparison framework composed of three parts. The first part includes the key concepts of ADLs and the configuration ontology, and the relations between them. The concepts include *components*, *connectors*, *configurations*, *connection points*, *attributes*, *resources*, *functions* and *constraints*. The relations include *topology*, *taxonomy* and *structure*. The second part considers the existence of different concepts for types and instances. The last part of the framework is the variation mechanisms provided by ADLs and the configuration ontology.

3.2 Acme

The basic concepts of Acme are *components*, *connectors* and *systems*. System is the Acme term for configuration. On the other hand, there are no constructs for resources or functions in Acme. Both components and connectors have connection points that are called *ports* for components and *roles* for connectors. *Design elements* include component, connector, port and role. Components are connected to connectors by defining an *attachment* between the port of a component and the role of a connector. One connector may connect multiple components. Components cannot be connected directly to each other and neither can a connector to another connector. [4]

Components and connectors can have attributes that are called properties in Acme. Properties are uninterpreted values, i.e. they do not have any semantics defined.

In Acme, design constraints can be defined using first order predicate logic. They can be either *invariant* or *heuristic*: invariant constraints must hold, whereas heuristic constraints are merely hints of what should be true for an Acme system. Constraints can be used to express various aspects of Acme systems: e.g. the existence and values of properties and the connections present in a system. [5]

In addition, Acme includes a structure called *representations* that can be used for describing an alternative view of a component or a connector. *Rep-maps*, or in other words, *representation maps*, can be used to specify the correspondences between different representations of a design element. There is, however, no semantics defined for either representations or rep-maps. One possible use of these constructs is representing the compositional structure of a component and the correspondences between the ports and roles of the compound component and those of the contained components. [4].

Although types are not first class entities in Acme, it has two type systems: one for design element types and, and another for systems. Types in the design element type system are sets of required structure, i.e. design element declarations, and values. New types can be formed from existing types through subtyping. System types are called *families*. A family consists of design element type definitions. Subtypes of families can be formed through single or multiple inheritance. Also, a system can be declared to be a member of many family types. [6]

What makes types a secondary concept in Acme is that design elements and systems need not have a type or be a member of a family, respectively. A design element being of a given type merely implies that the design element has the structure and values specified by that type. Similarly with families, a system being a member of a family signals that the type definitions of the family are type definitions of the system, too. Therefore, type systems of Acme can be considered a sort of macro expansion mechanism.

The syntax and semantics of Acme are formally defined, the latter in terms of a mapping to first order predicate logic.

There seem to be no constructs in Acme for modelling variety. What seems to come closest to modelling variability is the family construct. It can be used to specify a set of type definitions shared by a set of systems. Furthermore, constraints can be used to enforce the instantiation of certain design elements. Hence, the family definitions complemented with constraints seem to provide a mechanism for specifying product families with certain properties.

3.3 Wright

As in Acme, there are *components*, *connectors*, *systems*, *ports*, *roles* and *attachments* in Wright and their semantics are the same in both languages. There are no attributes, resources or functions. What distinguishes Wright from Acme and makes it special among ADLs is its way of specifying the behaviour of ports, roles, connectors and components, and the possibilities for analysis based on these specifications. Wright uses CSP (Communication Sequential Processes) specified in [14], a formal approach for two purposes: (1) specifying processes that reside in Wright elements and (2) defining semantics of non-CSP parts of the language. In short, CSP is a formal method for specifying and analyzing the behaviour of objects in terms of sequences of events in which they engage. The pattern of events that is possible for an object is termed a *process*. [7,8]

Each port and role is associated with a CSP process. In addition, each connector and component includes a separate glue and computation process, respectively. The glue of a connector defines the operation of the connector as an entity. That is, the glue coordinates the operations of the other processes in the connector. Ports are attached to roles to form systems. Which ports can be attached to which roles, is determined by their process descriptions. The basic idea is that a port can be attached to a role if the port will behave well in all situations enabled by the role. In other words, CSP defines a compatibility relation between ports and roles.

