Timo Asikainen

Representing Software Product Line Architectures Using a Configuration Ontology

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology

Espoo, August 27th, 2002

Supervisor: Professor Timo Soininen, Helsinki University of Technology
Instructor: Professor Timo Soininen, Helsinki University of Technology

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF THE MASTER'S THESIS

Author :	Timo Asikainen	
Title:	Representing Software Product Line Architectures Using a Configuration Ontology	
Date:	August 27th, 2002	Number of pages: 134
Department:	Department of Industrial Engineering and Management	
Professorship:	T-106 Information Processing Science	
Supervisor:	Prof. Timo Soininen	
Instructor:	Prof. Timo Soininen	

In the software engineering domain, software product lines have recently been identified as a form of software reuse where a fixed set of assets is used to produce a large number of variations of a single product. The set of assets typically includes software components and a software architecture or a number of them, into which the components are incorporated. Architecture Description Languages (ADLs) are notations used for formally or semi-formally describing the architecture of software systems.

Product configuration, on the other hand, is a research domain where the focus has been on modifying the general design of a product in prescribed ways to produce product individuals that match the specific needs of customers. A product that lends itself to configuration is termed a configurable product.

This thesis studies the possibility of representing software product line architectures using a configuration ontology, i.e., a conceptualisation of the knowledge people have on configurable products. In more detail, the thesis focuses on analysing and comparing the concepts of ADLs and a configuration ontology, and constructing mappings between individual ADLs and the configuration ontology. The underlying objective is to utilise the configuration ontology, configuration modelling languages built on it, and tool support developed for the configuration languages in developing, modelling, managing, and creating instances of software product lines.

It was found that ADLs and the configuration ontology share much of their concepts: many of the concepts of the ADLs are present in the configuration ontology as well, and these concepts could easily be mapped into the configuration ontology. Many other concepts could be adequately represented using the concepts of the configuration ontology. However, some concepts of the ADLs could not be found a satisfactory mapping to the configuration ontology; schemes for extending the ontology with these concepts are suggested. On the other hand, some concepts of the configuration ontology were not found a counterpart in any of the studied ADLs. Some of these concepts have been considered useful in modelling software, but have not been included in ADLs.

Keywords:	Software product line, Product configuration, Architecture Description Language (ADL), Configuration ontology

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä :	Timo Asikainen	
Työn nimi:	Ohjelmistotuotelinjojen arkkitehtuurien esittäminen konfigurointiontologian avulla	
Päiväys:	27.8.2002	Sivumäärä: 134
Osasto: Professuuri:	Tuotantotalouden osasto T-106 Ohjelmistotekniikka	
Valvoja: Ohjaaja:	Prof. Timo Soininen Prof. Timo Soininen	

Ohjelmistotuotelinjat ovat ohjelmistotuotannon muoto, jossa samoista, olemassa olevista tuotanto-hyödykkeitä tuotetaan suuri joukko variaatioita samasta perustuotteesta. Käytettäviin hyödykkeisiin kuuluu tyypillisesti joukko ohjelmistokomponentteja sekä yksi tai useampi ohjelmistoarkkitehtuuri. Ohjelmistoarkkitehtuurien kuvauskielet (ADL) ovat formaaleja tai puoliformaaleja menetelmiä ohjelmistoarkkitehtuurien kuvaamiseen.

Konfigurointi on tutkimusalue, jossa tutkimuksen kohteena ovat konfiguroituvat tuotteet. Konfiguroituvan tuotteen tunnusmerkkinä on se, että yleistä tuotetta moukataan yksittäisten asiakkaiden vaatimusten perusteella rutiininomaisesti siten, että syntyy asiakkaiden vaatimuksia hyvin vastaavia tuote-yksilöitä.

Työssä tutkittiin mahdollisuutta esittää ohjelmistotuotelinjojen arkkitehtuureja käyttäen konfiguroitavien tuotteiden kuvaamista varten kehitettyä konfigurointiontologiaa. Taustalla oleva tavoite oli käyttää em. konfigurointiontologiaa, sen käsitteistöä käyttäviä konfigurointikieliä ja näitä tukevia työkaluja ohjelmistotuotelinjojen kehittämiseen, mallintamiseen, mallien hallintaan sekä yksittäisten ohjelmistojen tuottamiseen. Tutkimusmenetelmä oli konstruktiivinen: ADL:ien ja konfigurointiontologian käsitteistön analysoimisen ja keskinäisen vertailun jälkeen esitettiin konstruktio, jossa näytettiin miten ADL:ien ja konfigurointiontologian käsitteet ovat kuvattavissa toisiinsa.

Työssä selvisi, että ADL:ien ja konfigurointiontologian käsitteistöillä on paljon yhtäläisyyksiä: monia ADL:ien käsitteitä vastaa konfigurointiontologiassa samanniminen ja -sisältöinen käsite. Lisäksi joukko ADL:ien käsitteitä voitiin tyydyttävällä tavalla esittää konfigurointiontologian käsitteiden avulla, vaikka käsitteiden välinen vastaavuus ei ollutkaan täydellinen. Kaikkia ADL:ien käsitteitä ei kuitenkaan onnistuttu esittämään konfigurointiontologian avulla; näiden käsitteiden esittämiseksi luonnosteltiin laajennuksia ontologiaan. Toisaalta konfigurointiontologiassa havaittiin olevan joukko käsitteitä, joille ei ole vastinetta ADL:issä, mutta joiden on kirjallisuudessa todettu olevan tärkeitä ohjelmistoja mallinnettaessa.

Avainsanat:	Ohjelmistotuotelinja, Konfigurointi, Ohjelmistoarkkitehtuurien kuvauskieli (ADL), Konfigurointiontologia

Table of Contents

1 Introduction

1.1 Background

An apparent trend in the developed world is the increasing significance of software. Today, software is not only used in general purpose computing devices, such as desktop and laptop computers and servers, but also in a wide range of devices in the form of embedded software. Examples of this type of devices include consumer electronics, cars, and telecom switches. Further, the number and range of types of handheld devices using software is drastically increasing: handheld computers are no longer merely toys of top executives, and mobile phones have surely solidified their position in our daily lives.

Another trend paralleling the one described above is the growing demand for variability of software, i.e., software systems must lend themselves to be customised to match a range of different requirements instead of a single set of requirements. Albeit consumers might be happy with the standard web browser and word processor, there are other, strong drivers for the variability of software. First, when software is embedded in concrete products, software is often seen as a potential means for varying the product. Second, the software running on e.g. handheld devices must be flexible with respect to the individual needs of users: the scarcity of resources such as disk space, memory, and computing power, requires that the software running on these devices exactly matches the needs of each user, as matching the needs of all users simultaneously is not possible with the given resources. Further, variability is not only required with respect to the static likings of the users, but also with respect to the coming and going needs of the customer and the changes in the environment where the device is located.

The currently dominating approach for software development is to produce a single system that covers the needs of an entire range of customers, or to modify such a single system to correspond to the individual requirements of a customer. Unfortunately, this approach is hardly applicable today, and surely unable to match the requirements of the future (Thiel et al. 2002). First, the cost and development effort following from the use of this approach are impossible to tolerate. Second, the approach is incapable of addressing the requirement of varying the system at runtime based on changes in user needs and environment in which the device is located.

On the other hand, the issue of achieving variability has been studied in the domain of *traditional products*, i.e. mechanical and electrical ones. This domain of research is called *product configuration*, and it pertains to modifying the general design of a product in prescribed ways to produce product individuals that match the specific needs of customers. Until lately, configuring software has not attracted much interest. But today the issue has been acknowledged in the software engineering domain as well, and the most systematic of software product lines come quite close to configurable products. (Bosch 2000; Clements et al. 2001).

Applying the results from configuration of traditional products likewise to software is an interesting research direction. The underlying assumption behind the idea is that although very different in nature, there are also enough similarities between software systems and traditional products that enable applying the same methods used with traditional configurable products to software product lines as well. These methods are based on computer support and cover the entire lifecycle of products from developing the products to describing, managing, and generating instances of them.

However, the above-mentioned assumption is indeed a mere assumption as long as there is no research supporting its validity. In order to apply the techniques developed for configurable products to software product lines as well, software product lines must lend themselves to be described with the same concepts as configurable products: if traditional and software products can be described in the same terms, and the tools processing the descriptions make no extra assumptions, the techniques developed for traditional configurable products could indeed be applied to software product lines as well.

This thesis studies the existence of a conceptual basis shared by software and traditional products, and if no such basis seems to exist, the possibilities for creating one are studied.

1.2 Research Problem and Goals

The research problem is defined in the form of research questions as follows:

1. One of the methods used for modelling software is architecture description languages (ADLs). What are the concepts for modelling software used in ADLs?

How do they compare to the concepts used for modelling traditional configurable products?

2. Given the set of concepts above, is there a viable mapping between these two sets of concepts, i.e., can the concepts used for modelling software systems be reduced to the set used for modelling traditional configurable products, and vice versa? If there is at least a partial mapping, what is it in detail?

3. If the mapping described in the second research question is a partial one, how should the set of concepts used for modelling traditional products be supplemented in order to enable mapping the entire set of software modelling concepts?

The goal of this thesis is to represent the architecture of a software system using the concepts of the product configuration domain by applying the mapping between the concepts and an ADL specified in the research questions. The representation can happen either by starting from an ADL description and mapping it explicitly to the concepts of the product configuration domain, or by modelling the architecture of the system with a subset of the product configuration concepts that corresponds to the concepts of the ADLs. Thereby, the mappings are not merely an analysis tool, but can additionally be applied for modelling software systems.

Answering the first research question will yield one viable conceptualisation used for modelling software. This conceptualisation provides a point of comparison to be used in answering the second and the third question. Answering the two latter questions will result in achieving the research goal.

In order to achieve the research goals, the mapping mentioned in the second research question should satisfy the following criteria.

1. *Unambiguity*. Each concept is mapped into a single set of concepts, i.e., there are no alternative mappings for a single concept.

2. *Elegance*. The mapping should be as simple as possible, and concepts in the domain for which the mapping is made should not be used in ways contradicting their intended or standard use.

3. *Modularity*. Each part of the domain can be mapped separately to the image.

4. *No explosion in size*. The size of the image of a model in the target domain should be in proportion to the original size of the model.

5. *Easy to understand*. The mapping should be easy to understand, i.e., it should be intuitively clear which elements in the target model correspond to the elements in the original model.

6. *Sufficient level of detail*. The mapping should be detailed enough to serve as a specification for implementation in some suitable programming language.

1.3 Method

The research method followed in this thesis is the constructive method. First, three prominent ADLs are analysed at the conceptual level; second, the concepts of the ADLs are compared with each other, and with the concepts of the product configuration domain; third, it is shown how architecture-based descriptions of software systems can be modelled using the configuration concepts by constructing a mapping between each ADL and the configuration concepts; and finally, an analysis will be presented on which concepts of the ADLs could and could not be found a mapping, along with suggestions on what could be done with the concepts that could not be found a mapping.

The domains involved in the research problem are analysed in a literature study. These domains are the product configuration domain and software product domain, and they are illustrated in Figure 1. As the research problem concentrated on the concepts of the two domains, the focus should lie on the conceptualisations of both of the domains. However, as can be seen in the figure, no conceptualisation of the software domain exists. Therefore, the focus in this domain is at a lower level of abstraction, which is the language level; this is illustrated with a thick border. On the other hand, there are ready conceptualisations, i.e., configuration ontologies, of the product configuration models. Of the conceptualisation, a specific, comprehensive, and widely accepted configuration ontology will be utilised. Hence, the focus in the domain is on a configuration ontology.

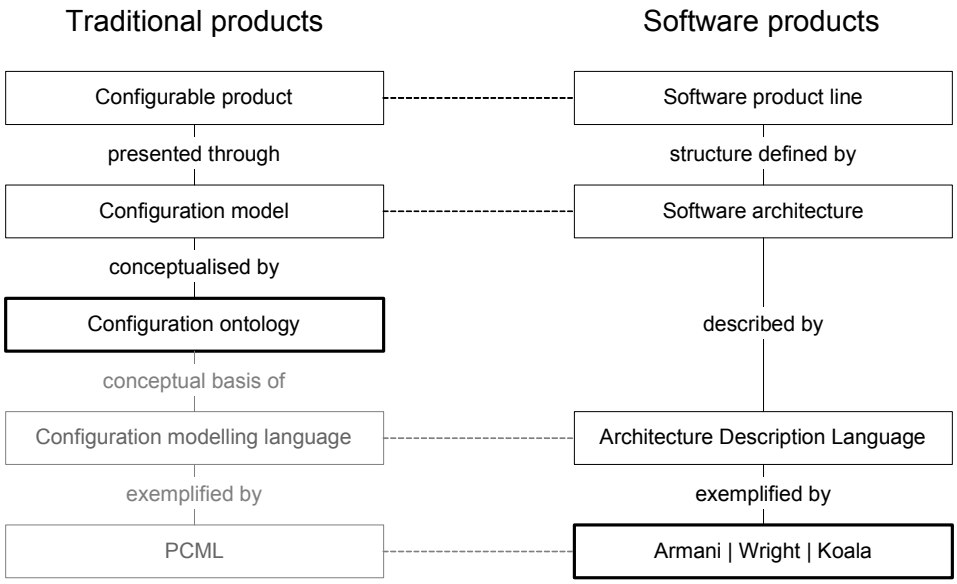


Figure 1 The domains of interest of the thesis

The construction is given by defining a mapping between the concepts in the three ADLs and those in a conceptualisation of configuration knowledge. In more detail, the mapping will consist of three parts. Firstly, it is shown how the concepts of each ADL can be mapped into the configuration ontology. This part of the construction supports mapping ADL descriptions into the configuration ontology. Secondly, constraints that limit the modelling facilities to those paralleled in the ADL are defined. This part of the construction supports using the configuration ontology and its supporting toolset for producing ADL-like software architectures. Thirdly and finally, it is shown how models in the configuration ontology, constrained by the constraints specified in the second part of the construction, can be mapped into the ADL.

The validity of the construction is established by the mapping satisfying the criteria of the above section: explicitly, the mappings are detailed enough and easy enough to understand for the reader to convince her of the validity of the mappings. Further, the validity of the mappings is demonstrated though an imaginary example.

1.4 Scope

The architecture is by no means the only relevant aspect of a software product line. However, architecture is a central issue in the product line approach to software engineering. Therefore, concentrating on architecture is a viable alternative to study software product lines.

ADLs are not the sole possibility for describing software architectures. Nevertheless, software architectures are generally considered the most sophisticated method of modelling software architecture. Further, there are many other ADLs besides the three studied in this thesis (Medvidovic et al. 2000). However, the studied ADLs are representative of the entire class of ADLs.

The studied ADLs include aspects that are not relevant to configuring software product lines. These aspects are mostly ignored in this thesis, as the emphasis of the thesis is on configuring software product lines.

There are other conceptualisations of configuration knowledge than the configuration ontology chosen for this thesis. However, the chosen ontology is both representative of other conceptualisations and often cited in the configuration literature.

1.5 Outline

The rest of the thesis is organised as follows. In Chapter 2, an overview of issues related to configurable products is given and a conceptualisation of the domain is presented. The focus is shifted to software systems in Chapter 3, in which software product lines are introduced, and thereafter ADLs as a method for describing their architecture are studied in some detail. In Chapter 4, the concepts for describing traditional configurable products and software systems are compared. Chapter 5 presents a synthesis in which mappings between each ADL and the configuration ontology domain are presented. Chapter 6 contains the discussion part of the thesis: the results for the mapping are reiterated and assessed for validity; potential extensions to the conceptualisation of configuration knowledge are suggested; and general topics concerning the thesis are discussed. A comparison between this thesis and previous work is carried out in Chapter 7. Chapter 8 presents projections for future work. Conclusions in Chapter 9 round off the thesis.

2 Product Configuration

Traditionally, there has been a difference in the sales-delivery processes for inexpensive consumer goods, such as televisions, computers and mobile phones, and expensive capital goods, such as paper machines, oil rigs and rock drilling machines. The predominant paradigm for producing consumer goods has been *mass production*: vast volumes of identical product individuals have been produced in large batches or using assembly lines. On the other hand, capital goods have in many cases been *one-of-a-kind products*: each product individual is designed based on customer requirements and thereafter manufactured.

During the last couple of decades, a trend changing the situation described above has emerged. Many companies that have been applying the mass production paradigm have noticed that customers are increasingly willing to pay for customised products, i.e. products that match some or all of the individual requirements of the customers. Elsewhere, the manufacturers of one-of-a-kind products have noticed that standardising parts of their range of products can lead to significant cost savings and improved quality, as the need for design work per manufactured product decreases and the lessons learned from the design of previous products can be efficiently utilised to avoid mistakes in the design process. (Pine 1993)

Hence, the trend can be seen as two manufacturing paradigms, namely mass production and one-of-a-kind products, converging towards each other (Tiihonen et al. 1997). What is in between the two old paradigms can be seen as a new paradigm, often described with the term *mass customisation* (Pine 1993). In short, mass customisation can be considered as a new paradigm replacing the old mass production paradigm. The main idea of mass customisation is to combine the good sides and benefits from mass customisation and one-of-a-kind products. More concretely, mass customisation seeks to provide all the customers with products matching their requirements with no overhead cost or increased lead-time compared to mass-produced products.

There are many methods for achieving mass customisation. One of the most important ones is based on modular product design: products are designed in a way that allows a number of constituent components of the product to be replaced with another variant. (Pine 1993). Combinational explosion results in very large numbers of variants for products containing many points where alternative components can be placed.

2.1 Configurable Products

The abstract idea of providing mass customisation through modular product design has been concretised in the concept of *configurable product*. But before giving an exact definition for this concept, some auxiliary definitions are needed.

> A *product* is defined as an abstract specification or design of an entity that a company sales. A *product instance* is a physical product that is to be delivered to a customer or a design of a physical product, which is concrete enough to serve as a specification for manufacturing. (Tiihonen et al. 1997)

A configurable product is defined as a product with specific characteristics.

> A *configurable product* is a product that comprises a large number of variants and serves the specific needs of the individual customer by allowing customer-specific adaptation of the product. A configurable product has the following characteristics: (Tiihonen et al. 1997)
>
> - Each delivered product instance is tailored to the individual needs of an individual customer.
> - The product has been pre-designed to meet a given range of different customer requirements.
> - Each product instance is specified as a combination of pre-designed components or modules. Thus, there is no need to design new components as a part of the sales-delivery process.
> - The product has a pre-designed general structure or architecture or a set of these.
> - There is no need to do creative or innovative design as a part of the sales-delivery process. Rather, the specification of a product instance can be done in a routine manner.

A configurable product is defined by a configuration model. Further, configurators are defined to be produced by completing configuration tasks:

> *Configuration model* defines the basic product properties and the possibilities for tailoring them. Configuring a product on the basis of customer requirements

> produces a *configuration*, a description of the product instance to be delivered. The task needed to produce a configuration is referred to as *configuration task*. (Tiihonen et al. 1997)

Examples of configurable products include personal computers from the computer industry, switching systems from the telecommunications industry, and passenger cars from the automotive industry (Felfernig et al. 2001).

In the process of producing configurations, product configurators may be of help:

> *Product configurator* is an information system for managing products and their variants, and for doing customer-specific adaptation of the product. (Tiihonen et al. 1997)

Putting pieces together, configurable products form an approach to satisfying variable customer needs. The approach consists of defining a product through a configuration model; a product configurator is used to support the process of creating and maintaining the configuration model. Thereafter, configurations of the product matching specific needs of a customer are formed through the configuration task using a product configurator. What is essential is that the effort required to create a configuration of the product is moderate. Especially, the definition of configurable product implies that no need for creative design exists.

2.2 Conceptualisations of Configuration Knowledge

The definition of configurable product and the examples from the above section gives an intuitive understanding of what configurable products look like and the ability to decide whether a given product is configurable. However, the definition does not specify any means for specifying configurable products. In the following sections, an approach for conceptualising knowledge about configurable products is presented.

2.2.1 The Configuration Ontology by Soininen et al.

Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen have presented in (1998) a conceptualisation of *configuration knowledge*, i.e. the knowledge that people have on configurable products. In the ontology, configuration knowledge is divided into three classes.