The second usage of CSP in Wright, defining semantics of non-CSP parts of the language, allows using tools operating on CSP to reason about properties, most notably about dead-lock freedom, of a Wright connector. This is an important class of tool support enabled by the rigorously defined semantics of Wright.

Wright allows describing hierarchical structure of both components and connectors. This is done by enclosing a system into the place of a process. In addition to the normal system specification, bindings between the port and role names in the enclosing element and those specified in the enclosed system need to be specified.

Wright distinguishes between component and connector types and instances. Each connector and component is of exactly one type. There is, however, no taxonomy of types.

In addition to component and connector types, Wright includes a construct called *style*. Styles are collections of type definitions and *constraints*. They are expressed in first order predicate logic and they can be used in a manner similar to that in Acme described above. In addition to component and connector type definitions, a style can include *interface type* definitions. They are process descriptions that can be used in port and role definitions.

Type definitions in styles can be parameterized. That is, parts of the type definition can be left open and a value can be filled in when the type is instantiated. New styles can be defined in terms of existing ones through subtyping: the new style has the same type definitions and constraints as the old one plus some additional type definition or constraints.

Variation mechanisms of Wright are similarly limited as for Acme, although Acme uses the term family where Wright uses style. In short, styles supplemented with constraints seem to be able to express variability.

3.4 Koala

As the languages described above, the Koala model has *components* as a main design element. But in other respects, Koala differs greatly from its peers. In Koala, there is no notion of connectors, resources, functions or constraints. *Configurations* are comprised of components connected to each other through *interfaces* that are the connection points in Koala. The connection between components is not symmetric: a distinction is made between *provided* and *required interfaces*. Loosely defined, a component having a provided interface means that the component offers some service for other components to use. Similarly, a required interface signals a service being required by the component from some other component. Koala interfaces are similar to those in COM or Java. [10]

There are some limitations on how interfaces can be connected to each other: only required and provided interfaces of the same interface type can be connected with each other and each required interface must be connected to a single provided interface. On the other hand, a provided interface can be connected to any number of required interfaces, including zero.

In addition to connecting interfaces to each other, it is possible to connect constituent parts of interfaces directly. These parts are called *functions*. Hence, interfaces in Koala are not atomic even when considered as connection points.

Koala has a type system: a distinction is made between both interface and component types and instances. There is, however, no taxonomy of component or interface types.

Compound components can be used to express compositional structure in Koala, i.e. other components can be contained within a component. An interface of a compound component can be bound to an interface defined by a contained component.

Koala includes a construct, *module*, which is a component without an interface of its own. Modules are used inside compound components for gluing interfaces. Suppose, for example, that each component contained in a compound component has an initialization interface to be called before using the component. Due to binding rules, it would not be possible to bind all these interfaces to any single interface of the compound component. Therefore, a new configuration specific module is added: when the initialization function for the compound component is called, the call is routed to the module, which in turn calls the initialization functions of all necessary components in the order desired.

In addition to the constructs already mentioned, Koala provides mechanisms for handling both the *internal diversity* of components and the *structural diversity* in a configuration. Internal variety is manifested as variation of component parameters. There may be dependencies between parameters: a parameter value may imply that another parameter has a certain value. Structural diversity pertains to alternative provided interfaces for a required interface: e.g. there may be multiple components that provide the same interface required by a certain component. The choice between the interfaces is made by a construct called *switch* either statically, that is at compile time, if the information required for the selection is available, or, otherwise, dynamically at runtime.

We have no information about whether Koala has formally defined syntax or semantics.

4 COMPARISON OF CONCEPTS OF THE ADLS WITH THE CONFIGURATION ONTOLOGY

In this section, we use our framework defined in the previous section for comparing the concepts and constructs found in the ADLs with those of the configuration ontology.