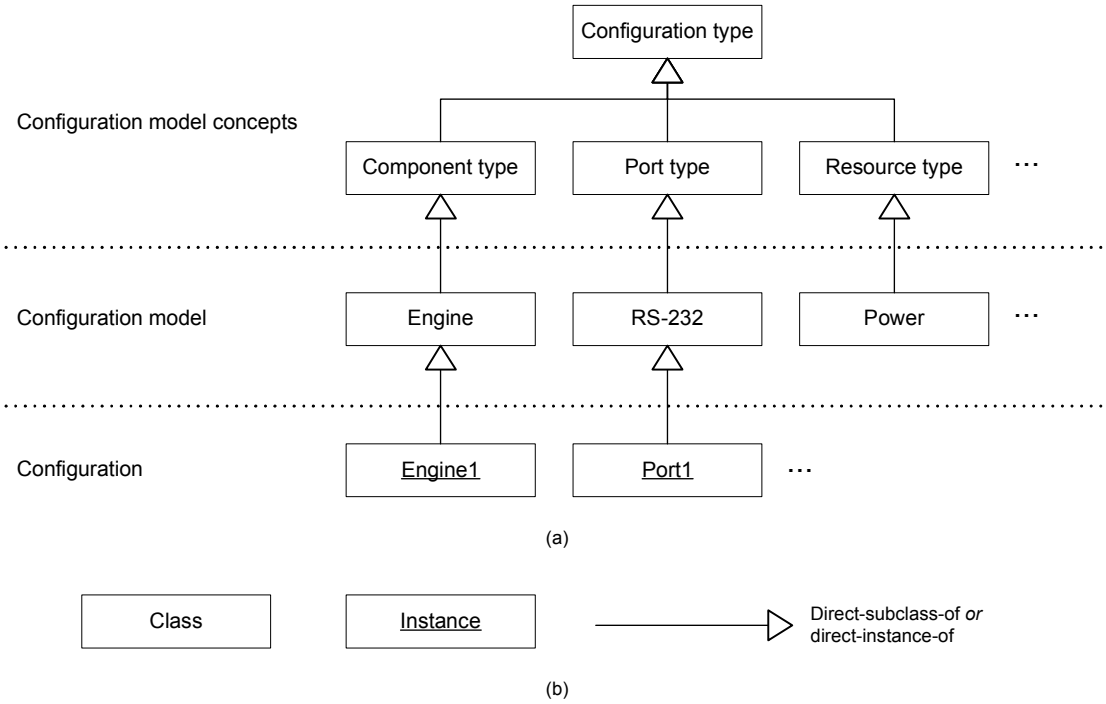


Figure 2 The basic structure of the ontology. Adapted from (Soininen et al. 1998).

Configuration model knowledge specifies the entities that may appear in a configuration, their properties, and the rules on how the entities and their properties can be combined.

Configuration solution knowledge specifies a (possibly partial) configuration.

Requirements knowledge specifies requirements on the configuration to be constructed.

The basic structure of the ontology is presented in Figure x. At the top, *configuration model concepts* consist of a set of *configuration specific classes* and *configuration specific relation definitions*. The qualifier 'configuration specific' refers to the fact that the sets of classes and functions are specific to the domain of configuration, not to a configuration as defined above. On a level immediately below in the figure, a set of *product specific classes* are defined as direct subclasses of the configuration specific classes. These classes are termed *types* in the ontology, and they form a *configuration model* of a product. Proceeding further down in the figure, a *configuration* is defined as a set of instances of the types occurring in the configuration model. These instances are termed *individuals* in the ontology.

It should be noticed that in the ontology, types occur in configuration models and individuals in configurations. In other words, these two kinds of entities are strictly separated from one other.

Taxonomy

In the ontology, types are organized into classification taxonomies using ISA-relation. In common terms, the ISA-relation defines an inheritance hierarchy for the types. The terms *direct subtype* and *direct supertype* are related to the ISA-relation in a natural way: for types T_1 and T_2, (T_1 ISA T_2) implies that T_1 is a direct subtype of T_2, and T_2 is a direct supertype of T_1. Similarly, the terms *subtype* and *supertype* can be defined in the terms of the transitive closure of the ISA-relation. Each individual is directly of exactly one type, and indirectly of all the supertypes of that type.

Configuration type is a common superclass for types occurring in a configuration model. They can be defined *attributes*.

Structure

An important aspect of a product, configurable or other, is its *structure*. Informally, the structure of a product is defined in terms of its constituent components. In the configuration ontology, *Component type* is a subclass of configuration type. A component type, i.e. a subclass of Component type, "represents a distinguishable entity in a product that is meaningful for product configuration in the sense that a configuration is composed on *component individuals* of component types" (Soininen et al. 1998).

Component types can be defined to include other component types as parts. This is conceptualised by the two component individuals of the types being in HAS PART-relation with each other.

Topology

Besides structure, an important aspect of a product is the connections between component individuals in a configuration.

A *port type* is an intensional definition of a connection interface. Port type is a subclass of configuration type. Each port type defines a set of port types whose port individuals can be connected to the port individuals of the port type in its *compatibility definition*. A component type specifies its connection possibilities by *port definitions* that are

conceptualised as component individuals and port individuals being in HAS PORT-relation.

Two component individuals that have port individuals of compatible port types may be connected to each other through their port individuals. The two port individuals are thereby in CONNECTED TO-relation. If any two ports of two component individuals are in the CONNECTED TO-relation, the component individuals containing the ports are in the CONNECTED TO-relation as well.

Resources

Resources are used in the ontology to model the production and use of entities, or the flow of such entities from one component individual to another.

Resource type defines the properties of a resource. Resource type is a subclass of configuration type. An individual of a resource type is called a *resource production individual*. A resource type specifies a computation definition that specifies whether the resource production or use must be *satisfied* or *balanced*. If the production or use is to be balanced, the production must exactly match the use; in the case they must be satisfied it is enough that production is greater than or equal to use. The computation is done in a *context*. A context is a set of component individuals that can contain all the component individuals in a configuration, or any subset of these.

The production and use of resources is specified by *production definitions* and *use definitions* in component types. These definitions specify the resource types and quantities produced or used.

Functions

The concepts of the ontology presented so far have been *technical concepts*. However, an important aspect of any product is the functions it provides to its users. For capturing this aspect, the ontology includes concepts for representing *functions*.

Function type is a subclass of configuration type. A function type is an abstract characterisation of the product that a customer or sales person would utilise to describe the products. An individual of a function type is called *function*. Function types can specify parts in a similar manner as components. Hence, functions can be composed of other functions, likewise components of other components. *Implementation constraints* can be used to specify the relation between technical concepts, i.e., the concepts of the

ontology discussed so far, and functions. E.g., an implementation constraint can specify that a function is implemented by a set of component individuals.

Constraints

Constraints can be used to capture aspects of products that cannot be reasonably modelled using the concepts of the ontology presented above. A constraint is a formal rule specifying a condition that must hold in a correct configuration. They can be expressed using mathematics and logics. Constraints can specify arbitrary interactions of types, individuals, and their properties. Formally, constraint is seen as a constraint instance that defines a constraints expression.

The ontology itself does not specify a language for specifying constraints, but assumes that such a language exists. Above, some special forms of constraints have already been mentioned: examples of these include connection constraints and implementation constraints.

2.2.2 Other Conceptualisations

Besides the ontology described above, there are other conceptualisations of configuration knowledge. The most prominent of these is used by an Austrian research group, including e.g. Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach as members (see, e.g. (Felfernig et al. 2001)). A comparison between these two conceptualisations reveals striking similarities. The concepts included and relations between them in the conceptualisation used by the Austrians are essentially the same as in that presented by Soininen et al: components, functions, resources and ports are the main concepts of the Austrian conceptualisation. Further they are related to each other in a way similar to that in the ontology by Soininen et al.: components can have other components as parts through aggregation, components can be connected to each other through ports etc. Finally, the Austrian research group refer in their work to the work by Soininen et al. as the basis of their own conceptualisation (Felfernig et al. 2001). Thereby, the two conceptualisations are indeed very close to each other.

2.2.3 Discussion

Historically, four approaches to configuration have been proposed. These include resource-based, connection-based, structure-based and function-based approaches. The

fact that the ontology by Soininen et al. has been designed to synthesise these four approaches suggests that it would be applicable wherever at least one of the previous approaches would be.

The best evidence of the practical applicability of an ontology is its being in wide-spread use in the industry. Unfortunately, there seems to exist no such evidence for the configuration ontology by Soininen et al., or any other conceptualisation of configuration knowledge.

However, there is some evidence of the practical usefulness of the ontology by Soininen et al.: it has been used to model at least one real-world, commercial product, namely a drilling machine (Tiihonen et al. 1998). What is more, the Austrian research group has adopted the ontology almost as such.

In conclusion, there is some evidence of the applicability of the ontology by Soininen et al. in the industry, and a considerable support for it in the academia. Further, there seems to be no conceptualisation of configuration knowledge following an approach radically different from the ontology. Thereby, in the rest of this thesis, the configuration ontology is employed as the conceptualisation of configuration knowledge and is from now on referred to as the *configuration ontology*, or, for brevity, the *ontology*.

3 Software Product Lines

This chapter introduces the concept of software product line. This concept has emerged from the software engineering domain. There are a number of definitions for the concepts. Clements and Northrop from the Software Engineering Institute (SEI) of the Carnegie-Mellon University (CMU) define the concept as follows (2001):

> 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 in his book (2000):

> A *software product line* consists of a product line architecture and a set of reusable component that are designed for incorporation into the product line architecture. In addition, the product line consists or 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. Further, both definitions include the set of products or systems developed using these assets.

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

There are still some minor differences between the definitions. First, the SEI definition includes the requirement missing from Bosch's definition that the systems belonging to the product line are developed "in a prescribed way". The Bosch definition, on the other hand, postulates that the core assets belonging to the product line are designed to "for incorporation into the product line architecture".

Rather than dwelling deeper into the details of the definitions, it is more important for this work to compare them with the characteristics of a configurable product defined

above in Section '2.1 Configurable Products'. This comparison is suggested by the apparent correspondence between configurable products and software product lines.

The comparison discloses many similarities between the concepts. Of the similarities, the most striking is that both software product lines and configurable products are defined to be based on a set of assets: these are termed components in Bosch's definition and in configurable products, and simply assets in the SEI definition. The SEI definition and the definition of configurable product share a market driven aspect: a software product line conforming to the SEI definition satisfies the needs of a specific market segment, whereas a configurable product is defined to satisfy a given range of customer requirements. On the other hand, the notion of an explicit architecture of configurable products is shared by the Bosch definition.

However, some characteristics of configurable product have no counterpart in the definitions of software product line: There is no reference to tailoring systems in a software product line to the individual needs of a customer in either of the definitions. Moreover, it is not explicitly stated that there would be no creative or innovative design as a part of the sales-delivery process of software product lines. On the other hand, the definition of configurable product covers all the relevant aspects of both definitions of software product line.

A viable conclusion from above would be that software product lines are somewhere between products designed based on customer orders and configurable products. Further, the most systematic software product lines come quite close to configurable products (Bosch 2000). Consequently, the same techniques used for configuring traditional configurable products can be likewise used for configuring software; configuring software in the context of software product lines should be understood similarly as configuration task in the context of traditional configurable products: i.e., designing a product individual of the configurable product conforming to some requirements.

Of course, the resemblance between configurable products and software product lines is so far on a very abstract level. Next, software architecture is discussed, as they are the principal level of designing, modelling, and managing product lines.

3.1 Software Architecture

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, the composition of design elements etc. (Garlan 2001). One definition for software architecture is (Bass et al. 1999):

> 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. Explicitly, software architecture does not concern the fine-grained structure or the properties of a software system, or the process used to develop it (Medvidovic et al. 1998). Thereby, software architecture is as such a far cry from a notion covering all aspects relevant to software development.

Of course, in the form defined above, software architecture is an abstract concept. Therefore, methods for describing software architecture in more concrete terms are needed in order to make software architecture useful for any practical purposes.

3.2 Architecture Description Languages

Historically, software architecture has been described with idioms such as 'client-server architecture' and 'pipeline'. In the software engineering community, idioms of this kind have well-understood semantics. (Shaw et al. 1996). Another simple method for describing software architectures is the use of box-and-line diagrams. In these diagrams, computational elements are depicted as boxes and the interconnections between them as lines. Although both of these methods manage to capture some important aspects of software architecture, they have been recognised to be inadequate for the task of software architecture description (Garlan 2001).

Architecture description languages (ADLs) are a promising candidate solution for the architecture description problem. Loosely defined, ADLs are formal notations with well-defined semantics. Their primary purpose is to represent architectures of software systems. A large number of ADLs have been proposed. The greatest common

denominator for the class of ADLs is the concept of computational elements present in all of them; in other respects, they differ from one other radically (Medvidovic et al. 2000).

In the following sections, three ADLs will be discussed in detail. This serves two purposes. First, based on the descriptions of individual ADLs, a picture of ADLs as a method of presenting software architectures can be formed. Second, in later section the possibilities of mapping concepts of ADLs to the configuration ontology are studied using the same ADLs as examples.

The names of entities in the ADLs are typeset in `Courier New` throughout the rest of the thesis.

3.2.1 Armani

This section discusses the concepts of Armani (Monroe 2001) and illustrates them using a running example. Armani is based on another ADL, namely Acme (Garlan et al. 1997; Garlan et al. 2000; Monroe et al. 2002). Acme was created as a joint effort of the academic software architecture community as an architecture interchange language that would serve as an intermediate language when transforming descriptions from ADLs to other ADLs. To support this goal, Acme was given the structural constructs that in the language's designers' point of view were the shared structural core of architecture description. Armani, on the other hand includes all the relevant properties of Acme. Moreover, compared with Acme, Armani is considerable better documented and thus a better object of analysis.

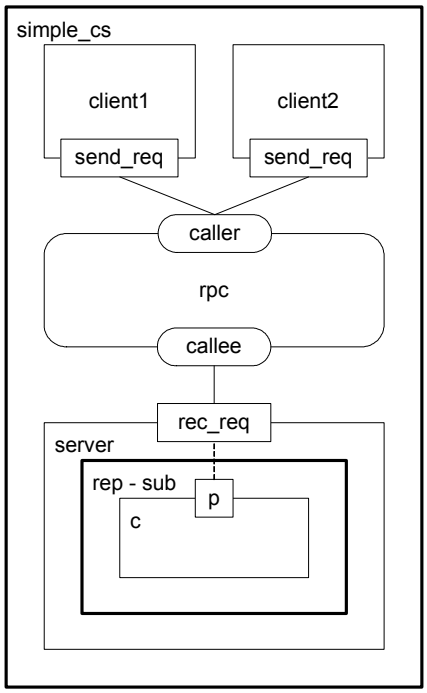
The basic concepts of Armani are components, connectors, ports, roles, systems, properties and representations.

Components and Ports

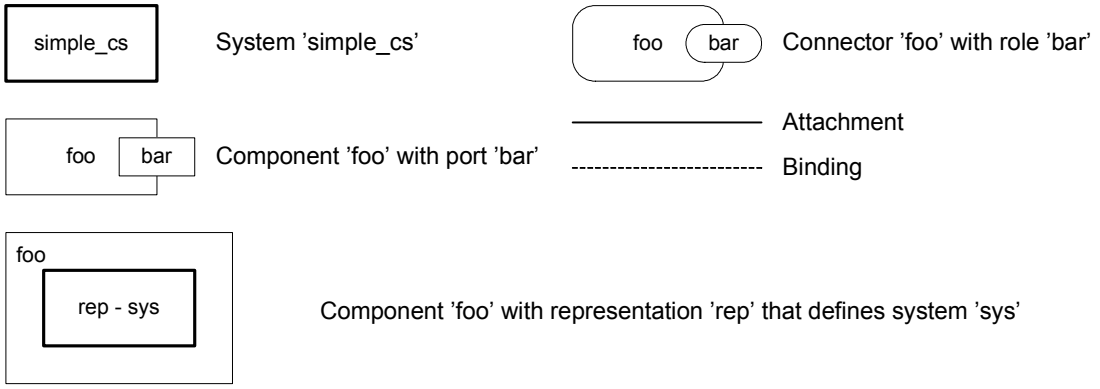
Components represent the primary computational units and data stores of a system. Intuitively, they correspond to the boxes in the box-and-line descriptions of software architectures. Typical examples of components include clients, servers, filters, objects, blackboards, and databases.

```
System simple_cs = {
  Component client = { Port send-request; };
  Component server = {
    Port rec-request;
    Representation rep = {
      System sub = {
        Component c = { Port p; }
      };
      Bindings {
        server.rec_req to server.rep.sub.c.p;
      };
    };
  };
  Connector rpc = { Roles { caller, callee }};
  Attachments {
    client1.send_req to rpc.caller;
    client2.send_req to rpc.caller;
    server.rec_req to rpc.callee;
  };
};
```

(a)



(b)



(c)

Figure 3 A client-server system in Armani. (a) The system represented textually. (b) The system represented graphically. (c) Legend of the notation used.

The interfaces of components are defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment. A port can represent an interface as simple as a single procedure call or more complex interaction, such as a collection of procedure calls that form a communication protocol.

A running example will be used for Armani and the other ADLs discussed the following sections, and in Chapter '5 Synthesis'. The example system is a client-server system with two clients. In abstract terms, the topology of the system consists of the two clients being connected to the server. Further, the server is defined an inner structure. The

structure of the system is described in terms of the concepts of each ADL in the following sections.

Figure 3 illustrates the client-server system in Armani. In Figure 3 (a), there is a textual representation of the system, and in Figure 3 (b) a graphical one. In both versions, components `client_1`, `client_2`, `server`, and `c` are defined. Further, both `client_1` and `client_2` define a port called `send_req`, `server` defines port `rec_req`, and `c` defines port `p`. All the above-mentioned components and ports are illustrated in Figure 3 (b).

The notation is explained in Figure 3 (c). The notation does not follow any standard but is developed for the purpose of this thesis; there is no graphical notation defined for Armani or a general graphical notation used across different ADLs.

Connectors and Roles

Connectors represent interactions and mediate the communication and coordination activities among components. Informally, they correspond to the lines in the box-and-lines diagrams. Examples of connectors include pipes, procedure calls and event broadcast. In addition to these simple forms of interactions, connectors can represent more complex forms of interactions, such as protocols or SQL-links between a database and an application using it.

Like components, connectors have explicit interfaces defined by a set of *roles*. A role in a connector defines a participant in the interaction defined by the connector. In Armani, components, ports, connectors, and roles together are termed *design elements*.

In the example system in Figure 3, only one connector is defined. This connector is `rpc`, and it defines two roles, `caller` and `callee`.

Systems and Representations

Systems represent configurations of components and connectors. To be more exact, the constituents of a system are a set of components, a set of connectors and a set of *attachments* between port and roles. The attachments describe the topology of a system and are defined between ports in components and roles in connectors. Systems are in a sense the base concept of the Armani language: of the basic concepts, only systems can exist independently of other concepts. Further, systems are the object of architecture description and therefore the fundamental class of objects in Armani.

In Figure 3, the example system has the name `simple_cs`. It includes components `client_1`, `client_2`, and `server` and connector `rpc`. The system defines three attachments: the port `send_req` in both `client_1` and `client_2` is attached to the role `caller` in connector `rpc`; port `rec_req` of `server` is attached to role `callee` in the same connector.

To address the need of hierarchical descriptions in software architectures, Armani includes the concept of *representation*. Specifically, any component or connector can be represented by any number of more detailed, lower-level descriptions. Each such description is termed a representation. Basically, a representation contains a system and a set of bindings between parts in the representing system and those in the represented component or connector.

In the rest of this thesis, the term *independent system* is used to refer to a system that is not defined in a representation. Correspondingly, the term *representing system* is used when referring to a system that is defined within a representation.

The use of presentations is illustrated in Figure 3 as well. Component `server` declares the representation `rep`, which defines system `sub`. This system, in turn, defines a single component `c` with port `p`. Additionally, `rep` defines that port `p` of `c` is bound with port `rec_req` of `server`.

Properties

All the basic concepts of Armani (components, connectors, ports, roles, systems, and representations) can be annotated with *properties*. They are named type-value pairs that can include any kind of information about the entity they are located in. Properties can be of an atomic type (integer, float, boolean, string) or have the structure of an enumeration, record, sequence, or set.

Additional Concepts

In addition to the above-mentioned basic concepts, Armani includes a number of other concepts that are second-class concepts in the sense that they bring nothing new into what kind of systems can be described with the language. These concepts will be discussed next.

```
Component Type T = { Port p; };

Component A : T = new T;          Component C = { Port p; };
Component B : T = { Port p; };    Component D : T = {}; // Invalid
```

Figure 4 Declaring and satisfying a design element type in Armani.

Design Element and Property Types

A *design element type* is a set of predicates concerning design elements. Each design element type is associated with one category of design elements, i.e. component, port, connector or role. A type can include two kinds of predicates. First, types can contain predicates about structure and properties. Second, a type can include *invariants* and *heuristics*. An instance is defined to *satisfy* a design element type if it satisfies all the predicates specified by that type. Moreover, instances can be *declared* to be of a type, which implies that a valid instance also satisfies the type.

The predicates about structure and properties have the same form as design element and property declarations. For example, Figure 4 includes a type definition that specifies component type `T`. Type `T` contains one predicate: there exists port `p`. Any component instance satisfying type `T` must satisfy this predicate. Further, Figure 4 includes the declaration of four components, `A`, `B`, `C`, and `D`. Of these, `A` declares type `T`, and also satisfies `T` as it imports the structure defined by `T` with the *new* operator. Instance `B`, on the other hand, does not import the structure, but the same port declaration is repeated in the component declaration; thereby, also `B` declares and satisfies type `T`. On the other hand, component `C` does not declare type `T`, but it still satisfies the type: `C` has the structure specified by the type. Finally, instance `D` declares `T`, but fails to satisfy the type: component `D` has whatsoever no structure. Therefore, component `D` is invalid.

In addition to design element types, Armani includes *property types* as well. A property type specifies a name and a structure, e.g. a float or a sequence of integers. For a property instance in a design element or a system to satisfy a property type, the property instance must have the structure specified by the property type.

Invariants and Heuristics

Invariants and heuristics are predicates defined in Armani Predicate Language. They are boolean-valued conditions concerning the structure and properties of the instances of the type in which they are defined. In order for a design element instance to satisfy an

invariant, the predicate specified in that invariant must be true for that design element. A heuristic is always satisfied, i.e. heuristics impose no constraints on design elements satisfying a type. Informally, heuristics are merely hints of what might be true for an instance of a type. They are included in the language in order to enable the designers using the language to capture rules concerning types that are less strict than invariants.

Subtyping can be used to create new design element types based on existing ones. The subtyping mechanism is familiar from other domains, e.g. UML (Unified Modeling Language) (Object Management Group 2001): the set of predicates of the existing type is inherited by the new type.

Styles and Design Analyses

Analogically to design element and property types, there are also system types termed *styles* or *families* in Armani. In this thesis, only the term style is used for simplicity. Styles bear a close resemblance to design element types, but there are also differences. Namely, design element types consist of required structure, invariants, and heuristics. Thereby, design element types are best viewed as sets of requirements that instances must satisfy. Styles, on the other hand, likewise define invariants and heuristics that the systems of the style must satisfy. However, a style can also include *design analyses*, design element types, and property types.

Design analyses are functions formed by combining the primitive functions defined in the Armani Predicate Language. Design analyses can be used in invariants and heuristics. Thereby, unlike design element types, styles are not merely entities restricting instances but contain also design vocabulary, i.e., design element and property types and design analyses that are best described as options offered to the developer using the style.

Finally, the term *design* is used to refer to a specification in the Armani language that is valid and independent. In other words, when Armani is compared with a programming language, a design in Armani corresponds to a program in the programming language. An Armani design can include design type declarations, style declarations, design analysis declarations, and a system declaration.

3.2.2 Wright

Components, Connectors, Ports, Roles, Systems

Wright (Allen et al. 1997; Allen 1997) is an architecture description language that has been designed to support the formal description of software system. It shares a number of concepts with Armani: as in Armani, there are *components*, *ports*, *connectors*, and *roles* in Wright. These concepts are mutual to the two languages both on the syntactic and semantic level: components are localised, independent computations; connectors represent interaction between components; ports represent the interaction in which components containing them can participate; and roles the behaviours of individual participants in an interaction. Wright allows each port and role to be attached to at most one role and port, respectively, at a time.

Depending on the source, Wright uses the same term *system* (Allen et al. 1997), or a different term *configuration* (Allen 1997) to refer to the same concept as system in Armani, i.e. a configuration of connectors and components with *attachments* between them. In this thesis, the term system is used.

Figure 5 depicts the client-server system used as an example. Previously, the same example was defined in Figure 3. Figure 5 (a) contains a textual representation of the system, and Figure 5 (b) a graphical one. The same notation that was given for Armani in Figure 3 (c) is applicable to Wright as well.

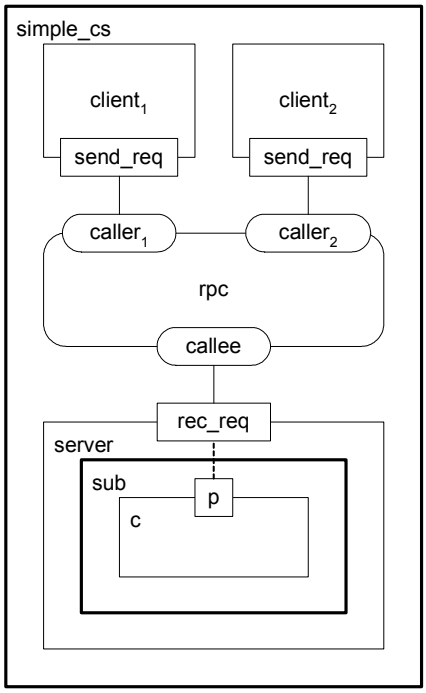
As demonstrated in the figure, Wright distinguishes between component types and instances. Additionally, Wright allows specifying arrays of components. An array by the name `client` of size two is instantiated from `client_type`. Of course, also arrays of connectors, ports, and roles can be specified; role arrays are demonstrated in the figure, as `rpc` specifies an array of size two by the name `caller`.

Behaviour modelling

The most significant feature of Wright lacking in Armani and in many other ADLs is *behaviour modelling*. Again, there is some disagreement between different sources about the extent to which behaviour is modelled. What the sources agree on is the fact that the behaviour of interfaces is described in terms of CSP.

```
System simple_cs = {
  Component client_type
    Port send-request;
    Computatation ...
  Component server_type
    Port rec_req
    Computation
      System sub
        Component c_type
          Port p
      Instances
        c: c_type
      End sub
      Bindings
        c.p = rec_req
      End Bindings
  Connector rpc_type
    Role caller1..2, callee
    Glue ...
  Instances
    client1..2: client_type
    server: server_type
    rpc: rpc_type
  Attachments
    client1.send_req as rpc.caller1;
    client2.send_req as rpc.caller2;
    server.rec_req as rpc-callee;
End simple_cs.
```

(a)



(b)

Figure 5 The client-server system in Wright. (a) The system represented textually. (b) The system represented graphically.

CSP (Communicating Sequential Processes) (Hoare 1985) provides a wide range of constructs for describing communicating entities, and describing all of them in any reasonable high level of detail is impossible in the context of this thesis. However, a short introduction to the topic adapted from (Allen et al. 1997) is presented to give an initial understanding of the issue.

A *process* describes an entity that can engage in communication *events*. An event is an atomic entity that can be informally described as the occurrence of some meaningful, real-world event. The set of events in which a process can engage is termed the *alphabet* of the process and is denoted as αP, where P is the associated process. *Prefixing* can be used to form new processes based on existing ones; e.g. a process that first engages in event e and then becomes process P is denoted $e \rightarrow P$. At some points, a process can behave in a number of different ways. When the choice between different behaviours is made within the same process, the term used is *decision*. If other processes outside the

scope of the process make the choice, the term *alternative* is used. Interactions of processes can be described by the process formed by composing two other processes with the ‖ operator: the process *parallel composition* of P and Q, $P \parallel Q$, is defined to have the alphabet of the intersection of the alphabet of processes P and Q, and it engages in the events of its alphabet when both P and Q would engage in the event.

As mentioned above, in Wright a CSP process is associated in every interface, i.e. port and role. Further, in each connector there is a CSP process called *glue*. In (Allen 1997), all the components are additionally defined a *computation*, a CSP process describing the behaviour of the component. The process definitions are used on various levels and for various purposes in Wright. First, the process descriptions of a port and a role are used to decide if a port and a role can be attached to each other. A port and a role that can be attached are termed *compatible*; compatibility is a static property the value of which can be evaluated based on the CSP processes of a port and a role. Second, the behaviour of an entire connector is characterised by parallel composition of its glue process and all its ports. Third, the behaviour of an entire Wright system is defined to be the parallel composition of all the computations in the system's components and all the glues in the system's connectors. With their behaviours defined, connectors, roles and system can be analysed with respect to different properties, e.g. dead-lock freedom and other safety properties, by using tools processing CSP.

Hierarchical Decomposition

The similarities between Armani and Wright extend still further: Wright provides a mechanism for specifying hierarchical compositions corresponding to the representation construct in Armani. However, Wright gives a more elaborate semantic content to the mechanism: in Wright, the composition is expressed by defining a system in place of the computation of a component or the glue of a connector. *Bindings* are defined between the interfaces of the entity (component or connector) being decomposed and the interfaces of the decomposing system. The semantics of the representation mechanism is roughly that the computation process of the component being represented is the computation process of the representing system, but only the parts of the process defined by the bindings exposed to outside the representing system.

The decomposing mechanism is illustrated in Figure 5, with both a textual and the corresponding graphical presentation: the `server` component is represented by a system called `sub`.

Styles

In addition to the previously mentioned concepts, Wright includes the notion of *style*. Basically, a style describes a set of definitions shared by a set, or a family, of systems. Hence, the style construct in Wright corresponds the concept of style in Armani. But there are some differences in what can be included in styles between the two languages.

In Wright, a style can include connector and component type declarations. These declarations take the same form as type declarations in systems. Additionally, styles can include *interface types* that are process descriptions defined outside the context of a component or a connector. These can be used to define the processes in different Wright elements. In essence, when used in the place of the process in a port or role, interface types can be considered to be port and role types, as ports and roles define only a name and a process.

Further, component and connector type declarations in styles can be *parameterised* both with respect to the processes defined in the type and the number of interfaces. This mechanism pertains to leaving some aspects of a type declaration unspecified and type be filled in when instantiating the type. The C++ template system (Stroustrup 1997) is analogous to this parameterisation mechanism.

What extends the meaning of styles beyond defining useful design vocabulary is the possibility of defining *structural constraints* that pose limitations on the properties of a system following a specific style. Constraints are specified in first order predicate logic and they can refer all the relevant entities in Wright systems: components, connectors, ports, roles, attachments, computations, and glue. In addition to this type of structural constraints that concern the structure of systems, also *semantic constraints* concerning the behaviour of systems can be defined using CSP. These constraints limit the range of behaviour that can occur in the systems of the style.

3.2.3 Koala

Components and Interfaces

Koala (Ommering et al. 2000; Ommering 2000; Ommering 2001; Ommering 2002) is an ADL and a component model that has been developed at Philips Consumer Electronics to be used in developing embedded software for consumer electronic devices.

The main design elements of Koala are *components* that contain explicit connection points called *interfaces*. Each component and interface is of a single *type*. Interfaces are small sets of semantically related *functions*. Component in Koala is defined as an encapsulated piece of software with an explicit interface to its environment. Interfaces in Koala are similar to those in COM or Java. In COM (Microsoft Corporation 1995), interfaces are specifications of interaction points of components; they contain name and type information of functions and attributes, but no implementation of the functions. In Java (Arnold 2000), interfaces are pure types containing only function signatures: they do not contain any implementation or attributes.

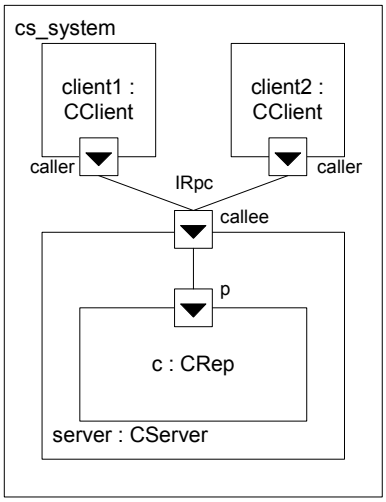
Components and interfaces are depicted in Figure 6, which defines the client-server system in Koala. Two instances, `client1` and `client2`, of the `CClient` component type, and one (`server`) of `CServer` type are defined. Further, the example includes component instance `c` of `CRep` type.

Compound components can be used to express compositional structure in Koala, i.e. a component can *contain* other components. An example of compound components can be seen in Figure 6 (a) and (b), the component `server` is defined to contain component `c`. The term *independent component* is used when talking about two or more components such that none of the components are contained within each other. In Figure 6, `client1`, `client2`, and `server` form a set of independent components.

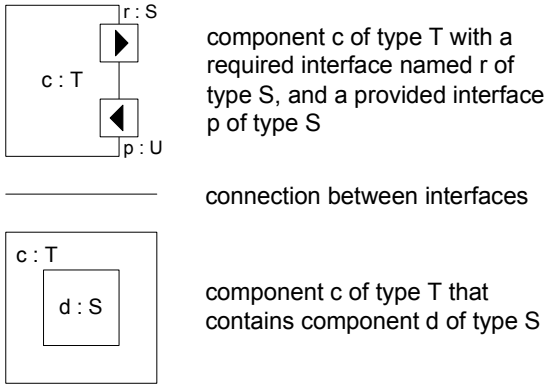
A distinction is made between *provided* and *required* interfaces. Loosely defined, a component having a provided interface means that the component offers a certain service for other components to use. Similarly, a required interface signals a certain service being required by the component from some other component in the form of a provided interface. Figure 6 (c) explains the notion used for provided and required interfaces, and for other Koala constructs used in Figure 6 as well.

```
component cs_system
{
  contains
    component CClient client1;
    component CClient client2;
    component CServer server;
  connects
    client1.caller = server.callee
    client2.caller = server.callee
}

component CClient
{
  requires IRpc caller;
}

component CRep
{
  requires  IRpc p;
}

interface IRpc
{
  void MakeCall(int n);
  ...
}

component CServer
{
  provides   IRpc callee;
  contains
    component  CRep c;
  connects
    caller = c.p;
}
```

(a)



(b)



(c)

Figure 6 The client-server system in Koala. (a) The system represented textually. (b) The system represented graphically. (c) Legend of the notation used.

Bindings

In Koala, topology is defined through *bindings* between interfaces. However, there are limitations on how interfaces can be bound to each other. These *binding rules* are best expressed in graphical terms. As depicted in the legend in Figure 6 (c), the interfaces of a component are drawn on the border of the component: a provided interface is drawn as a triangle the tip of which points outwards from the component; correspondingly, the tip of an provided interface points inwards. With this notation, the binding rules are that an interface must be bound by its tip to the base of exactly one interface, and by its base to any number of interfaces, including zero. The term *tip interface* is used to refer to the interface in the binding the tip of which is bound. The term *base interface* is defined correspondingly.

In addition, the binding rules concern the types of interfaces. Interfaces of arbitrary types may not be bound to each other; instead, the type of the tip interface in a binding must be a *supertype* of the type of the base interface. The supertype relation between interfaces is based on the content of interfaces, i.e., functions: given two interface types `A` and `B`, `A` is a supertype of `B` if and only if `B` contains all the functions of `A`. Then, `B` is correspondingly a subtype of `A`.

The above form of supertype relation becomes handy when considering how exactly interfaces are bound in Koala. Namely, a binding between two interfaces is implemented as bindings between the functions constituting the interfaces. That is, calls to a function in the base interface are textually replaced with calls to the function with the same name in the tip interface. Thereby, it is sufficient to require that the tip interface is a supertype of the base interface.

Figure 6 (b) contains examples of bindings: the required interfaces called `Caller` in both component `client1` and `client2` are bound by their tips to the base of the provided interface `callee` of component `server`. These bindings are examples of bindings between independent components. The binding between interfaces `callee` and `p`, on the other hand, is an example of a binding between interfaces in a contained and a containing component.

It should be noted that bindings between independent components are always between a required and a provided interface, whereas bindings between contained and containing components are always between a pair of required or provided interfaces. Although both forms of bindings are handled uniformly in Koala, bindings between independent components and contained and containing components could be classified similarly as attachments and bindings in Armani and Wright: bindings between independent components compare to attachments, and between a contained and containing component to binding in the Armani and Wright. Similarly, contained components in Koala compare to the representation mechanism in the other two ADLs.

In addition to simple bindings between interfaces, more complex bindings between interfaces can be defined. *Module* is the construct in Koala for implementing this. Basically, modules can define arbitrary glue code between interfaces. There are a number of important special types of patterns that can be achieved with modules.

First, instead of binding all the functions in the tip interface to functions of the same base interface, it is also possible to bind functions one by one to each other. This form of binding is termed *function binding*. Functions in the tip interface can be bound to functions in a single base interface, or in a number of base interfaces.

A second special case of the possibilities provided by a module is binding a tip interface with multiple base interfaces; this would not be possible without a module between the interfaces, because the binding rules prohibit a binding from a tip interface to multiple base interfaces. This kind of binding is illustrate in Figure 7 (a), where component `a` is bound simultaneously with components `b`, `c`, and `d`.

Still another important special case of bindings is *switch*. Switch is a construct that conceptually binds a tip interface into multiple base interfaces, and selects one of the interfaces to which the binding will be eventually made when the software is running. The selection between interfaces is done based on a parameter value that is obtained from a yet another interface. The use of switches is illustrated in Figure 7 (b). In the figure, interface `p` of component `a` is bound to switch `s`. Depending on a parameter `s` obtains from component `b`, the binding will be either to `c` or `d`.

Configurations

A *configuration* in Koala is defined as a component that is not contained in another component and that has no interfaces on its border. A configuration represents an independent piece of software, i.e., one that could be installed as the embedded software of a device, such as a television.

Diversity Mechanisms

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 diversity is manifested as variation of component *parameters*. Components can have a set of parameters. In Koala, there is typically a set of top-level parameters in a configuration that are independent of each other and often 'seen' by the end user. Rest of the parameters of a configuration depend on other parameters in a hierarchical manner: e.g., the values of a certain set of other parameters can be determined based on the values of the top-level parameters, and these lower level parameters can in turn determine the values of the still lower level parameters etc.

Diversity interfaces are interfaces specialised in providing components with the possibility of querying the parameter values they are interested in. However, there is no difference between a diversity interface and an ordinary provided interface beyond the fact that a diversity interface is specialised in providing parameters to other interfaces.

Structural diversity is implemented with switches, a special form of binding interfaces introduced above. Switches use parameters obtained from outside interfaces to produce diversity by determining which binding should be selected from a range of alternatives.

Additionally, components can declare both required and provided interfaces as *optional*. An optional interface is an interface that a component may, but is not guaranteed, to require or provide.

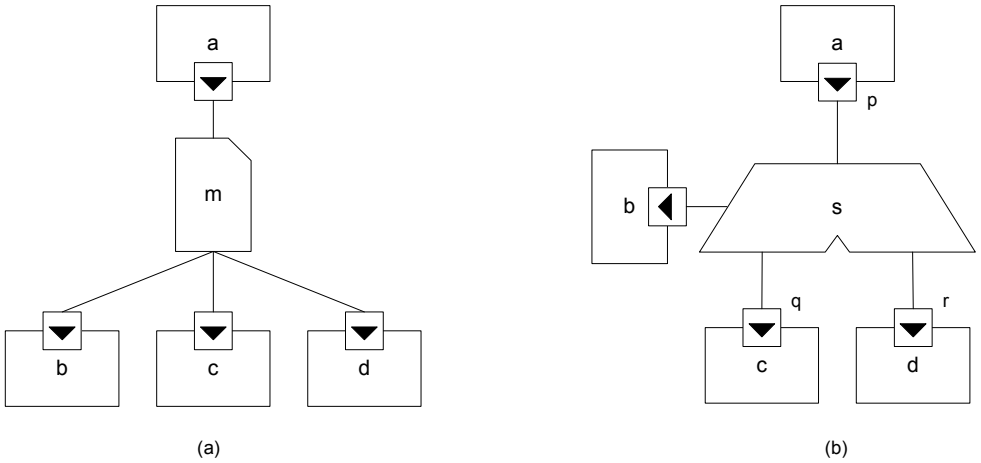


Figure 7 Modules in Koala. (a) A one-to-many module. (b) Switch.

4 Comparison of Concepts of the ADLs with the Configuration Ontology

This chapter uses a comparison framework to compare the concepts in the three ADLs analysed above with each other and with the corresponding concepts in the configuration ontology.

The comparison framework consists 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 and structure. The second part discusses the roles of types and instances in the languages, and the relations between types. The last part of the framework is the variation mechanisms provided by ADLs and the configuration ontology. In this thesis, any feature of a language or a conceptualisation that allows describing a number of different systems with a single representation is considered a variation mechanism; the way in which the systems differ from each other is not discriminated, i.e. the difference may arise from e.g. different structure or the behaviour of the systems.

4.1 Key Concepts and the Relations between Them

The results of the comparison are presented in Table 1. In the following, the results concerning each concept are elaborated.

Components and Connectors

Component is the central concept Armani, 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, or a configuration, in configuration modelling as well as in the architecture description languages. However, although similar, the semantics of component are not the same in all the disciplines: whereas Armani and Wright define component abstractly as a locus of computation, the definition of Koala is much more specific by specifying in more detail what components should be like. Further, the configuration ontology definition for component is the most general of the four, as any distinguishable whole is defined to be a component in the ontology.