4.1 Key Concepts and the Relations between Them

Component is the central concept of Acme, Wright and Koala. It is also present in the configuration ontology with that same name. The semantics are as well similar: components represent the defining parts of a system in configuration modelling, too. In addition, systems as defined in Acme and Wright and configurations as defined in Koala have a counterpart in the configuration ontology, namely configuration.

The notion of connection points is also common to all the studied modelling methods. In Acme and Wright they are called ports and roles in components and connectors, respectively. In Koala connection points are termed interfaces and in the ontology ports. The semantics of connection points are also similar in all the disciplines: they denote the mechanism for connecting other entities.

Connectors are first-class citizens in Acme and Wright. However, there are no connectors in the configuration ontology or Koala. Thus, there is a major difference in how the disciplines handle architectural connection – an important issue in both ADLs and in the configuration ontology.

What then is the reason for this disagreement in architectural connection? We believe that at least a partial reason for the importance of connectors in Acme and Wright can be found in the underlying assumptions of them and several of the ADLs not studied in this paper: a major issue in software architecture has been reusing existing components. Furthermore, there has been considerable effort in the software engineering community to reuse heterogeneous components, which cannot be connected directly to each other due to different communication mechanisms and various other reasons. Therefore, connectors have been introduced in ADLs as a vehicle for connecting heterogeneous components.

In Koala, the situation is rather different: components are homogenous and there seems to be no problem in connecting them directly, i.e. without connectors. Hence, Koala is much closer to the configuration ontology than Wright and Koala.

Resources, a feature present in the configuration ontology but not in any of the ADLs, is similar to the notion of provided and required interfaces present in Koala in the sense that they are both anti-symmetric. What is more, resources are produced and consumed by components, just as interfaces are provided and required. However, resources are produced and consumed in certain quantities, which gives them more expressive power compared with the notion of provided and required interfaces.

In addition to simulating provided and required interfaces, resources can be used to model other relevant quantities. Such quantities include memory, power, output capacity and throughput. The software engineering community has considered similar issues important [15]. Hence, resources could very well be an important feature of the configuration ontology when used to model software architecture.

Modelling functions is another feature of the configuration ontology that all three ADLs presented in this paper lack. Functions are an important aspect of software engineering usually termed features in the domain [16]. We believe that also functions could be very useful when modelling software with the ontology.

All the ADLs have some mechanisms for modelling structure. However, the configuration ontology provides much stronger mechanisms: the configuration ontology provides a wide range of variation mechanisms. Furthermore, in the configuration ontology a component can be a part of many components simultaneously, which is not possible in any of the ADLs.

All the disciplines except Koala have explicit mechanisms for expressing constraints. Further, in all disciplines where constraints exist, they are logical expressions about the non-behavioural properties of a system modelled in that discipline. A difference is that in the configuration ontology, there is no direct support for heuristic constraints as defined in Acme. Support for modelling preferences and optimization criteria have been identified as important and developed in other research on configuration.

4.2 Distinction between Types and Instances

All the three ADLs have some distinction between types and instances. In Acme, the distinction is rather weak, as the type systems can be seen as a simple macro expansion mechanism. Nevertheless, there is taxonomy between the Acme types. The situation is rather similar in Wright: types bear a little meaning as such. The only function of types seems to be facilitating in defining and altering recurring patterns. In Koala, interface types are strong in the sense that only interfaces of the same type can be connected. There is, however, no taxonomy for the interface types. The component types seem to have no function beyond defining the structure of a set of components. Hence, component types seem to be as a construct as weak as types in Acme and Wright.

In the configuration ontology, strong distinction between types and instances is one of the basic assumptions and is made for all kinds of entities. Types are organized in taxonomies.

4.3 Variation Mechanisms

A question closely related with the distinction of types and instances is: What is being modelled, one product or a product family. The configuration ontology aims at modelling product families. Configuration model knowledge defines the common properties of the family members. A lot of variation mechanisms are provided.