Table 1 The main concepts of the ADLs and the configuration ontology

	Armani	Wright	Koala	Configuration ont.
Component	Primary computational units of a system	Localised, independent computation	An encapsulated piece of software with an explicit interface to its environment	Distinguishable whole in a product
Connector	Mediate interactions among components	An interaction among a collection of components	N/A	N/A
Configuration	System: collection of components, connectors, and a description of the topology of the components and connectors	System: a collection of component instances combined via connectors	A set of components connected together to form a product	A set of instances of the types occurring in the configuration model
Connection point	Port: a representation of the external interface of a component Role: a representation of the external interface of a connector	Port: an interaction in which the component can participate Role: a specification of the behaviour of a single participant in the interaction	Interface: a small set of semantically related functions	Port type: intensional definition of a connection interface Port individual: a place in a component individual where some other port individual can be connected to
Attribute	Property: annotation that store additional information about the language elements	N/A	An atomic property of a component instance	A characteristic of a type
Resource	N/A	N/A	N/A	An entity that is produced and used by component individuals, or flows from one component to another
Function	N/A	N/A	N/A	Function type: an abstract characterisation of the product that a customer or sales person would utilise to describe the it
Constraint	Invariant: design constraint that must hold in a design Heuristics: suggestion for creating effective design	Syntactic constraint: a structural property defined in a style that must be obeyed by any system in the style Semantic constraint: a behavioural property defined in a style that must be obeyed by any system in the style	N/A	A formal rule, logical or mathematical, which specifies a condition that must hold in a correct configuration
Topology	Attachments between ports and roles	Attachments between ports and roles	Bindings between interfaces or individuals functions in independent components	Connections between port individuals
Structure	Components and connectors can define representations consisting of systems and bindings between interfaces in the representation and the represented entity	The computation of a component or the glue of a connector can be replaced with a system; binding between the ports and roles of the system and of the component or connector	Components can contain other components; bindings between interfaces in compound components and the components contained in them	Part definitions in component types specify the roles in which other component individuals can be parts of the individuals of the component type

Connectors are entities present in Armani and Wright with the same semantics. However, there are no connectors in Koala or in the configuration ontology. Thus, there is a major difference in how the approaches handle architectural connection, or topology.

Armani differs from the other ADLs and from the configuration ontology in that component and connector instances and types are not separated from each other: i.e., a component or connector instance needs not to be of any explicit type, but may define its type by itself.

Configuration

The definitions for configuration are very similar to each other: in all the ADLs, configuration is seen as a collection of components connected together. However, in the configuration ontology a configuration is seen more broadly: a configuration includes not only component individuals, but also the individuals of other kind of types occurring in the configuration model besides component types. The other kind of types include port, resources, and function types. Their individuals are port individuals, resource production individuals, and functions. Resource production individuals specify how resources defined by resource types are produced and consumed by components; functions imply that a property of a product described by the function type is present in a configuration containing the function.

Connection Points

The notion of connection points is also common to all the studied modelling methods. In Armani 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 similar in all the approaches, except for Koala: in Koala, an interface is defined to be small set of semantically related functions, whereas the other approaches define ports and roles as external interfaces. Obviously, the term interface is overloaded with a double meaning: seemingly, the word is used in Koala to describe a set of points where interaction may occur, and a single, atomic point of interaction in the other approaches.

Although the definitions of an interface is similar in Armani, Wright, and the configuration ontology, there are some minor differences. It could be argued that the

Wright definition for ports and roles is more specific than the corresponding definitions in Armani and in the configuration ontology: Wright specifies ports as interactions wherein the component containing them can participate, and roles as specifications of single participants in interactions; there is no notion in Armani and the configuration ontology corresponding to that of interaction in Wright.

Attributes

All the disciplines, except for Wright, include a notion of attributes. Further, the semantics of attributes seems to be the same in wherever they are included: attributes store information that is relevant to an entity.

Resources

Resources are present in the configuration ontology but not in any of the ADLs. However, resources are similar to the notion of provided and required interfaces present in Koala in the sense that they are both anti-symmetric: just as the production of resources must match their consumption, there must be a provided interface for each required interface. 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. Further, the notion of provided and required interfaces and the associated binding rules in Koala are intuitively closer to connection constraints in the configuration ontology than resources.

Functions

Modelling functions is another feature of the configuration ontology that all three ADLs presented in this paper lack.

Constraints

All the disciplines under comparison, except for 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; here, Wright is the exception to the rule, as it includes semantic constraints that concern behavioural properties of systems, in addition to the form of constraints

shared with the other ADLs and the configuration ontology. Of the studied disciplines, Armani is the only to support heuristic constraints.

Topology

All the disciplines model topology in roughly the same manner, i.e., attachments or bindings between intensional connection points. However, there are some differences between the disciplines: in Armani and Wright, connectors are needed between components, whereas there is no similar construct in Koala and the configuration ontology. Further, the limitations on what kind of connections can exist are different: in Armani, the limitation is that attachments must be between a port and a role; in addition to this, Wright requires that each port and role is involved only in a single attachment, and that the CSP process in the port and the role are compatible; Koala has its own binding rules for binding interfaces; in the configuration ontology, compatibility definitions between types are needed in order for any connections to exists, and the connections are additionally subject to connection constraints in the port types and definitions.

Structure

All the studied disciplines model structure as well. Unfortunately, all the ADLs use different names for same entities. Therefore, a unified terminology is introduced:

- The term *component* is used to refer to components and connectors in Armani, Wright, and components of Koala.
- The term *interface* is used to refer to ports and roles in Armani and Wright, and interfaces in Koala.
- The term *compound component* is used to refer to any component or connector that contains a representation in Armani, and to any component or connector in Wright, whose computation or glue, respectively, is specified by a system declaration, and any component containing components in Koala.
- The term *contained component* is used to refer to any component or connector that is defined in the context of a compound component.

- The term *binding* is used to refer to bindings in Armani and Wright, and to those connections between interfaces in Koala that are between interfaces in a compound and contained component.
- The term *attachment* is used to refer to attachments in Armani and Wright, and to those connections between interfaces in Koala that are between interfaces in two independent components.

Of the ADLs, Armani and Wright enable defining systems within component and connectors. Armani allows multiple representations, whereas Wright allows only one. In Koala and the ontology, on the other hand, structure of components is specified in terms of components: Koala used the notion of contained components and the ontology part definitions to express this. In the configuration ontology, part definitions in component types are used for specifying structure.

It is possible to specify bindings between the interfaces in the contained components and the compound component: in Armani and Wright bindings and attachments are strictly different concepts, but in Koala a single concept is used for both. No mechanism for specifying bindings is included in the configuration ontology. However, there can be connections between ports in compound and contained components.

4.2 Distinction between Types and Instances

All the three ADLs studied have some distinction between types and instances. However, the handling of types is not alike in any two studied languages. This section compares the handling of types and instances in each ADL and the in the configuration ontology. In the analysis, special attention is paid to the following issues:

- How can an instance be of a type?
- What is the number of types an instance can be of?
- The relations between types (taxonomy and other)

The data for each language concerning each of the above-mentioned issues is presented in Table 2.

Table 2 Summary of the handling of types and instances

	Armani	Wright	Koala	Configuration ontology
Being of a type	Design elements: By declaration or by implicitly satisfying all the predicates defined by the type Styles: Declaring a system in a style and satisfying the constraints defined by the style	Components and connectors: Instantiating an entity with the type System: Declaring the system be of a style and satisfying the constraints defined by the style	Interfaces and components: By declaration	An individual is directly of a type when it is instantiated with that type; the individual is valid with respect to the type if it has all the properties specified by the type
Number of types per instance	Design elements: Any number of both declared and satisfied types. Systems: any number of styles	Components and connectors: exactly one type Systems: zero or one styles	Interfaces and components: Exactly one type	Directly of exactly one type and indirectly of any number of supertypes
Relations between types	Design element types: the set of supertypes is the set of element types declared as the supertypes of a type	Styles: a style is a sub-style of another style if it has all the constraints of the other style	Interface types are defined to be sub- and supertypes of each other based on the functions they contain.	Types organised in classification taxonomies by the isa-relation. Additional relations between types, e.g. the compatibility relation

Being of a Type

First, all the disciplines except Armani have roughly the same mechanism for instance satisfying a type: An instance can only be of a given type if the instance is created to be of the type. Armani's notion on an instance being of a type is dual. As discussed in the section about Armani (Section 3.2.1), an instance can both declare and satisfy a type, and declaring a type implies satisfying the type, otherwise the instance is not valid.

The configuration ontology specifies additionally that an instance of a type is also indirectly an instance of all the supertypes of that type. Hence, the ontology specifies a dual notion on an instance being of a type. In Armani, an instance that satisfies a type satisfies all the supertypes of the satisfied types: this is due to the fact that types are defined as sets of predicates, and the set of predicates of a supertype is a subset of those of the subtype.

Number of Types per Instance

The dual notion on defining an instance to be of a type in the ontology and in Armani is also reflected in the number of types an instance can be of. In Armani, an instance can have an arbitrary number of both declared and satisfied types. In the configuration

ontology, each instance is directly of one type and indirectly of any number of types. Wright and Koala rely on single typing: each component and connector in Wright and interface in Koala is exactly of one type; Wright systems are optionally of a style.

Relations between Types

In Armani, the relations between design element types and styles are based on declarations: a type can be declared to be a subtype of a set of types. In Wright, there are no relations between component and connector types. On the other hand, there is a notion of sub-style in Wright: subtyping is based on the content of styles, as specified in Table 2. However, styles can be declared using existing styles as the basis. In Koala, interface types are related to each other as subtypes and supertypes. The relations are based on the content of the interface types: a subtype contains all the functions of its supertype. Additionally, invariants in Armani and constraints in Wright can be used to specify arbitrary relations between types.

By far the most sophisticated relations between styles can be found in the configuration ontology: Types are organised in classification taxonomies based on the ISA-relation. In addition, types can be in a number of relations with each other: these relations include the part definitions concerning component types, port definitions concerning component and port types, production and use definitions concerning resource types and component types, compatibility definitions between port types, and implementation constraints between function types and component, port, and resource types. Further, similarly as invariants in Armani and constraints in Wright, general constraints can be used to specify other, arbitrary relations between types.

4.3 Variation Mechanisms

This section compares the variation mechanisms of the ADLs and of the configuration ontology. As can be recalled from the introduction, any feature of a language or a conceptualisation that allows describing a number of different systems with a single representation is considered a variation mechanism.

An apparent variation mechanism in Armani and Wright are styles. As explained in the respective sections on these ADLs, styles are combinations of design vocabulary, i.e. predominantly type definitions, available for the systems and constraints about the structure, and behaviour in Wright, of systems in that style. It can be argued that

constraints are needed to make styles a variation mechanism: a style with only design vocabulary does not describe any set of systems, as using the parts of design vocabulary is perfectly optional to the systems in that style. Constraints, on the other hand, can be used to enforce some properties of the systems using in the style and can be consequently said to describe the set of systems in the sense specified in the definition of a variation mechanism. Thereby, styles with constraints are a variation mechanism.

The above-described variation mechanism of Armani in Wright is likewise embedded in the configuration ontology: the constraint mechanism of the ontology can be used to express the same constraints as in the ADLs, and types can naturally be presented in the ontology as well. Of course, it must be assumed that the constraint language used in conjunction with the ontology is capable of expressing the same constraints as the constraint languages of Armani and Wright.

A major difference is that a style definition is essentially an *open* specification in the sense that the systems using the style can and are expected to supplement it with additional declarations. A configuration model, on the other hand, is *closed* in the sense that during the configuration process, the model is not subject to change, but the configuration is created based on the model. In other words, a style must be supplemented with additional declarations in order to create a system instance, whereas a configuration is created by reducing the set of configurations described by the configuration model to a single configuration through a series of configuration decisions.

Koala, on the other hand, has no construct corresponding to styles in Armani and Wright. However, Koala includes mechanisms for both internal and structural diversity. As was explained when analysing Koala, internal diversity pertains to components having parameter values that affect their functionality. Structural diversity, in turn, is implemented with modules and reflected in variation in the bindings between components. Switch is a special case of the diversity that can be created with modules: when there are multiple provided interfaces to serve a required interface, a switch is used to decide the provided interface to be actually used. In a sense, switches are a variation mechanism corresponding to alternative connections.

Another form of variability in Koala is provided by optional interfaces. Basically, optional interfaces create variability to the topology of a compound component, or a configuration. On a higher level of abstraction, optional interfaces make the range of

services required and provided by components variable, and thereby essentially affect the functionality of components and configurations. Intuitively, optional interfaces and optional parts correspond to each other.

The configuration ontology is aimed at modelling configurable products. To support this aim, variation mechanisms are embedded in all the modelling facilities of the ontology. Therefore, a full account of the variation mechanisms of the ontology is not presented here; the interested reader should refer to the section were the ontology was introduced (Section 2.2.1), or to the original papers describing the ontology and its use (Soininen et al. 1998; Tiihonen et al. 1998). However, as has been discussed above, there is a corresponding mechanism for most of the variation mechanisms identified in the ADLs: styles and constraints correspond roughly to configuration models; parameters in Koala components are paralleled by attributes in the ontology; optional interfaces correspond to optional ports that can be expressed with the cardinality in port definitions. However, it is not obvious how the variability provided by switches could be encoded in the configuration ontology, let alone the variability provided by modules in their general form. One way of modelling modules will be given in Chapter '5.5.1 Mapping Koala to the Ontology' below.

5 Synthesis

This chapter presents a construction, in which for each ADL the following tasks are completed. First, those concepts in the ADL that can be mapped into the configuration ontology are given a mapping. Second, ontological constraints are given. The constraints serve three purposes. First and foremost, they form an incremental extension to the ontology required for representing architectures with a semantics equivalent with the ADL; second, the constraints characterise the configuration models that correspond to the ADL; third, the ontological constraints allow reasoning about the relative expressive power of each ADL and the configuration ontology: the modelling facilities the use of which is prohibited by the ontological constraints represent expressive power of the ontology not matched by the ADL. As the last part of each section, a mapping from configuration models to the ADL is given.

Before the actual construction, a model of the entities in each ADL and of the configuration ontology is presented.

5.1 A Model of Entities in the ADLs and in the Ontology

This section presents a model of the entities with their properties in each ADL and in the configuration ontology. Only entities and properties relevant for the construction following later in this section are presented. This concerns especially the configuration ontology. Again, the interested reader is asked to refer to the original papers presenting the ontology (Soininen et al. 1998; Tiihonen et al. 1998).

The entities are presented in tables from Table 3 through Table 8. The short name of the property is the identifier that is used to refer to the property in the construction.

5.2 On the Style of Presenting the Mapping

Mappings between disciplines will be presented in the form of definitions. Each definition will specify the steps needed to map a concept into the other domain. The definitions use a mixture of styles to specify the mapping. First, definitions specify *functions* that take entities in a domain as their input and return corresponding entities in other domains. However, all entities cannot be fully mapped into the ontology simply by

recursively applying functions to entities. This is due to the fact that in addition to functions, definitions contain *procedural actions* needed to map an entity.

Thereby, what should be done to map a concept into another domain is to first apply the definition specifying a mapping for the concept. Then, whenever the definition refers to a function not specified in the definition, definition containing the referred function should be applied to the entities that are the arguments of the function. That is, not only should the return value of the function be assigned to some property of an entity, but also the procedural actions specified in the definition containing the function should by taken.

The mappings are presented using top-down approach: in common terms, the concepts aggregating other concepts are presented before the aggregated concepts. This approach has the advantage that whenever a mapping is given for an entity, the reader will already know how the concept aggregating the concept being mapped has been mapped. The significance of this advantage is emphasised by the fact that most whole-part relationships between entities are mapped into some kind of whole-part relationships in the target domain. However, the top-down approach has the drawback that the mappings for entities typically refer to the mappings of such entities that have not yet been given a mapping.

The following typographical conventions are followed: Arial is used to typeset the names of entity types (e.g., Component type); *Times New Roman Italic* is used for names of variables and functions (*c*, *c.name, system*$_{AO}$); SMALL CAPITALS are used for entities that are in some sense constant, e.g. enumerated values, names of general types, and names of relations (CONCRETE, ADLSYSTEM, ISA). Finally, `Courier New` is still used for exemplars (`cs_system`).

Further, the functions used in the mapping are named using the following conventions. In the names of functions, each ADL and the configuration ontology is referred to with a letter: 'A' stands for Armani, 'W' for Wright, 'K' for Koala, and 'O' for the configuration ontology. Further, each function that maps an entity *e* in discipline *x* to discipline *y* is named e_{xy}. For example, a function that maps a component in Armani to the configuration ontology is name $component_{AO}$.

Table 3 Entities of the configuration ontology

Entity	Properties	Short name	Description
Configuration model	Component types	components	The set of Component types of the configuration model
	Port types	ports	The set of Port types of the configuration model
	Attribute value types	attributes	The set of Attribute value types of the configuration model
Component type	Name	name	The name of the type
	Abstraction definition	abstraction	Possible values are: CONCRETE and ABSTRACT. Only individuals of CONCRETE types can occur in complete configurations.
	Dependency definition	dependency	Possible values: INDEPENDENT and DEPENDENT. Only individuals of INDEPENDENT component types may occur in a valid configuration without being a part of some other component individual.
	Part definitions	parts	The set of Part definitions of the type
	Port definitions	ports	The set of Port definitions of the type
	Supertypes	types	The set of Component types that are supertypes of the type
	Attributes	attributes	The set of Attribute definitions of the type
Port type	Name	name	The name of the type
	Abstraction	abstraction	See the same property in Component type.
	Supertypes	types	The set of Port types that are supertypes of the type
	Attributes	attributes	The set of Attribute definitions of the port type
Part definition	Name	name	The role in which the component individuals are parts of the individuals of the component type
	Set of possible part types	types	The set of Component types the individuals of which are allowed to occur as parts with the associated part name
	Cardinality	cardinality	The number of component individuals that must occur as parts with the part name
Port definition	Name	name	The role in which the port individuals are ports of the individuals of the component type
	Set of possible part types	types	The set of Port types the individuals of which are allowed to occur as ports with the associated part name
	Cardinality	cardinality	The number of port individuals that must occur as ports with the port name
	Connection constraints	constraints	The set of Constraints that must be satisfied by the two ports and the components the port individuals are ports of in order for them to be allowed to be connected to each other
Attribute value type	Name	name	The name of the type
Attribute definition	Name	name	The name of the attribute
	Value type	type	The attribute value type of the attribute
	Necessity	necessity	The possible values are NECESSARY and OPTIONAL. Attributes that are defined NECESSARY must have exactly one value, whereas OPTIONAL attributes have at most one value.
Constraint	Name	name	The name of the constraint
	Expression	expression	The constraint expression of the constraint

The properties of the entities represented in table from Table 3 through Table 8 referred to using the dot notation. E.g. the set of component types defined in Configuration model *M* are referred to as *m.components* (see Table 3).

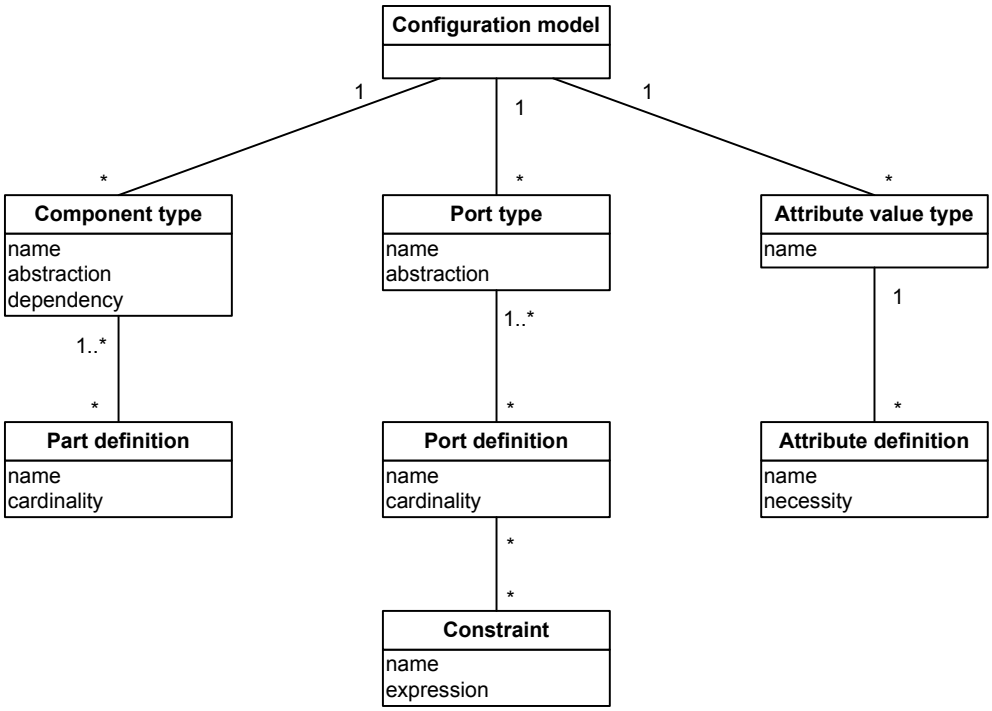


Figure 8 Entities of the configuration ontology

Table 4 Basic entities of Armani

Entity	Properties	Short name	Description
Common properties	Name	Name	The name of the entity
	Properties	properties	The **Property**s of the entity
System	Components	components	The **Component instance**s contained in the system
	Connectors	connectors	The **Connector instance**s contained in the system
	Attachments	attachments	The **Attachment**s of the system
	Styles	styles	The **Style**s declared by the system
Component	Ports	ports	The **Port**s declared in the component
	Representations	reps	The **Representation**s of the component
	Types	types	The set of **Component type**s declared by the component
Connector	Roles	roles	The **Role**s declared in the connector
	Representations	reps	The **Representation**s of the connector
	Types	types	The set of **Connector type**s declared by the connector
Port	Types	types	The set of **Port type**s declared by the port
Role	Types	types	The set of **Port type**s declared by the role
Representation	System	sys	The **System** defined in the representation
	Bindings	bindings	The set of **Binding**s between ports and roles in the representation, and those in the **Component** or **Connector** containing the representation
Attachment	Port	port	The **Port** that the attachment concerns
	Role	role	The **Role** that the attachment concerns
Binding	First interface	first	Each binding concerns two interfaces, i.e., **Port** or **Role**, and the order of the interfaces is semantically irrelevant.
	Second interface	second	
Invariant	Expression	expression	The invariant expression
Heuristic	Expression	expression	The heuristic expression

Note: Attachments and binding do not have the common properties "name" and "properties".

Table 5 Additional entities of Armani

Entity	Properties	Short name	Description
Common properties	Name	Name	The name of the entity
	Properties	properties	The **Property**s of the entity
Component type	Ports	Ports	The **Port**s declared in the component type
	Representations	Reps	The **Representation**s of the connector
	Invariants	invariants	The **Invariant**s of the type
	Heuristics	heuristics	The **Heuristic**s of the type
	Types	Types	The set of **Component type**s the type extends
Connector type	Roles	Roles	The **Role**s declared in the component
	Representations	Reps	The **Representation**s of the connector
	Invariants	invariants	The **Invariant**s of the type
	Heuristics	heuristics	The **Heuristic**s of the type
	Types	Types	The set of **Connector type**s the type extends
Port type	Invariants	invariants	The **Invariant**s of the type
	Heuristics	heuristics	The **Heuristic**s of the type
	Types	Types	The set of **Port type**s the type extends
Role type	Invariants	invariants	The **Invariant**s of the type
	Heuristics	heuristics	The **Heuristic**s of the type
	Types	Types	The set of **Role type**s the type extends
Property type	Type declaration	declaration	The property type declared
Style	Component types	components	The **Component type**s declared in the style
	Connector types	connectors	The **Connector type**s declared in the style
	Port types	Ports	The **Port type**s declared in the style
	Role types	Roles	The **Role type**s declared in the style
	Property types	properties	The **Property type**s declared in the style
	Invariants	invariants	The **Invariant**s declared in the style. The semantics are that the systems of the style must satisfy the invariants to be true
	Heuristics	heuristics	The **Heuristic**s declared in the style. Heuristics are predicates that the systems of the style are suggested to satisfy.
	Design analyses	Analyses	Functions in the Armani Predicate Language that can be used in forming invariants and heuristics in the style and in systems of the style.
	Styles	Styles	The set of **Style**s the style extends

In addition to tables, the entities are presented as UML class diagrams in figures from Figure 8 though Figure 11. The structure of systems and representations illustrated in Figure 9 (b) applies to Wright as well. However, there are minor differences in the cardinalities; these differences have been explicated in the figure using non-standard notation.

The figures do not attempt to capture all the aspects of the disciplines, but rather to give an overview of the relations between different entities in a graphical form. Issues that would have resulted in complex figures have been omitted: Koala binding rules are an example of this kind of issue.

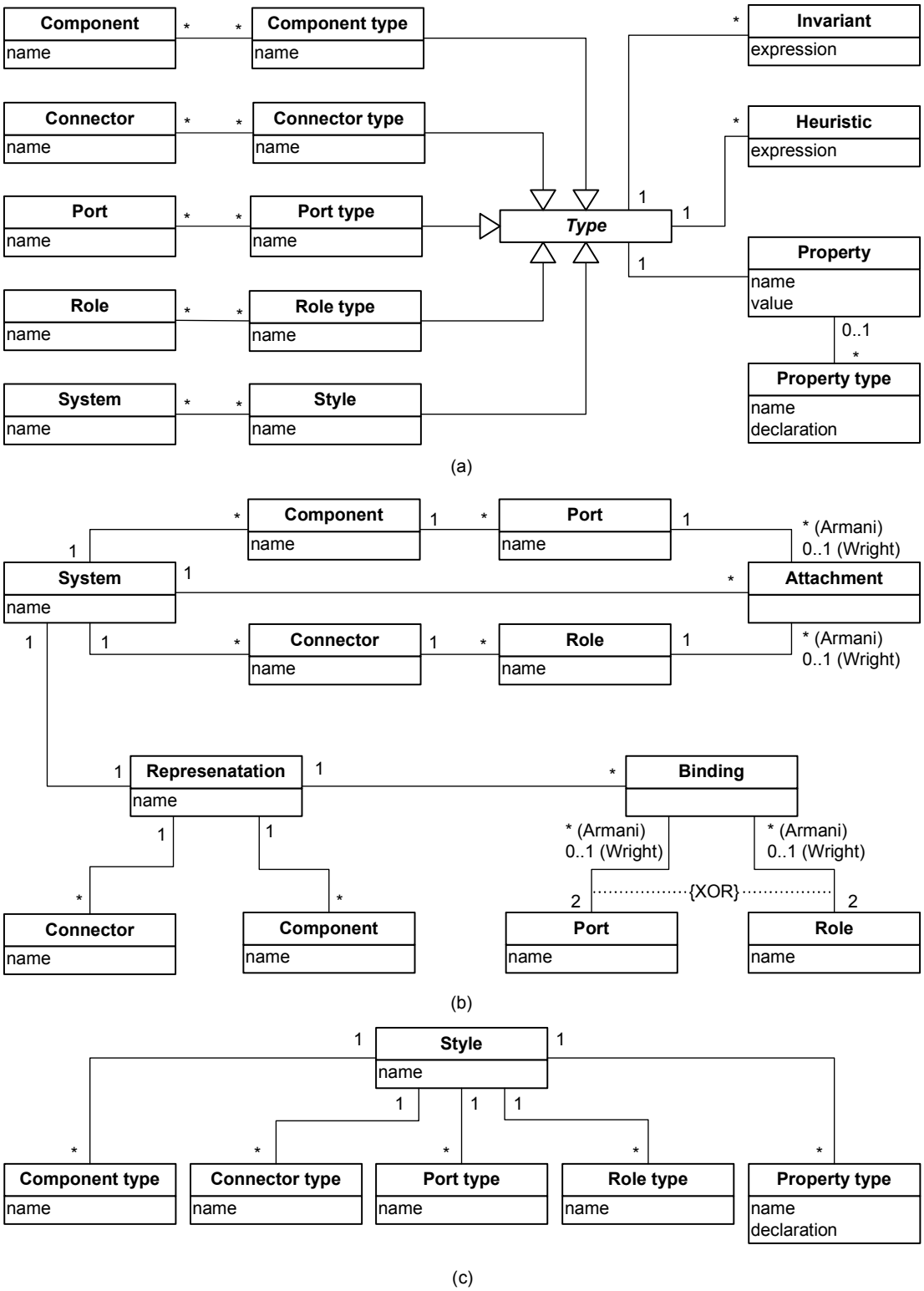


Figure 9 The entities of Armani. (a) The associations between design elements and design element types, systems and styles, and the constructs associated to both design element types and styles. (b) The structure of systems and representations. (c) The structure of styles.

Table 6 Basic entities of Wright

Entity	Properties	Short name	Description
System	Name	name	Name of the system
	Component types	comp_types	The set of Component types declared by the system
	Connector types	conn_types	The set of Connector types declared by the system
	Component instances	components	The set of Component instances of the system
	Connector instances	connectors	The set of Connector instances of the system
	Attachments	attachments	The Attachments of the system
	Style	style	The Style declared by the system
Component type	Name	name	Name of the type
	Ports	ports	The Ports declared by the type
	Computation	computation	The computation process of the type
	Representation	rep	The System acting as the Computation
Connector type	Name	name	Name of the type
	Roles	roles	The Roles declared by the type
	Glue	glue	The glue process of the type
	Representation	rep	The System acting as the Glue
Component instance	Name	name	Name of the instance
	Type	type	The Component type of the instance
	Cardinality	cardinality	The number of instantiated components
Connector instance	Name	name	Name of the instance
	Type	type	The Connector type of the instance
	Cardinality	cardinality	The number of instantiated connectors
Port	Name	name	Name of the port
	Process	process	The process of the port
	Cardinality	cardinality	The number of ports instantiated
Role	Name	name	Name of the role
	Process	process	The process of the role
	Cardinality	cardinality	The number of ports instantiated
Representation	System	sys	The System in the representation
	Bindings	bindings	The set of Bindings between ports and roles in the representation, and those in the Component or Connector containing the representation
Attachment	Port	port	The Port that the attachment concerns
	Role	role	The Role that the attachment concerns
Binding	First interface	first	Each binding concerns two interfaces, i.e., Ports or Roles, and the order of the interfaces is semantically irrelevant.
	Second interface	second	

Table 7 Additional entities of Wright

Entity	Properties	Short name	Description
Style	Component types	components	The set of Component types declared by the style
	Connector types	connectors	The set of Connector types declared by the style
	Interface types	interfaces	The set of Interface types declared by the style.
	Syntantic constraints	constraints	The set of syntactic constraints declared by the style
	Semantic constraints	semantic	The set of semantic constraints declared by the style
Interface type	Name	name	Name of the type
	Process	process	The process specified by the type

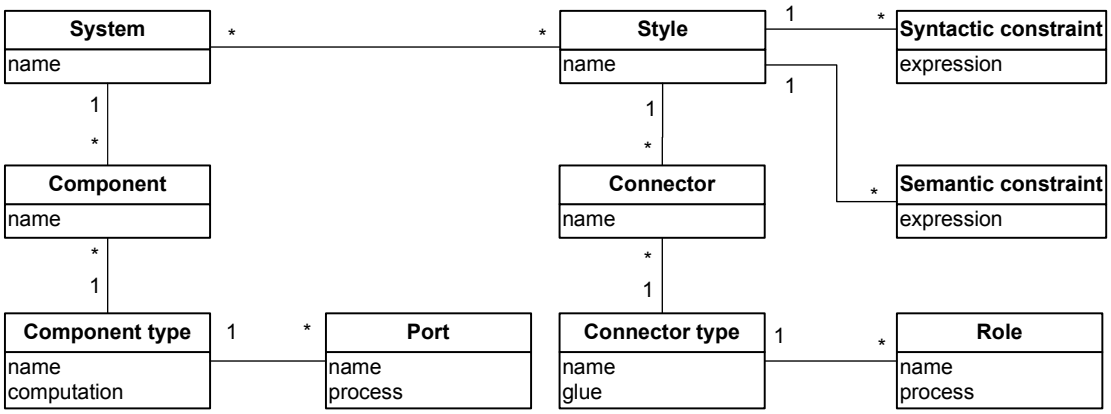


Figure 10 Entities of Wright

Table 8 Entities of Koala

Entity	Properties	Short name	Description
Configuration	Name	name	The name of the configuration
	Components	components	The Component definitions contained in the configuration
	Bindings	bindings	The set of Bindings between the component definitions
Component type	Name	name	Name of the type
	Interfaces definitions	interfaces	The Interface definitions of the component
	Contained components	components	The Component definitions contained in the configuration
	Bindings	bindings	The set of Bindings defined in the type
Interface type	Name	name	Name of the interface
	Functions	functions	The Functions constituting the interface
Function	Name	name	The name of the function
Interface definition	Name	name	The name by which the interface is provided or required
	Type	type	The Interface type of the definition
	Direction	direction	The direction of the definition. Possible values are: PROVIDED and REQUIRED
	Necessity	necessity	The necessity of the interface. Possible values are: OPTIONAL and MANDATORY. An interface with the value OPTIONAL is an optional interface, whereas the value MANDATORY indicates a non-optional (regular) interface.
Component definition	Name	name	The name by which the component is contained
	Type	type	The Component type of the contained component
Binding	Required interface	required	The required Interface definition involved in the binding
	Provided interface	provided	The provided Interface definition involved in the binding
Function binding	Required function	required	The required Function involved in the binding
	Provided function	provided	The provided Function involved in the binding
Module	Interfaces	interfaces	The set of Interface definitions connected to the module

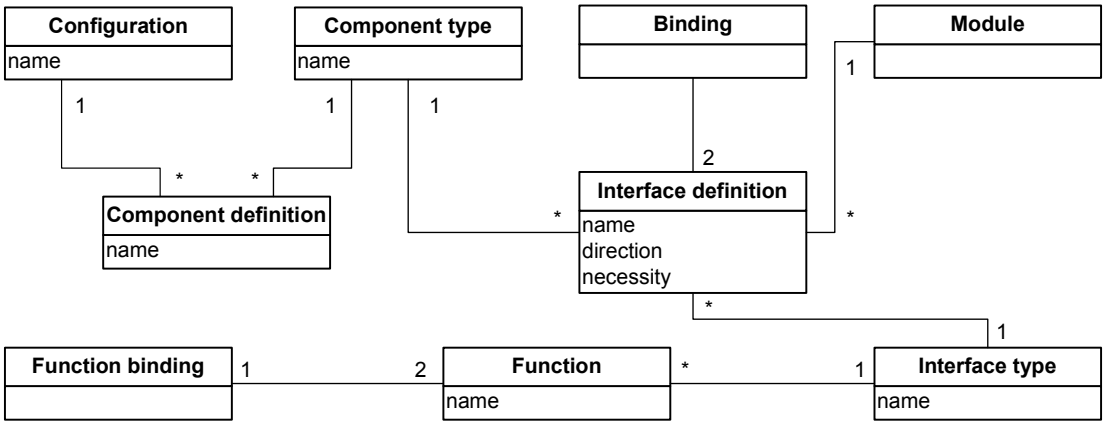


Figure 11 Entities of Koala

5.3 Mapping between Armani and the Ontology

This section consists of three subsections. First, a mapping from Armani to the ontology will be shown. Second, ontological constraints needed to make the resulting configuration models sound will be explicated. Third, a mapping from configuration models to Armani will be given.

5.3.1 Mapping Basic Concepts of Armani to the Ontology

This section presents a mapping from Armani to the configuration ontology.

Overview of the Mapping

Before going into details of the mapping, an overview is given in order for the reader to get the big picture. First, the fact that there are components in Armani and in the ontology suggests that in Armani components could be mapped into components in the configuration ontology. The same argument applies to ports.

Further, as the structure of connectors is similar to that of components, and the relation between connectors and systems is the same as that between components and systems, connectors are mapped into components in the configuration ontology as well. Given that connectors in Armani are mapped into components in the configuration ontology, ports in the ontology are the natural counterpart of Armani roles.

The fact that systems in Armani contain components and connectors and that components and connectors of Armani are represented as components in the ontology suggests that systems should be mapped into components in the ontology as well: after all, component type is the only concept in the ontology that can be specified a structure in terms of other component types. Moreover, the representing systems are mapped into components as well. This is motivated by two reasons. First independent systems are mapped into systems. Hence, the mapping will create a symmetry between the mappings for the two classes of systems. Second, representing systems have names and can define properties; component is a concept that can accommodate these features of representing systems.

Above, it was stated that concepts in Armani are mapped into components and systems. To be more exact, all the entities discussed above are mapped into component and port types. This is in a sense counter to intuition: instances are mapped into types, although

individuals would also be available in the configuration ontology. However, unlike instances of Armani, all the individuals in the configuration ontology are directly of a type. Consequently, mapping instances in Armani to individuals of the ontology would require that corresponding types to be defined. Hence, the approach of mapping instances into types is adopted in order to avoid each entity in Armani being mapped into at least two entities in the ontology.

In summary, components, connectors, systems, and systems in representations are mapped into component types in the configuration ontology. Respectively, both ports and roles in Armani were mapped into port types.

General Definitions

In order to distinguish components and ports in the configuration ontology corresponding to different entities in Armani, a taxonomy of component and port types is introduced and presented in Figure 12.

All the types in Figure 12 with their names beginning with 'ADL' are abstract: i.e., their individuals cannot occur in complete configurations. In addition, all component types except ADLSYSTEM are dependent: i.e., the individuals of these types cannot occur in valid configurations without being a part of some other component individual.

Further, compatibility definitions are needed to allow connections between ports and roles, and to disallow connections between two ports or roles. In detail, the following compatibility definitions are added to the ADLPORT and ADLROLE types.

Definition 5-1 ADLPORT is compatible with ADLROLE, and ADLROLE is compatible with ADLPORT.

Systems

As stated above, systems are mapped into components. This task is carried out by the function $system_{AO}$ defined below.

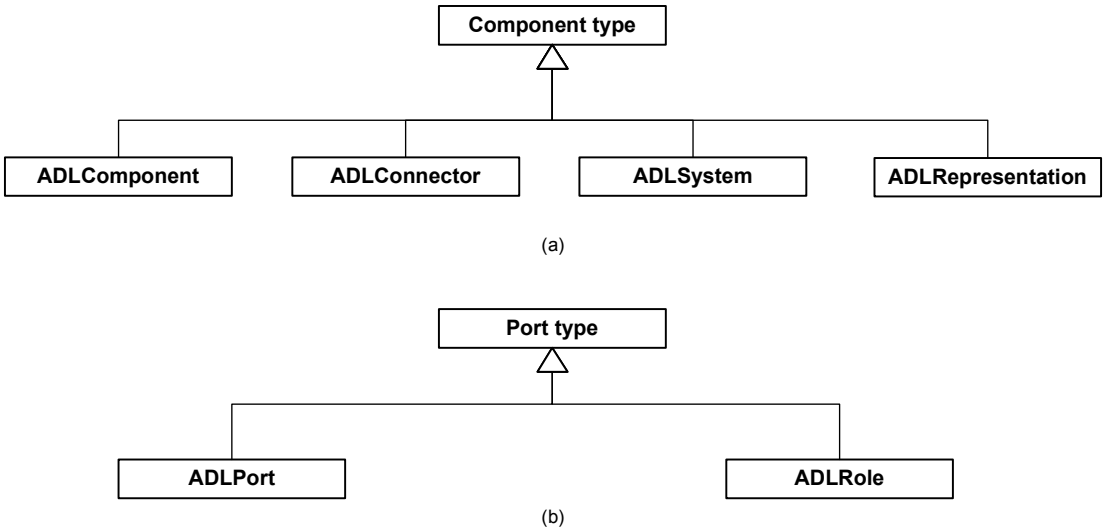


Figure 12 Component and port types used in mapping Armani. (a) The taxonomy of component types. (b) The taxonomy of port types.

Definition 5-2 System *s* is mapped into Configuration model *M* and Component type $system_{AO}(s)$. Initially, *M* contains the component and port types of Figure 12 and the compatibility definitions of Definition 5-1. The function $system_{AO}$ is defined as follows:

$system_{AO}(s)$:= Component type *c*, where

c.name := *s.name*

c.abstraction := CONCRETE

c.dependency := INDEPENDENT

c.parts := $component_{AO,\ def}(s.components) \cup connector_{AO,\ def}(s.connectors)$

c.types := { ADLSYSTEM }

c.attributes := $property_{AO}(s.properties)$

Further, for each Attachment *a* in *s.attachments*, and each Role *r* such that *r* = *a.role*, $role_{AO}(r)$ is modified as follows:

constraints := $role_{AO}(r).constraints \cup \{ attachment_{AO}(a) \}$.

M is modified as follows:

components := $M.components \cup system_{AO}(s)$. ■

As the definition shows, an Armani system is mapped into a concrete and independent subtype of ADLSYSTEM. The components and connectors of *s* are mapped to parts of *c* by the functions $component_{AO,\ def}$ and $connector_{AO,\ def}$; these functions are specified below when defining the mapping for Armani components and connectors. Similarly, properties of *s* are mapped into attributes of *c* using the $property_{AO}$ function, also

specified below. Further, the function $attachment_{AO}$ returns a constraint based on the attachment it receives as input.

Example. The example involving a client-server system that was used when analysing ADLs will be used to demonstrate the construction as well. Figure 13 depicts the results from mapping the client-server system in Armani (Figure 3) to the configuration ontology.

The notation used for depicting the configuration models is UML. There are two reasons for using UML. First, there is no standard graphical notation for depicting configuration models or configurations. Second, the constructs of UML can be used to capture the relevant aspects of the configuration models represented with it.

In Figure 13 (a), it can be seen that `cs_system` is defined as a subtype of ADLSYSTEM. Both ADLSYSTEM and `cs_system` are types that are represented by classes in the figures. Further, the fact that `cs_system` is a subtype of ADLSYSTEM is illustrated with a generalisation arrow between the classes. ■

In the following definitions in this section, *M* refers to a specific configuration model: in common terms, *M* is the configuration model that results from mapping a system using the above definition. That *M* is indeed a specific configuration model in the definitions below can be understood by observing that all entities in Armani are defined in the context of an independent system, except for independent systems themselves. Thereby, each entity is associated with a specific system. And as mapping an independent system results in a configuration model, each entity is associated with a single configuration model as well.