As stated in the analysis of Acme and Wright, both of these languages can be seen to provide some support for modelling variability: there are no explicit variation mechanisms, but the combination of system types and constraints seem to be able to express common structure shared by a set of products.

In Koala, there is some knowledge about the common properties of all the products: component and interfaces definitions are stored in a component repository and they are common to different systems to be constructed [10]. In fact, type definitions shared by a set of products is exactly the same phenomenon be have already seen in provided by the family construct Acme and by the style construct in Wright. As there are no constraints to complement the shared type definition in Koala, the support provided by Koala for variability is weaker than that Acme and Wright.

In the previous section, it was stated that Koala could model both internal and structural variety. How does this statement relate to the above observation that Koala provides a weaker support for variability than Acme and Wright? We claim that we are dealing with two distinct forms of variability. The variability in Acme and Wright can be used to span a set of products with many similarities, or in other words, a product family. On the other hand, the variation mechanisms in Koala seem to model behavioural variety of software embedded in a physical product instance: e.g. a television set can behave differently depending on some parameters. Of course, it could be argued that the television set in our example is, in fact, a product family. Nevertheless, we consider the variation mechanisms discussed above examples of different phenomena.

5 MODELLING SOFTWARE ARCHITECTURE WITH THE CONFIGURATION ONTOLOGY

In this section, we strive to synthesize the configuration ontology with the domain of software architecture. We do this by mapping the concepts in the ADLs to some concept or concepts in the configuration ontology. Components, ports, properties, and constraints are represented in the obvious manner using their direct counterparts, whereas the representation of connectors and roles is more problematic. Hence, we will present a mapping of connectors to components, and provided and required interfaces to ports with the aid of type specifications.

5.1 Modelling connectors as a type of component

In translating the semantics of connectors in Acme and Wright into concepts in the configuration ontology, it helps to observe that components and connectors have structures very similar to each other. Therefore, it is natural to view connector as a subtype of component with special semantic constraints. Indeed, defining connector to be a subtype of component will enable us to express part of the semantics associated with connectors. Furthermore, we can define roles in connectors to be ports in the connector-type components. To enforce the right use of connectors, we define suitable constraints that enforce the right use of connectors: e.g. in Wright, the only class of allowed connections is that between a component and a connector.

Subtyping can also be used for distinguishing provided and required interfaces from one other. By defining common supertypes for provided and required interfaces it is possible through multiple inheritance to have two versions of each port type, a provided and a required. By using constraints it is possible to assert that invariants concerning provided and required interface types hold. For instance, the fact that in Koala a required interface must be connected to exactly one provided interface of the same interface type can be easily captured using constraints.

5.2 Capturing diversity

Internal diversity of Koala can be captured with attributes defined by components and constraints. Dependencies between different parameters can be captured using constraints between attribute values of component types.

In the configuration ontology, cardinality of a port defines the amount of ports that can be connected to it. Cardinality can be used to capture some aspects of structural diversity in Koala. By defining cardinality greater than one for a port representing a required interface, multiple provided interfaces represented as ports could be connected to that port. This is only a partial solution as it says nothing about deciding which ports should actually be connected; constraints can be used to model this.

6 EXTENSIONS NEEDED FOR MODELLING SOFTWARE ARCHITECTURE

Albeit the configuration ontology captures a major part of aspects of all the studied ADLs, each of them has some features the modelling of which would require extending the ontology.

Capturing all of the idea behind **heuristic constraints** of Acme may require adding some method of representing optimization criteria and preferences in the configuration ontology.

There is no mechanism in the configuration ontology for **modelling behaviour** similar to the way how CSP is used in in Wright. In fact, the configuration ontology ignores behavioural aspects entirely. In case considering behaviour should be required in the configuration ontology, it would be natural to extend the constraint language to cover behavioural aspects, as the constraint language can be seen as the extension mechanism of the ontology.