Components and Connectors

Components in Armani systems, both independent and representing, are mapped into component types using the function $component_{AO,\ type}$ and into part definitions using the function $component_{AO,\ def}$. Both these functionsa are defined in the definition below.

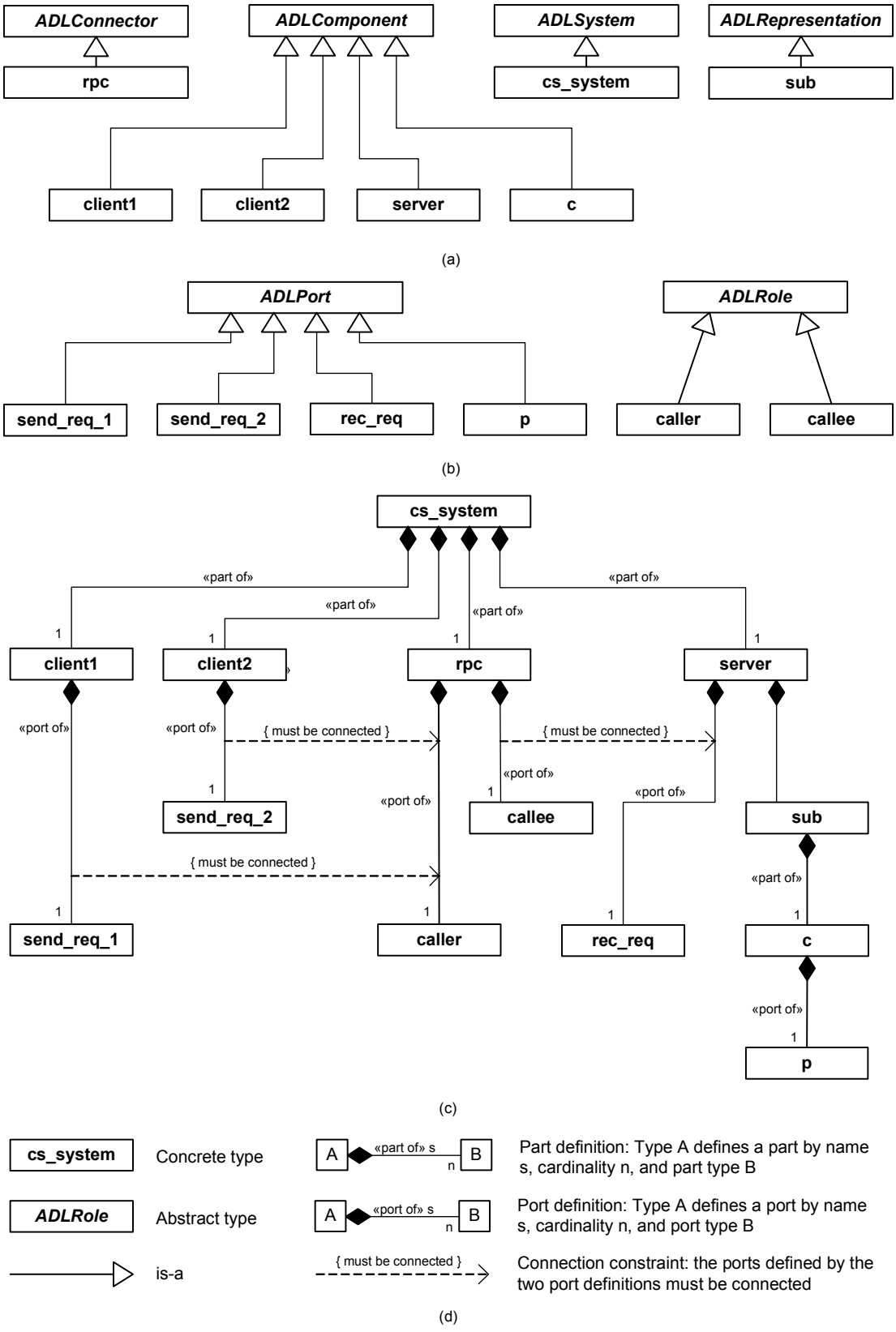


Figure 13 Mapping the client-server system from Armani to the configuration ontology. (a) The taxonomy of component types. (b) The taxonomy of port types. (c) The part and port structure in the configuration model. (d) Legend of the notation used.

Definition 5-3 Component *c* is mapped into Component type $component_{AO,\ type}(c)$, and Part definition $component_{AO,\ def}(c)$. The function $component_{AO,\ type}$ is defined as follows:

$component_{AO,\ type}(c)$:= Component type *t*, where

t.name := *c.name*

t.abstraction := CONCRETE

t.dependency := DEPENDENT

t.parts := $representation_{AO}(c.representations)$

t.ports := $port_{AO}(c.ports)$

t.types := { ADLCOMPONENT }

t.attributes := $property_{AO}(c.properties)$

The function $component_{AO,\ def}$ is defined as follows:

$component_{AO,\ def}(c)$:= Part definition *d*, where

d.name := *c.name*

d.types := $component_{AO,\ type}(c)$

d.cardinality := 1

M is modified as follows:

M.components := $M.components \cup component_{AO,\ type}(c)$. ■

The above definition uses the function $representation_{AO}$ for mapping the representations of c into the configuration ontology. The function is specified below in Definition 5-9.

It should be observed that a single component in Armani results in two entities in the configuration model: a component type and a part definition. The component type is created to exist independently of other entities in the configuration model. However, in contrast, part definitions cannot exist independently of other entities; as was already seen in Definition 5-2, the $component_{AO,\ def}$ function was used to produce the part definitions of the component type corresponding to an Armani system. In a sense, the above definition cannot be applied independently of other definitions, as the resulting part definition cannot exist independently of other entities. This is, however, in accordance with the semantics of components in Armani: they cannot exist independently of systems.

It is intuitively clear that a component in Armani is mapped into component type in the ontology: an Armani component contains information that corresponds to the properties component types can define in the configuration ontology.

However, that a component is mapped into a part definition as well is not as intuitive. The motivation for this kind of mapping can be seen by observing that in Armani, there is a relationship between a system and a component contained in it; the relationship is not a property of either the system or the component. Part definition is the concept in the configuration ontology corresponding to the relationship. The alternatives for mapping the relationship are: first, promoting the relationship between the system and the component to an entity; second, map the relationship when mapping the system; and third, mapping the relationship when mapping the component.

Above, the third approach was adopted, albeit the other two approaches would have been viable as well. The first approach would have had the drawback that the entity model would have introduced an entity in the entity model without a counterpart in the language. Additionally, the entity would have had only two properties, namely references to a component and a system.

There are several reasons that make the adopted approach better than the second. In short, the adopted approach gives a better understanding of what is the effect of inserting or removing a component to the configuration model: now it should be clear that inserting a component into a system results in the addition of a component type and a part definition using the component type. Other reasons for adopting the approach are more objective and stylistic in nature and are not discussed for brevity.

Connectors are mapped in the same manner as components. This time, function $connector_{AO,\ type}$ maps an Armani connector in a component types, and $connector_{AO,\ def}$ into a part definition. These functions along with rest of the mapping are defined as follows:

Definition 5-4 Connector *c* is mapped into Component type $connector_{AO,\ type}(c)$ and Part definition $connector_{AO,\ def}(c)$. The function $connector_{AO,\ type}$ is defined as follows:

$connector_{AO,\ type}(c)$:= Component type *t*, where

t.name	:= *c.name*
t.abstraction	:= CONCRETE
t.dependency	:= DEPENDENT
t.parts	:= $representation_{AO,\ def}(c.representations)$
t.ports	:= $role_{AO}(c.roles)$
t.types	:= { ADLCONNECTOR }
t.attributes	:= $property_{AO}(c.properties)$

The function $connector_{AO,\ def}$ is defined as follows:

$connector_{AO,\ def}(c)$:= Part definition d, where

$d.name$:= $c.name$

$d.types$:= $connector_{AO,\ type}(c)$

$d.cardinality$:= 1

M is modified as follows:

$components$:= $M.components \cup connector_{AO,\ type}(c)$. ■

Example. In Figure 13 (a), ADLCOMPONENT has four subtypes (`client1`, `client2`, `server`, and `c`) and ADLCONNECTOR one subtype (`rpc`) corresponding to components and connectors of the sample Armani system. These types have resulted from applying the functions $component_{AO,\ type}$ and $connector_{AO,\ type}$ to the components and connectors of the Armani system.

The effect of functions $component_{AO,\ def}$ and $connector_{AO,\ def}$ is illustrated in Figure 13 (b). There, the above-mentioned subtypes of ADLCOMPONENT and ADLCONNECTOR are related to `cs_system` and `sub` through part definitions.

Some observations about the notation used for illustrating part definitions is in place. Basically, the part-whole relationship between the two types is defined as an association between the two types. The association is of a special form, namely *composition*. The filled diamond at the end of the association lines stands for this. The semantics of composition in UML correspond to the semantics of a part definition: composition is a special case of aggregation, where a part instance is included in at most one whole at a time (Object Management Group 2001); although the configuration ontology would allow specifying that an instance of a component type can be a part of multiple component instances at a time, it is correct to disallow it for components corresponding to Armani components, as Armani has no notion of entities being in a part-whole relation with multiple wholes at a time. Further, the composite projects its identity to its parts (Object Management Group 2001). This assures that it is semantically correct to refer to the whole of a part.

Further, the compositions used for modelling part definitions are decorated with the stereotype 'part of", depicted as «part of» in the figure. The stereotype used is not one of the built-in stereotypes of UML, and should thus be defined. The definition is however omitted, as the main purpose of the stereotype here is to distinguish illustrations of part

definitions from those of port definitions that are likewise modelled with composition. ■

Ports and Roles

Ports and roles are mapped analogously to components and connectors: each port and role is mapped into a port type and a port definition. For Armani ports, function $port_{AO,\ type}$ returns the port type corresponding to the port, and $port_{AO,\ def}$ the port definition.

Definition 5-5 Port *p* is mapped into Port type $port_{AO,\ type}(p)$ and Port definition $port_{AO,\ def}(p)$. The function $port_{AO,\ type}$ is defined as follows:

$port_{AO,\ type}(p)$:= Port type *t*, where

t.name	:= *p.name*
t.abstraction	:= CONCRETE
t.types	:= { ADLPORT }
t.attributes	:= $property_{AO}(p.properties)$

The function $port_{AO,\ def}$ is defined as follows:

$port_{AO,\ def}(p)$:= Port definition *d*, where

d.name	:= *p.name*
d.types	:= $port_{AO,\ type}(p)$
d.cardinality	:= ∞

M is modified as follows:

ports := $M.ports \cup port_{AO,\ type}(p)$. ■

Argumentation similar to that given for mapping components in Armani into component types and definitions in the ontology applies to ports as well. Further, using mappings following the same pattern should make the mappings easier to comprehend.

Further, it is worth noticing that the cardinality of *t* is set to infinite. This is due to the fact that Armani does not restrict the number of attachments made to a single port in any way, but the configuration ontology allows a port individual to be connected to at most one port individual at a time. Therefore, cardinality is set to infinite to give port definitions the same semantics as ports have in Armani.

The mapping for roles is defined below. The definition is very similar to that for ports above; function $role_{AO,\ type}$ corresponds to $port_{AO,\ type}$ above, and $role_{AO,\ def}$ to $port_{AO,\ def}$.

Definition 5-6 Role *r* is mapped into Port type $role_{AO,\ type}(r)$ and Port definition $role_{AO,\ def}(r)$. The function $role_{AO,\ type}$ is defined as follows:

$role_{AO,\ type}(r)$:= Port type *t*, where

t.name	:= *t.name*
t.abstraction	:= CONCRETE
t.types	:= { ADLROLE }
t.attributes	:= $property_{AO}(p.properties)$

The function $role_{AO,\ def}$ is defined as follows:

$role_{AO,\ def}(r)$:= Port definition *d*, where

d.name	:= *r.name*
d.types	:= $role_{AO,\ type}(r)$
d.cardinality	:= ∞

M is modified as follows:

ports := $M.ports \cup role_{AO,\ type}(r)$. ■

Example. The port types corresponding to ports (`send_req_1`, `send_req_2`, `rec_req`, `p`) and those corresponding to roles (`caller`, `callee`) in the sample system are illustrated in Figure 13 (b). These port types have resulted from applying the functions $port_{AO,\ type}$ and $role_{AO,\ type}$. Further, the port definitions resulting from functions $port_{AO,\ def}$ and $role_{AO,\ def}$ can be seen in Figure 13 (c).

Observations similar to those made for modelling part definitions with UML apply likewise to port definitions. In fact, it is even better grounded to model port definitions with composition, as the ontology provides no possibility for port individuals to be simultaneously ports of two distinct component individuals. The reader should not be confused by the fact that composition is a kind of 'part-of' association in UML, as the term part has different semantics in UML and the configuration ontology. ■

Properties

The mapping for properties follows the same pattern as the mapping for components, connectors, ports and roles: a single property results in an attribute value type, and an attribute definition. The former is produced by the function $property_{AO,\ type}$, and the latter by the function $property_{AO,\ def}$. The mapping is defined as follows.

Definition 5-7 Property *p* is mapped into Attribute value type $property_{AO,\ type}(p)$ and Attribute definition $property_{AO,\ def}(p)$. The function $property_{AO,\ type}$ is defined as follows:

$property_{AO,\ type}(p)$:= Attribute value type *t*, where

t.name := *p.type.name*

t.type := *p.type* if *p.type* exists; otherwise, return nothing.

The function $property_{AO,\ def}$ is defined as follows:

$property_{AO,\ def}(p)$:= Attribute definition *d*, where

d.name := *p.name*

d.type := *p.type*, if *p.type* exists, otherwise STRING

M is modified as follows:

attributes := $M.attributes \cup property_{AO,\ type}(p)$. ■

Attachments

The construct in the configuration ontology that comes the closest to attachments is connection constraints. Therefore, attachments are mapped into constraints that are then inserted to port definitions corresponding to Armani ports as connection constraints (see Definition 5-2).

Natural language is used for expressing constraints in the configuration ontology, as there is no formal constraint language specified for it.

In the definition below, the function $attachment_{AO}$ specifies the mapping for attachments.

Definition 5-8 Attachment *a* is mapped into Constraint $attachment_{AO}(a)$. The function $attachment_{AO}$ is defined as follows:

$attachment_{AO}(a)$:= Constraint *c*, where

c.expression := $port_{AO}(a.port)$ and $role_{AO}(a.role)$ must be connected. ■

Example. In Figure 13 (c), connection constraints corresponding to attachments are drawn as lines between port definitions; see Figure 13 (d) for details of the notation. The constraint is expressed in the UML notation for constraints. ■

Representations

Finally, the mapping for representations resembles that for systems. However, in addition to creating a component type, a corresponding part definition must be as well created. The mapping into a type is carried out with the aid of function $representation_{AO,\ type}$ and that into a part definition by the function $represenatation_{AO,\ def}$.

Definition 5-9 Representation *r* is mapped into Component type $representation_{AO,\ type}(r)$ and Part definition $representation_{AO,\ def}(r)$. The function $representation_{AO,\ type}$ is defined as follows:

$representation_{AO,\ type}(r)$:= Component type *c*, where

c.name := *r. name*

c.abstraction := CONCRETE

c.dependency := DEPENDENT

c.parts := $component_{AO,\ def}(r.sys.components) \cup connector_{AO,\ def}(r.sys.connectors)$

c.types := { ADLREPRESENTATION }

c.attributes := $property_{AO}(r.sys.properties)$

Further, for each Attachment *a* in *r.sys.attachments* and each Role *q* such that *q* = *a.role*, $role_{AO}(q)$ is modified as follows:

constraints := $role_{AO}(q).constraints \cup \{ attachment_{AO}(a) \}$.

The function $representation_{AO,\ def}$ is defined as follows:

$representation_{AO,\ def}(r)$:= Part definition *d*, where

d.name := *r.sys.name*

d.types := $representation_{AO,\ type}(r)$

d.cardinality := 1

M is modified as follows:

components := $M.components \cup representation_{AO,\ type}(r)$. ■

Example. The representation `rep`, including the system `sub`, is mapped into a component type `sub` (Figure 13 (a)). Further, the representation is defined to be a part of component type `server` (Figure 13 (c)). Component type `sub` itself has a part, where the part type is `c`. ■

5.3.2 Mapping Additional Concepts of Armani to the Ontology

The name *M* is used in the definitions below to refer to a configuration model. This model can be a configuration model corresponding to an Armani system, as it was discussed in the above in Section 5.3.1. Alternatively, the model can correspond to a style (see Definition 5-14 below). In any case, the configuration model referred to as *M* in the definitions is a specific configuration model.

Design Element Types

The mappings for design element, i.e., component connector, port, and role, types are similar to the mappings of type information of design elements: e.g. the mapping of component types is similar to the function $component_{AO,\ type}$. However, the resulting component and port types differ from the types resulting from mapping design elements in that the types are abstract. The abstractness serves two purposes: first, component types corresponding to design element types and design element types are distinguished from each other; second, there would be no use in making these types concrete, as all the design elements are created a concrete type of their own when mapping them to the ontology.

Component types in Armani are mapped into component types in the configuration ontology by the function $component_{AO,\ type}$ as follows.

Definition 5-10 Component type *t* such that is mapped into Component type $componentType_{AO}(t)$. The function $componentType_{AO}$ is defined as follows:

$componentType_{AO}(t)$:= Component type *c*, where

c.name	:= *t.name*
c.abstraction	:= ABSTRACT
c.dependency	:= DEPENDENT
c.parts	:= $representation_{AO}(t.representations)$
c.ports	:= $port_{AO}(t.ports)$
c.types	:= { ADLCOMPONENT } ∪ $componentType_{AO}(t.types)$
c.attributes	:= $property_{AO}(t.properties)$

M is modified as follows:

components	:= *M.components* ∪ $componentType_{AO}(t)$
constraints	:= *M.constraints* ∪ $invariant_{AO}(t.invariants)$. ■

The function $invariant_{AO}$ referred to in the definitions is defined below. In short, it maps Armani invariants into constraints in the configuration ontology.

The mappings for other design element types are similar to both the above mapping of component types and the corresponding mappings of the design elements themselves presented above (see Definition 5-4 for connectors, Definition 5-5 for ports, and Definition 5-6 for roles). In short, all design element types are mapped into abstract

types; in other respects the mappings are the same as those for design elements. The mappings are not given explicitly here.

It should be noted that the above definition handles also the issue that component types can be defined supertypes in Armani. In more detail, the fact that *c.types* is assigned to the value of the union of ADLCOMPONENT and the set of types resulting from mapping the component types declared by *c* as its supertypes.

Design elements, i.e., components, connectors, ports, and roles, declaring types are mapped similarly to ordinary design elements (see Definition 5-3 for components, Definition 5-4 for connectors, Definition 5-5 for ports, and Definition 5-6 for roles) with the exception that the concrete type that are created in the configuration ontology is defined to be a subtype of all the abstract types corresponding to the types declared by the design element in Armani.

For components, the mapping is carried out by functions $typedComponent_{AO,\ type}$ and $typedComponent_{AO,\ def}$, of which the former returns the component type, and the latter the part definition. Formally, the mapping is defined as follows.

Definition 5-11 Component *c* that declares a set of types is mapped into Component type $typedComponent_{AO,\ type}(c)$ and $typedComponent_{AO,\ def}(c)$. The function $typedComponent_{AO}$ is defined as follows:

$typedComponent_{AO,\ type}(c)$:= $component_{AO,\ type}(c)$ modified with
    types := $component_{AO,\ type}(c).types \cup componentType_{AO}(c.types)$

The function $component_{AO,\ type}$ is defined in Definition 5-3.

The function $typedComponent_{AO,\ def}$ is defined as follows:

$typedComponent_{AO,\ def}(c)$:= $component_{AO,\ def}(c)$ modified with
    types := $typedComponent_{AO,\ type}(c)$

M is modified as follows:

M.components := $M.components \cup typedComponent_{AO,\ type}(c)$. ■

The modification made to $component_{AO,\ type}$ are needed to reflect the fact that the component instance is declared to be of a number of types.

Again, mapping connectors, ports and roles that declare types is similar to mapping components declaring types and the mappings for connectors, ports, and roles not declaring entities (see, once again, Definition 5-4 for connectors, Definition 5-5 for

ports, and Definition 5-6 for roles). In short, mappings for design elements declaring types are the same as the mappings for design elements not declaring types, with the exception that the resulting concrete type in the ontology is defined to be a subtype of the abstract types corresponding to the types declared by the element in Armani. These mapping are not given explicitly.

Property Types

Property types in Armani are mapped into attribute value types in the ontology by the function $propertyType_{AO}$.

Definition 5-12 Property type *p* is mapped into Attribute value type $propertyType_{AO}(p)$. The function $propertyType_{AO}$ is defined as follows:

$propertyType_{AO}(p)$:= Attribute value type *t*, where

t.name := *p.type.name*

t.type := *p.type*.

M is modified as follows:

M.attributes := $M.attributes \cup propertyType_{AO}(p)$. ■

The mapping of property types is similar to mapping design element types in that similar mapping was already seen above when mapping the type information embedded in properties.

Invariants

Invariants in Armani are mapped to constraints in the configuration ontology. However, the details of how the content of expressions is mapped cannot be given, as there is no constraints language specified in the ontology.

Invariants are mapped by function $invariant_{AO}$ as specified in the definition below.

Definition 5-13 Invariant *i* in a context is mapped into Constraint $invariant_{AO}(i)$. The function $invariant_{AO}$ is defined as follows:

$invariant_{AO}(i)$:= Constraint *r*, where

r.expression := the semantics of *i* must hold the in 'part' of the configuration model corresponding to the context. ■

The above definition refers to the context in which an invariant is defined. In practical terms, an invariant can be defined within a design element type or a style. In the former

case, the invariant should be mapped to a constraint concerning the individuals of the type the design element type is mapped to. In the latter case, the invariants are mapped to concern entire configuration models: as will be shown next, styles are mapped into partial configuration models.

Styles

Styles are mapped into configuration models that are partial in the sense that they do not correspond to Armani systems as such, but are supposed to be used in conjunction with configuration models corresponding to Armani systems.

Formally, the mapping is carried out by the function $style_{AO}$ defined below.

Definition 5-14 Style *s* is mapped into Configuration model $style_{AO}(s)$. The function $style_{AO}$ is defined as follows:

$style_{AO}(s)$:= Configuration model M, where

$M.component := componentType_{AO}(s.components) \cup connectorType_{AO}(s.connectors)$

$M.ports := portType_{AO}(s.ports) \cup roleType_{AO}(s.roles)$

$M.attributes := propertyType_{AO}(s.properties)$

$M.constraints := invariant_{AO}(s.invariants)$.

In addition, *M* contains the component and port types of Figure 12.■

As can be seen in the definition, *M* contains component and port types corresponding to different design element types defined in the style. Further, *M* includes attribute value types corresponding to the property types, and constraints corresponding to invariants defined in the style. Further, the general definition of Figure 12 are needed, as the they serve as the supertypes of abstract component and port types in *M*.

Systems that are of a one or a number of styles are mapped as ordinary systems with the exception that the resulting configuration model is supplemented with the configuration models corresponding to the declared styles. Formally, the mapping is carried out by the function $system_{AO}$, the same function used for mapping styles that are of no style.

Definition 5-15 System *s* that declares a set of styles is mapped into Configuration model *M* and Component type $system_{AO}(s)$. Initially, *M* contains the component and port types of Figure 12 and the compatibility definitions of Definition 5-1. The function $system_{AO}$ is defined in Definition 5-2.

M is modified as follows:

$$M := M \cup style_{AO}(s.styles). \blacksquare$$

In the above equation, the semantics of the union of configuration models is that the configuration models resulting from mapping the styles are included in the configuration model resulting from mapping the system.

Table 9 Summary of mapping Armani to the configuration ontology

Armani		Configuration ontology
Entity	Properties	
System		Concrete and independent subtype of the ADLSYSTEM, configuration model
	Component	Part definitions of ADLCOMPONENT type in the subtype of ADLSYSTEM
	Connector	Part definitions of ADLCONNECTOR type in the subtype of ADLSYSTEM
	Attachment	Connection constraints in port definitions corresponding to ports in Armani
	Style	Partial configuration model included in the configuration model
Component		Concrete and dependent subtype of ADLCOMPONENT
	Port	Port definition of ADLPORT type in the subtype of ADLCOMPONENT
	Representation	Part definition of ADLREPRESENTATION type in the subtype
	Type	The type is a subtype of abstract types
Connector		Similarly as componets, but types are ADLCONNECTOR and ADLROLE
Port		Concrete subtype of ADLPORT, port definition in an ADLCOMPONENT type
Role		Concrete subtype of ADLROLE, port definition in an ADLCONNECTOR type
Representation		Conrete subtype of ADLREPRESENTATION, and a part definition
	Binding	**No mapping**
	Other properties	Similarly as in System
Property		Attribute definition in the corresponding type
Component type		Abstract and dependent subtype of ADLCOMPONENT
	Port	Port definition in the subtype of ADLCOMPONENT
	Invariant	Constraints concerning the subtype of ADLCOMPONENT
	Heuristic	**No mapping**
Connector type		Similarly as Componen type, but types are ADLCONNECTOR ADLROLE
Port type		Abstract and dependent subtype of ADLPORT
	Invariant	Constraints concerning the subtype of ADLPORT
	Heuristic	**No mapping**
	Type	The subtype is a subtype of other abstract types
Role type		Similarly as Port type, but type is ADLROLE
Property type		Attribute value type
Style		Partial configuration model
	Properties	Items in the partial configuration model

Table 9 summarises the above mappings of this and the previous section. As has been highlighted with **bold** typeface, no mapping could be found for binding between the interfaces in representation and those in the represented entities, and for heuristics.

These issues are discussed in more detail in Section '6.2 Unmapped Concepts and Potential Extensions'.

5.3.3 Ontological Constraints

This section represents a set of constraints that serve as an incremental extension to the configuration ontology. The extension enables representing architectures with the semantics equivalent to Armani.

Definition 5-16 There are no resource and no function types in the configuration model. ■

The above constraint is set is due to the fact that when the mapping from Armani to the ontology, nothing was mapped into resources and functions. There is no obvious corresponding concept in Armani for functions and resources.

Next, as a mapping between a single Armani system or style and a configuration model is desired, it must be asserted that there is at most one subtype of ADLSYSTEM in the configuration model. A model without a subtype of ADLSYSTEM corresponds to an Armani style, and a model where there is exactly one subtype of ADLSYSTEM corresponds to a system. Further, configuration models corresponding to systems and styles both have some invariant properties.

Definition 5-17 There exists at most one subtype of ADLSYSTEM in the configuration model. If there is a subtype of ADLSYSTEM, it must be independent and concrete. If there exists no subtype of ADLSYSTEM, there may not exist any concrete component or port types either. ■

The fact that components, connectors and representations are always part of something else is reflected in the following constraint.

Definition 5-18 Only subtypes of ADLSYSTEM are allowed to be independent. ■

The taxonomy of types must be restricted.

Definition 5-19 Each component type must be a subtype of exactly one of the following types: ADLCOMPONENT, ADLCONNECTOR, ADLSYSTEM, ADLREPRESENTATION. Further, component types are not allowed to define concrete supertypes. ■

The need for these constraints can be seen by observing that if a component type were not be a subtype of any of the above-mentioned types, there would be no unambiguous mapping from the type to the ADLs. The same condition would result if some component type had more than one supertype in the above-mentioned set of types. A similar argument applies to port types.

Definition 5-20 Each port type must be a subtype of exactly one of the following types: ADLPORT, ADLROLE. Further, port types may not define concrete supertypes. ■

The above requirement is due to the fact that only design elements are mapped into concrete types, and, on the other hand, only design element types may be declared as types of design elements and supertypes of design element types.

Next, the part structure of component individuals must be constrained. In detail, the entities of ADLs represented by component types in the ontology may define only the following parts:

- Systems, both in representations and other, may define components and connectors.
- Components and connectors may define only ports and roles, respectively, and representations.

The corresponding constraints in the configuration model are:

Definition 5-21 For each Part definition *d* of a subtype of ADLSYSTEM or ADLREPRESENTATION, the set of possible part types of *d* must include only concrete subtypes of either ADLCOMPONENT or ADLCONNECTOR. ■

Definition 5-22 Subtypes of ADLSYSTEM and ADLREPRESENTATION must not define ports. ■

Definition 5-23 For each Part definition *d* of a subtype of ADLCOMPONENT or ADLCONNECTOR, the set of possible part types of *d* must include only subtypes of ADLREPRESENTATION. ■

Definition 5-24 For each Port definition *d* of a subtype of ADLCOMPONENT, *d.types* must include only concrete subtypes of ADLPORT. ■

Definition 5-25 For each Port definition *d* of a subtype of ADLCONNECTOR, *d.types* must include only concrete subtypes of ADLROLE. ■

Further, it must be required that in all part and port definitions, the set of possible part or port types, respectively, is of size one. Otherwise, the mapping from the type to the ADLs would be ambiguous.

Definition 5-26 For each Part definition *d*, *d.types* must be of size one, and and *d.cardinality* must be one. ■

Definition 5-27 For each Port definition *d*, *d.types* must be of size one, and *d.cardinality* must be infinite. ■

As already stated when defining mapping from Armani to the ontology, port definitions must have an infinite cardinality due to the fact that Armani does not restrict the number of connections made to a single port or role.

As the two above definitions guarantee that the set of possible types in part and port definitions contains a single type, the type will be for simplicity referred by the name *type* (compare with Table 3).

5.3.4 Mapping the Ontology to Armani

This section presents how a configuration model can be mapped into an Armani system or style.

Component Types

As can be recalled from above, different entities in Armani were mapped into component types. In the following definitions, component types are mapped into different entities in Armani based on their supertypes.

First, subtypes of ADLSYSTEM are mapped into Armani systems. Formally, the mapping is carried out by the function $ADLSystem_{OA}$.

Definition 5-28 Component type *c* in Configuration model *M* such that *c.type* is a subtype of ADLSYSTEM is mapped into System $ADLSystem_{OA}(c, M)$. The function $ADLSystem_{OA}$ is defined as follows:

$ADLSystem_{OA}(c, M)$:= System *s*, where

s.name := *c.name*

s.properties := $attribute_{OA}$*(c.attributes)*
s.components := $ADLComponent_{OA,\ def}$*(c. parts)*
s.connectors := $ADLConnector_{OA,\ def}$*(c.parts)*
s.attachments := $connections_{OA}$*(M, c)*. ■

As is apparent from the above definition, the attributes and parts of *c* become properties, components, and connectors of *s*; functions $attribute_{OA}$, $ADLComponent_{OA,\ def}$, and $ADLConnector_{OA,\ def}$, respectively, defined below are used to map entities in configuration models into these entites. Further, the attachments of *s* are extracted from the configuration model by the function $connections_{OA}$, likewise defined below.

Example. The same client-server example that was used for demonstrating mapping Armani to the ontology is used to demonstrate the mapping back from the ontology to Armani. Figure 3 presented the system in Armani, and Figure 13 the same system mapped from Armani in to the configuration ontology. When giving examples in this section, the same figures are used.

Applying the above definition results in `cs_system` component type being mapped into the Armani by the same name. ■

Abstract component and port types in configuration models are mapped into design element types. For component types, the function $ADLComponent_{OA,\ type}$ defined below is used to perform the mapping.

Definition 5-29 Abstract Component type *t* such that *t* is a subtype of ADLCOMPONENT is mapped into Component type $ADLComponent_{OA,\ type}(t)$. The function $ADLComponent_{OA,\ type}$ is defined as follows:

$ADLComponent_{OA,\ type}(t)$:= Component type *c*, where
c.name := *t.name*
c.properties := $attribute_{OA}$*(t.properties)*
c.ports := $ADLPort_{OA,\ def}$*(t.ports)*
c.reps := $ADLRepresentation_{OA,\ def}$*(t.parts)*
c.types := $ADLComponent_{OA,\ type}$*(t.types)*. ■

The functions $attribute_{OA}$, $ADLPort_{OA,\ def}$, and $ADLRepresentation_{OA}$ referred to in the above definition map the attribute definitions, port definitions, and representations, respectiviely of *c* into corresponding entities in Armani, i.e., properteis, ports, and representations. These functions are defined below in this section. Further, the

supertypes of *t* are mapped into supertypes of *c* by recursively applying the $ADLComponent_{OA,\ type}$ function.

Types that are subtypes of ADLCONNECTOR are mapped into connector types similarly as subtypes of ADLCOMPONENT to component types above. Of course, the mapping is done into Armani connectors, and the ports defined by *c* are mapped into roles of connectors, using a function similar to $ADLPort_{OA,\ def}$.

Port Types

The definition for mapping abstract subtypes of ADLPORT is given below. In essence, the mapping is carried out by function $ADLPort_{OA,\ type}(t)$.

Definition 5-30 Abstract Port type *t* such that *t* is a subtype of ADLPORT is mapped into Port type $ADLPort_{OA,\ type}(t)$. The function $ADLPort_{OA,\ type}$ is defined as follows:

$ADLPort_{OA,\ type}(t)$:= Port type *p*, where

p.name := *t.name*

p.properties := $attribute_{OA}(t.properties)$

p.types := $ADLPort_{OA,\ type}(t.types)$. ■

The mapping for subtypes of ADLROLE is the same as that for subtypes of ADLPORT. Therefore, the mapping is not defined explicitly.

Part Definitions

Part definitions of component types in the configuration ontology are mapped into components, connectors, and representations. First, the definition that results in components is given. Formally, the definition is carried out by the function $ADLComponent_{OA,\ def}$.

Definition 5-31 Part definition *p* such that *p.type* is a subtype of ADLCOMPONENT is mapped into Component $ADLComponent_{OA,\ def}(p)$. The function $ADLComponent_{OA,\ def}$ is defined as follows:

$ADLComponent_{OA,\ def}(p)$:= Component *c*, where

c.name := *p.name*

c.properties := $attribute_{OA}(p.type.attributes)$

c.ports := $ADLPort_{OA,\ def}(p.type.ports)$

c.reps := $ADLRepresentation_{OA,\ def}(p.type.parts)$

c.types := $ADLComponent_{OA,\ type}(p.type.types)$. ■

As can be seen from the above definition, most of the information of the resulting Armani component is obtained from *p.type*. In fact, the above definition greatly resembles the definition for mapping abstract subtypes of ADLCOMPONENT (Definition 5-29).

Part definitions where the part type is a subtype of ADLCONNECTOR are mapped into connectors. Once again, the mapping is analogous to that of component, and is therefore not given explicitly; the differences between the mappings for components and connectors are similar to those discussed above in conjunction with Definition 5-29.

Example. The part definitions between types `cs_system` and `client1`, `client2`, `rpc`, and `server` are mapped components and ports of `cs_system` in Armani; the resulting elements have the same names as the above-mentioned types. ■

Finally, part definitions where the part type is a subtype of ADLREPRESENTATION are mapped into representations using the following definition and the function $ADLRepresentation_{OA,\ def}$ defined in it.

Definition 5-32 Part definition *p* in Configuration model *M* such that *p.type* is a subtype of ADLREPRESENTATION is mapped into Representation $ADLRepresentation_{OA,\ def}(p, M)$. The function $ADLRepresentation_{OA,\ def}$ is defined as follows:

$ADLRepresentation_{OA,\ def}(p, M)$:= Representation *r*, where

r.name := *p.name*

r.sys.properties := $attribute_{OA}(p.type.attributes)$

r.sys.name := *p.type.name*

r.sys.components := $ADLComponent_{OA,\ def}(p.type.parts)$

r.sys.connectors := $ADLConnector_{OA,\ def}(p.type.parts)$

r.sys.attachments := $connections_{OA}(M, p)$. ■

This mapping closely resembles the mapping for system; see Definition 5-28 and the discussion following it.

Example. The part definition called `rep` between `server` and `sub` is mapped into representation `rep` of component `server` in the Armani system `cs_system`. ■

Port Definitions

Port definitions are mapped into ports and roles in Armani. For ports, the mapping is carried out by the function $ADLPort_{OA,\ def}$ as follows.

Definition 5-33 Port definition *p* such that *p.type* is a subtype of ADLPORT is mapped into Port $ADLPort_{OA,\ def}(p)$. The function $ADLPort_{OA,\ def}$ is defined as follows:

$ADLPort_{OA,\ def}(p)$:= Port *r*, where

r.name := *p.name*

r.properties := $attribute_{OA}(p.type.attributes)$.

r.types := $ADLPort_{OA,\ type}(p.type.types)$. ■

Port definitions where the port type is a subtype of ADLROLE are mapped into roles in Armani. The mapping is similar to that for ports, and is thereby omitted.

Example. The port definition between `client1` and `send_req_1` is mapped into port named `send_req_1` in the Armani system. Correspondingly, `send_req_2` is mapped into port by the same name in `client2`, `p` into port in `c`, and `caller` and `callee` into roles in `rpc`.

It should be noticed that two ports resulting from the mapping, namely `send_req_1` and `send_req_2`, have different names than the original ports in the components (`send_req`). Actually, this condition was introduced already when mapping the Armani system in the configuration ontology: as there were two ports with the same name in the Armani system, they were implicitly given different names so that they would not have the same name in the configuration ontology. For practical purposes, some general scheme for resolving this kind of naming conflicts should be developed. ■

Attribute Definitions

Attribute definitions occurring in types are mapped into properties. Formally, the mapping is carried out by the function $attribute_{OA}$.

Definition 5-34 Attribute definition *a* is mapped into Property $attribute_{OA}(a)$. The function $attribute_{OA}$ is defined as follows:

$attribute_{OA}(a)$:= Property *p*, where

p.name := *a.name*

p.type := *a.type*. ■

Constraints

Constraints that imply that port individuals must be connected are mapped into attachments in systems. Formally, the mapping is defined below in terms of the function $connections_{OA}$.

Definition 5-35 The function $connections_{OA}(M, t)$ takes Configuration model M and Component type t as arguments and returns a set of Attachments s: $s := \{ a \mid a.port = ADLPort_{OA}(p)$ and $a.role = ADLRole_{OA, def}(r)$, p and r are Port definitions in M such that p and r are port definitions in parts of t, and M implies that p and r must be connected $\}$. ■

Characterising the constraints that should be mapped into attachments in more detail is not possible. First, the assumption that such constraints would always be connection constraints in port definitions is not valid. Second, constraints implying that two ports must be connected cannot be assumed to have any certain pattern, especially when there is no specific constraint language in the configuration ontology.

Example. The constrains between port definitions in Figure 13 are mapped into attachments. Explicitly, the constraint between `send_req1` and `caller`, and `send_req_2` and `caller` are mapped into attachments between ports and roles having the same name in the Armani system. Similarly, the constraint between `callee` and `rec_req` is mapped into a constraint between `callee` and `rec_req`.

It should be noticed that the model resulting from mapping the client-server system back from the ontology to Armani does not include the binding between `p` and `rec_req`; this is due to the fact that the binding could not be mapped into the ontology. ■

Other constraints are mapped into invariants. Once again, it is not possible to show exactly how different constraints are mapped, as the ontology does not specify a constraint language. At the abstract level, however, constraints are mapped as follows using the function $constraint_{OA}$.

Definition 5-36 Constraint c is mapped into Invariant $constraint_{OA}(c)$. The function $constraint_{OA}$ is defined as follows:

$constraint_{OA}(c)$:= Invariant i, where
i := $c.expression$. ■

Configuration Models without Subtype of ADLSystem

Configuration models that contain no subtypes of ADLSYSTEM are mapped into styles. Formally, the mapping is carried out by the function $model_{OA}$.

Definition 5-37 Configuration model *m* such that *m* contains no subtypes of ADLSYSTEM is mapped into Style $model_{OA}(m)$. The function $model_{OA}$ is defined as follows:

$model_{OA}(m)$:= Style *s*, where

$s.components := ADLComponent_{OA,\ type}(m.components)$

$s.connectors := ADLConnector_{OA,\ type}(m.components)$

$s.ports := ADLPort_{OA,\ type}(m.ports)$

$s.roles := ADLRole_{OA,\ type}(m.ports)$

$s.invariants := constraint_{OA}(m,$ GLOBAL$)$. ■

In the above definition, design element types and constraints are mapped into the content of the style using functions that have been defined above. An observation about the way how the $constraint_{OA}$ is used is in place: no component type is specified, but the identifier GLOBAL is used to express that constraints that are not specific to any design element type are mapped into invariants of the style. Constraints that are specific to some design element type are mapped in the respective functions mapping these types.

5.4 Mapping between Wright and the Ontology

In this section, a similar handling as was given in the above section for Armani, will given for Wright.

5.4.1 Mapping Basic Concepts of Wright to the Ontology

This section presents a mapping from Wright to the configuration ontology.

Overview of the Mapping

As was found out above (see, e.g. in Table 1), Armani and Wright have the same set of basic concepts. Consequently, it should come as no surprise that Wright is mapped into the ontology in a similar manner to Armani.

General Definitions

The taxonomies for component and port types presented in Figure 12 and the compatibility definitions of Definition 5-1 apply to Wright as well.

Systems

The mapping of Wright systems is similar to the mapping of Armani systems (see Definition 5-2). A minor difference is that Wright systems cannot be defined properties. Formally, the function $system_{WO}$ defined below is used to carry out the mapping.