Koala includes the method of **function binding**, in which the constituent functions of interfaces are connected directly to each other instead of connecting interfaces [10]. This construct gives an internal structure to Koala interfaces. Given that interfaces of Koala are modelled with ports in the configuration ontology, this contradicts with the underlying assumption of ports being undividable connection points. As a result, there is a mismatch between interfaces in Koala and ports in the configuration ontology.

There is a number of possible ways to capture ports with internal structure. The first one is to make Koala functions the basic level of connection. Unfortunately, this approach introduces major problems. Firstly, interfaces would lose their counterpart in the configuration ontology. Secondly, applying the approach would likely lead to increased complexity in models of software products: the fact that an interface can contain several functions implies this.

The second approach would be to introduce compositional structure for ports of the configuration ontology. Applied to the problem at hand, interface types correspond to port types that have ports corresponding to functions as their parts. This approach is appealing: it models the relation between interfaces and functions in a way corresponding to the intuitive understanding of the issue. This approach would require major changes to the ontology, however.

Binding of interfaces of a compound component with the interfaces of the inner parts is another feature of Koala lacking a counterpart in the ontology. It seems that the ontology would need to be extended in order for it to model this phenomenon.

7 DISCUSSION AND COMPARISON WITH PREVIOUS WORK

There is an apparent difference in the natures of the sets of product variations modelled in different disciplines. In the configuration domain, this set is typically termed as configurable product or a product family. One of the defining characteristics of this concept is a pre-designed general structure with a lot of variation in the configurations [17]. On the other hand, above it was found that Koala supports no common structure for a set of products. In fact, Koala is not targeted at modelling a product family or a set of them, but product populations, which are defined as a set of products with many commonalities but also with many differences [9]. Hence, the underlying aims of the ADL modeling and configuration modelling are not totally similar.

In the previous section it was stated that no satisfactory mapping could be found for function binding of Koala. One possibility to respond to this and similar problems is to ignore the problematic feature. Even though ignoring aspects of ADL that are of practical or theoretical importance is nothing we would do light-heartedly, we still believe that doing so in some cases will increase the usefulness of the configuration ontology in modelling software products. Therefore, the question is: which features of ADLs should be modelled. This is a question that can only be fully answered by empirical research into the nature of software product families.

Research closely related to this paper has been conducted earlier. We do not know of earlier attempts of comparing the concepts of software architecture description to those of configuration modelling. This is the main contribution of our paper.

In their work, Männistö et al. have pointed out the existence of the research area of configurable software and identified some key concerns in the area [18]. They have not, however, studied the concepts of ADLs in detail or proposed any mapping from these concepts to those of configuration modelling domain.

On the other hand, [19] presents a formalized software configuration management (SCM) ontology. The concepts of the SCM ontology are, however, different from those of the configuration ontology. They are aimed at representing the modules, files, or packages, their versions and the dependencies between these. The ontology does not take into account the connections and interfaces between components of a system.

Felfernig et al. have proposed a scheme for constructing configurators based on UML descriptions of configuration knowledge [12]. Basically, their approach could be used for creating configurators for software products as well. Their approach is, however, different from our approach: theirs is based on presenting configuration knowledge in UML, while our approach is based on modelling software with the concepts of product configuration.

In [20], Kühn has presented an approach to software configuration based on structure and behaviour. He uses statecharts, a method similar to finite state machines, for specifying the behaviour of a module. This approach is similar to Wright in that it describes both structure and behaviour. With its focus on using behavioural constraints for making decisions during the configuration process, this approach is different from ours.

8 CONCLUSIONS AND FUTURE WORK

Above, we have presented an analysis of three ADLs and compared their concepts to a conceptualization of configuration knowledge. The aim has been to find a mapping from the concepts of ADLs to those of the configuration ontology. Our goal is to use configuration ontology and its supporting toolset for configuring software.

We found counterparts and close correspondences in the configuration ontology for the main elements of the ADLs we have studied and were able to propose a mapping between them that shows that configuration languages can be used for representing architectural knowledge. For instance, both share the notion of components. Furthermore, compositional structure, systems formed of connected components and constraints are phenomena present in both disciplines. Hence, it seems that the concepts of the configuration ontology can be used for modelling software products. However, capturing some aspects of ADLs seems to require extending the configuration ontology. These aspects include function binding and binding the connection points of compound components with connection points in its inner parts. Another important aspect is modelling behaviour. Of the ADLs, Wright models behaviour. Additionally, the approach presented by Kühn also emphasizes behaviour [20]. The question whether behavioural aspects really are important and should be modelled when configuring software product families, should be resolved through empiric studies. The existence of Koala, a commercial ADL with no behaviour modelling, suggests that modelling behaviour is not absolutely necessary.

There are still open questions and a need for further work. It is necessary to define the mapping of the ADL concepts to the configuration ontology more rigorously. Moreover, an ontology and a configuration language for software products should be defined. This will probably require investigating more thoroughly the current ADLs and the conceptualizations of disciplines such as SCM, generative and feature based programming [16,21], and, of course, the developments in the UML community, as well as case studies of real software product families. After completing this, case studies are needed to verify the applicability of the configuration language to modelling software. Another issue to be concerned is the computational complexity of configuring software products. Theoretical complexity analysis can provide insight into this issue, but only experiments with real products will give relevant information on the practical feasibility from this point of view. When moving towards empirical studies, it is also necessary to consider which of the existing configurators and their modelling languages best support software configuration at a more detailed level than in this study.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Academy of Finland (project 51394) and National Technology Agency of Finland (Tekes).

REFERENCES

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

[2] S. Vestal, *A Cursory Overview and Comparison of Four Architecture Description Languages*. Technical Report. Honeywell Systems & Research Center, 1993

[3] N. Medvidovic and R. M. Taylor, 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Transactions on software engineering*, **26**, 70-93, (2000).

[4] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: An Architecture Description Interchange Language', *in: Proceedings of CASCON'97*, 1997.

[5] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: Architectural Description of Component-Based Systems', *in: Foundations of Component-Based Systems*, Cambridge University Press, 2000.

[6] R. T. Monroe, D. Garlan, and D. Wile. *Acme Reference Manual*. Available at "http://www-2.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/ACME%20StrawManual.html". Cited March 11, 2002

[7] R. Allen and D. Garlan, 'A Formal Basis for Architectural Connection', *ACM Transaction on Software Engineering*, (1997).

[8] R. Allen, A Formal Approach to Software Architecture. Doctoral dissertation, 1997.

[9] R. van Ommering, 'Configuration Management in Component Based Product Populations', *in: Proceedings of Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

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

[11] T. Soininen, J. Tiihonen, T.Männistö, and R.Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM*, **12**, 357-372, (1998).

[12] A. Felfernig, G. Friedrich, and D. Jannach, 'UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**, 449-469, (2000).

[13] D. Garlan, 'Software Architecture', *in: Encyclopedia of Software Engineering*, J. J. Marciniak, ed. John Wiley & Sons, 2001.

[14] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[15] D. Garlan and D. E. Perry, 'Introduction to the Special Issue on Software Architecture', *IEEE Transactions on software engineering*, (1995).

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

[17] J. Tiihonen, T. Lehtonen, T. Soininen, et al, 'Modeling Configurable Product Families', *in: Proceedings of the 12th International Conference on Engineering Design (ICED'99)*, 1999.

[18] T. Männistö, T. Soininen, and R. Sulonen, 'Product Configuration View to Software Product Families', *in: Proceedings of Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

[19] T. Syrjänen, 'Including Diagnostic Information in Configuration Models', *in: Proceedings of the First International Conference on Computational Logic*, 2000.

[20] K. Kühn, 'Modeling Structure and Behavior for Knowledge-Based Software Configuration', in: *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling, and Design*, 2000.

[21] C. Prehofer, 'Feature-Oriented Programming: A Fresh Look at Objects', *in: Proceedings of ECOOP'97*, 1997.