Definition 5-38 System *s* is mapped into Configuration model *M* and Component type $system_{WO}(s)$. Initially, *M* contains the component and port types of Figure 12 and the compatibility definitions of Definition 5-1. The function $system_{WO}$ is defined as follows:

$system_{WO}(s)$:= Component type *c*, where

$c.name$:= $s.name$

$c.abstraction$:= CONCRETE

$c.dependency$:= INDEPENDENT

$c.parts$:= $componentInstance_{WO}(s.components) \cup connectorInstance_{WO}(s.connectors)$

$c.types$:= { ADLSYSTEM }

Further, for each Attachment *a* in *s.attachments*, and each Role *r* such that *r* := *a.role*, $role_{WO}(r)$ is modified as follows:

$constraints$:= $role_{WO}(r).constraints \cup \{ attachment_{WO}(a) \}$.

M is modified as follows:

$components$:= $M.components \cup system_{WO}(s)$. ■

As when mapping from Armani, *M* refers to a specific configuration model, i.e. to the configuration model of the above definition.

In the above definition, functions $componentInstance_{WO}$ and $connectorInstance_{WO}$ not introduced so far are used for specifying the part definitions of the *c*. The function $attachment_{WO}$ returns a constraints based on the attachments it receives as its input. All these functions are defined later in this section.

Example. Figure 14 depicts the mapping of the client-server from Wright to the configuration ontology. The legend from Figure 13 (d) applies to this figure as well. The

mapping of the system in itself is the same as for Armani; the differences between languages become visible later. ■

Components and Connectors

In Wright, component types and their instances are separated from each other. The following definition defines a mapping for component types; the function $componentType_{WO}$ maps the component types in Wright into component types in the configuration ontology.

Definition 5-39 Component type *c* is mapped into Component type $componentType_{WO}(c)$. The function $componentType_{WO}$ is defined as follows:

$componentType_{WO}(c)$:= Component type *t*, where

t.name	:= *c.name*
t.abstraction	:= CONCRETE
t.dependency	:= DEPENDENT
t.parts	:= $representation_{WO}(c.rep)$
t.ports	:= $port_{WO}(c.ports)$
t.types	:= { ADLCOMPONENT }

M is modified as follows:

components := $M.components \cup componentType_{WO}(c)$. ■

The function $componentType_{WO}$ is essentially the same as function $component_{AO,\ type}$ in Definition 5-3. The functions $representation_{WO}$ and $port_{WO}$ defined later in this section map the representation and ports of *c* into the configuration ontology.

Instances of components are mapped into part definitions. The function $componentInstance_{WO}$ defined below is the formal means for carrying out this task.

Definition 5-40 Component instance *c* is mapped into Part definition $componentInstance_{WO}(c)$. The function $componentInstance_{WO}$ is defined as follows:

$componentInstance_{WO}(c)$:= Part definition *d*, where

d.name	:= *c.name*
d.types	:= $componentType_{WO}(c)$
d.cardinality	:= *c.cardinality*. ■

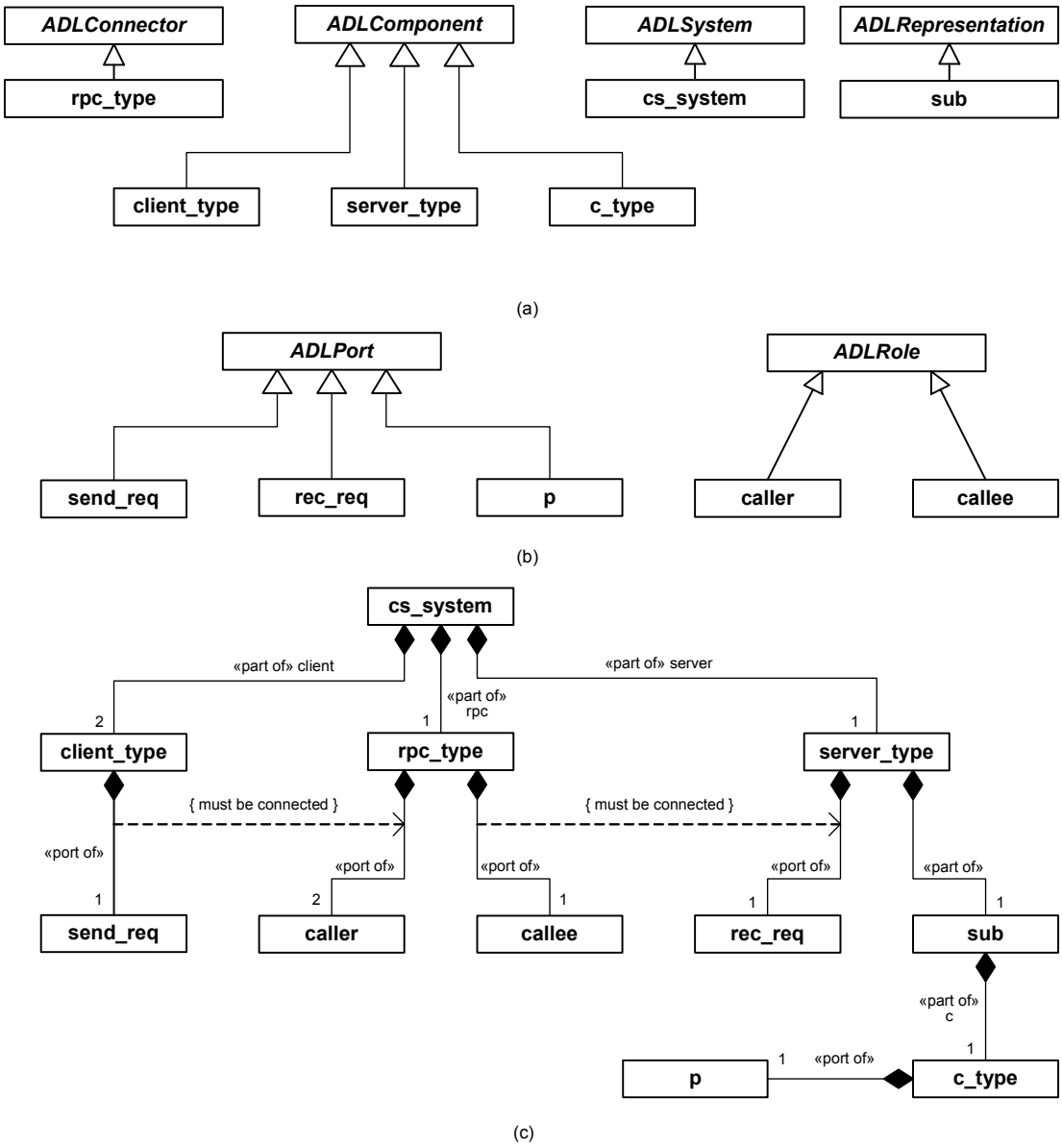


Figure 14 Mapping the client-server system from Wright to the configuration ontology. (a) The taxonomy of component types. (b) The taxonomy of port types. (c) The part and port structure in the configuration model.

Once again, the mapping for both connector types and instances is essentially the same as that for components; the only differences are that ADLCONNECTOR is used instead of ADLCOMPONENT, and connectors contain roles instead of ports. Therefore, the explicit mapping for connector types and instances is omitted.

Example. Figure 14 (a) illustrates that the component types `client_type`, `server_type`, and `c_type` are mapped into subtypes of ADLCOMPONENT with their respective names. Correspondingly, connector type `rpc_type` is mapped into a subtype of ADLCONNECTOR by the same name.

Further, it can be seen from Figure 14 (c) that the types mentioned in the above section are related to `cs_system` and `sub` through part definitions. Particularly, it should be noted that `client_type` is a part of `cs_system` with a cardinality of two; this corresponds to the array of size two of `client_type` instances in the Wright system (see Figure 5). ■

Ports and Roles

Unlike for components and connectors, the types and instances of ports and roles are not separated for ports and roles. Therefore, the mapping for these concepts follows the same lines as the mapping for ports and roles in Armani. The mapping for ports is given below; formally, the mapping is carried out by the functions $port_{WO,\ type}$ and $port_{WO,\ def}$.

Definition 5-41 Port *p* is mapped into Port type $port_{WO,\ type}(p)$ and Port definition $port_{WO,\ def}(p)$. The function $port_{WO,\ def}$ is defined as follows:

$port_{WO,\ type}(a)$:= Port type *r*, where

r.name	:= *p.name*
r.abstraction	:= CONCRETE
r.types	:= { ADLPORT }

The function $port_{WO,\ def}$ is defined as follows:

$port_{WO,\ def}(p)$:= Port definition d, where

d.name	:= *p.name*
d.types	:= $port_{WO,\ type}(p)$
d.cardinality	:= *c.cardinality*

M is modified as follows:

ports := *M.ports* ∪ $port_{WO,\ type}(p)$. ■

The mapping for roles is once again omitted, as it is practically the same as the mapping for ports.

Example. Figure 14 (b) illustrates port types in the configuration model. The subtypes of ADLPORT correspond to the ports of the Armani system `cs_system`, and those of ADLROLE to the roles of the same system. Figure 14 (c), in turn, depicts the port definitions. Especially worth noticing is the port definition involving `send_req` as the port type: this definition has a cardinality of two, which reflects the fact that component type `client` was declared to have a port array named `send_req` of size two. ■

Attachments

As in Armani, attachments in Wright systems are mapped into constraints. Further analogously to Armani, the attachments are placed in the port definitions, as this is the natural place for the constraints; the placement of the constraints into the port definitiosn was specified above in Definition 5-38. Formally, the mapping is carried out by the function $attachment_{WO}$ defined below.

Definition 5-42 Attachment *a* is mapped into Constraint $attachment_{WO}(a)$. The function $attachment_{WO}$ is defined as follows:

$attachment_{WO}(a)$:= Constraint *c*, where

c.expression := $port_{WO}(a.port)$ and $role_{WO}(a.role)$ must be connected. ■

Example. Connection constraints resulting from mapping Wright attachments to the ontology are illustrated in Figure 14 (c). ■

Representations

The representations in Wright are mapped in a manner similar to Armani representations. Now, the function $representation_{WO,\ type}$ is used to create a component type based on the represenatation, and $representation_{WO,\ def}$ the corresponding part definition.

Definition 5-43 Representation *r* is mapped into Component type $representation_{WO,\ type}(r)$ and Part definition $representation_{WO,\ def}(r)$. The function $representation_{WO,\ type}$ is defined as follows:

$representation_{WO}(r)$:= Component type *c*, where

d.name := *r.sys.name*

c.abstraction := CONCRETE

c.dependency := DEPENDENT

c.parts := $componentInstance_{WO}(r.sys.components) \cup connectorInstance_{WO}(r.sys.connectors)$

c.types := { ADLREPRESENTATION }

Further, for each Attachment *a* in *r.sys.attachments* and each Role *q* such that *q* = *a.role*, $role_{WO}(q)$ is modified as follows:

constraints := $role_{WO}(q).constraints \cup \{ attachment_{WO}(a) \}$.

The function $representation_{WO,\ def}$ is defined as follows:

representation$_{WO,\ def}$*(r)* := Part definition *d*, where

d.name := *r.sys.name*

d.types := *representation*$_{WO,\ type}$*(r)*

d.cardinality := 1

M is modified as follows:

components := *M.components* ∪ *representation*$_{WO}$*(r)*. ■

Example. Figure 14 (a) illustrates that `sub` is defined to be a subtype of ADLREPRESENTATION. Additionally, Figure 14 (c) depicts that component type `server` is defined a part where the part type is `sub`; the type `sub` itself is defined to have a part corresponding to the component `c` in the original Wright system. ■

5.4.2 Mapping Additional Concepts of Wright to the Ontology

As in Armani, styles in Wright are collections of design vocabulary and constraints, and the systems of the style can use the design vocabulary and must follow the constraints. Therefore, Wright styles are mapped to partial configuration models, just like Armani styles above. Of course, the content of styles cannot be straightforwardly mapped using the same definitions as for Armani above, as the constructs contained in styles are different in the two ADLs.

Component and Connector Types

Component and connector types can be mapped by following the same process as for mapping the types in the context of a system. The mapping for component types was defined in Definition 5-39. The mapping for connector types was stated to be similar to that for component types already in conjunction with the above-mentioned definition.

Syntactic Constraints

Definition 5-44 Syntactic constraint *c* is mapped into Constraint *constraint*$_{WO}$*(c)*. The function *constraint*$_{WO}$ is defined as follows:

constraint$_{WO}$*(c)* := Constraint *r*, where

r.expression := the semantics of *c* must hold. ■

Styles

As in Armani, styles are mapped into partial configuration models. The configuration model corresponding to a style is then included in the configuration model resulting

from mapping a system of the style. These mappings are formalised in the following two definitions: in the first definition, the function $style_{WO}$ maps Wright styles into configuration models, and in the second definition the function $system_{WO}$ already defined in Definition 5-38 serves as the basis for mapping systems with styles.

Definition 5-45 Style *s* is mapped into Configuration model $style_{WO}(s)$. The function $style_{WO}$ is defined as follows:

$style_{WO}(s)$:= Configuration model M, where

$M.components$:= $componentType_{WO}(s.components) \cup connectorType_{WO}(s.connectors)$

$M.constraints$:= $constraint_{WO}(s.constraints)$.

In addition, *M* contains the component and port types of Figure 12. ■

Definition 5-46 System *s* that declares a set of styles is mapped into Configuration model *M* and Component type $system_{WO}(s)$. Initially, *M* contains the component and port types of Figure 12 and the compatibility definitions of Definition 5-1. The function $system_{WO}$ is defined in Definition 5-2.

M is modified as follows:

M := $M \cup style_{WO}(s.styles)$. ■

Table 10 summarises the above mapping. All the component and port types are concrete. As has been highlighted with **bold** typeface, no mapping could be defined for bindings between the interfaces in representations and those in the represented entities. Further, no behaviour-related aspect was found a mapping: these include process descriptions in components, connectors, ports and roles, interface types, and semantic constraints.

5.4.3 Ontological Constraints

This section presents the ontological constraints for Wright. As the constraints and the reasons underlying them are most similar to those in Armani, most of the constraints are lightly motivated and explained. For more details, the reader is instructed to refer to Section '5.3.3 Ontological Constraints', where constraints for Armani are introduced.

As in the case of Armani, there is no obvious corresponding concept in Wright for functions and resources.

Table 10 Summary of mapping Wright to the configuration ontology

Wright		Configuration ontology
Entity	Properties	
System		Independent subtype of ADLSYSTEM, configuration model
	Component type	Dependent subtype of ADLCOMPONENT
	Connector type	Dependent subtype of ADLCONNECTOR
	Component instance	Part definition of type ADLCOMPONENT in subtype of ADLSYSTEM
	Connector instance	Part definition of type ADLCONNECTOR in subtype of ADLSYSTEM
	Attachment	Connection constraint in port definitions corresponding to ports
	Style	Partial configuration models included in the configuration model
Component type		Dependent subtype of ADLCOMPONENT
	Port	Port definition of the subtype of ADLCOMPONENT
	Computation	**No mapping**
	Representation	Part definition of the subtype
Connector type		Similarly as Component type, but the type is ADLCONNECTOR
Component instance		Part definition in the subtype of ADLSYSTEM
	Type	The type of the part definition
	Cardinality	The cardinality of the part definition
Connector instance		Similarly as Component instance
Port		Concrete subtype of ADLPORT, port definition
	Process	**No mapping**
	Cardinality	Cardinality in the port definition
Role		Similary as Port, but a subtype of ADLROLE
Representation		Dependent subtype of ADLREPRESENTATION, a part definition
	Binding	**No mapping**
	Properties	Similarly as the properties of systems
Attachment		Connection constraints in port definition with ADLPORT type
Style		Partial configuration model
	Syntactic constraint	General constraint in the configuration model
	Semantic constraint	**No mapping**
	Interface type	**No mapping**
	Other properties	Items in the configuration model

Definition 5-47 There are no resource and no function types in the configuration model. ■

Each configuration model corresponds either to a system or a style.

Definition 5-48 There exists at most one subtype of ADLSYSTEM in the configuration model. If there is a subtype of ADLSYSTEM, it must be independent and concrete. ■

The fact that components, connectors and representations are always part of something else is reflected in the following constraint.

Definition 5-49 Only subtypes of ADLSYSTEM are allowed to be independent. ■

The taxonomy of types must be restricted.

Definition 5-50 Each component type must be a direct concrete subtype of exactly one of the following types, except for these types themselves: ADLCOMPONENT, ADLCONNECTOR, ADLSYSTEM, ADLREPRESENTATION. ■

Definition 5-51 Each subtype of Port type must be a direct and concrete subtype of exactly one of the following types, except for these types themselves: ADLPORT, ADLROLE. ■

The constraint that component and port types are not allowed to define supertypes stems from the lack of taxonomy of types in Wright. Correspondingly, the constraint that the subtypes must be concrete stems from the fact that nothing was mapped from Wright to abstract component or port types.

The possibilities for specifying structure and topology are constrained.

Definition 5-52 For each Part definition *d* of a subtype of ADLSYSTEM or ADLREPRESENTATION, the set of possible part types of *d* must include only concrete subtypes of either ADLCOMPONENT or ADLCONNECTOR. ■

Definition 5-53 Subtypes of ADLSYSTEM and ADLREPRESENTATION must not have port definitions. ■

Definition 5-54 For each Part definition *d* of a subtype of ADLCOMPONENT or ADLCONNECTOR, the set of possible part types of *d* must include only subtypes of ADLREPRESENTATION. ■

Definition 5-55 For each Port definition *d* of a subtype of ADLCOMPONENT, *d.types* must include only concrete subtypes of ADLPORT. ■

Definition 5-56 For each Port definition *d* of a subtype of ADLCONNECTOR, *d.types* must include only concrete subtypes of ADLROLE. ■

Further, it must be required that in all part and port definitions, the set of possible part or port types, respectively, is of size one. Otherwise, the mapping from the type to the ADLs would be ambiguous.

Definition 5-57 For each Part definition *d*, *d.types* must be of size one. ■

Definition 5-58 For each Port definition *d*, *d.types* must be of size one. ■

As the two above definitions guarantee that the set of possible types in part and port definitions contains a single type, the type will be for simplicity referred by the name *type* (compare with Table 3).

Finally, as each component and connector in Wright may define at most one representation, the number of part definitions must be constrained for these types.

Definition 5-59 For each Component type *t*, *t.parts* must be of size zero or one. ■

Definition 5-60 For each Connector type *t*, *t.parts* must be of size zero or one. ■

5.4.4 Mapping the Ontology to Wright

This section presents how a configuration model can be mapped into a Wright system or style.

Component Types

Subtypes of ADLSYSTEM are mapped into Wright systems using the function $ADLSystem_{OW}$.

Definition 5-61 Component type *t* in Configuration model *M* such that *t.type* is a subtype of ADLSYSTEM is mapped into System $ADLSystem_{OW}(t, M)$. The function $ADLSystem_{OW}$ is defined as follows:

$ADLSystem_{OW}(t, M)$:= System *s*, where
- $s.name := t.name$
- $s.comp\_types := ADLComponent_{OW,\ type}(t.components)$
- $s.conn\_types := ADLConnector_{OW,\ type}(t.\ components)$
- $s.components := ADLComponent_{OW,\ def}(t.parts)$
- $s.connectors := ADLConnector_{OW,\ def}(t.parts)$
- $s.attachments := connections_{OW}(M, t)$. ■

The functions referred to in the definition are all defined later in this section. Their semantics are implied by their names: e.g. $ADLComponent_{OW,\ type}$ maps subtypes of ADLCOMPONENT into componen types in Wright, and $ADLComponent_{OW,\ def}$ maps part definitions including subtypes of ADLCOMPONENT as part type in componen instances.

Component types that are subtypes of ADLCOMPONENT are mapped into component types in Wright. Correspondingly, subtypes of ADLCONNECTOR are mapped into connector types. However, as the mappings for the two subtypes are similar, only the

mapping for subtypes of ADLCOMPONENT is given explicitly. Below, the function $ADLComponent_{OW,\ type}$ carries out the mapping.

Definition 5-62 Component type t such that $c.type$ is a subtype of ADLCOMPONENT is mapped into Component type $ADLComponent_{OW,\ type}(t)$. The function $ADLComponent_{OW,\ type}$ is defined as follows:

$ADLComponent_{OW,\ type}(t)$:= Component type c, where

$c.name$:= $t.name$

$c.ports$:= $ADLPort_{OW,\ def}(t.ports)$

$c.rep$:= $ADLRepresentation_{OW}(t.parts)$. ■

In the above definition, functions $ADLPort_{OW,\ def}$ and $ADLepresentation_{OW}$ are used to map the port and part definitions, respectively, of the component type into ports and represenation of the resulting Wright component type.

Part Definitions

Part definitions where the part type is a subtype of ADLCOMPONENT are mapped into component instances using the function $ADLComponent_{OW,\ def}$ defined below.

Definition 5-63 Part definition p such that $p.type$ is a subtype of ADLCOMPONENT is mapped into Component instance $ADLComponent_{OW,\ def}(p)$. The function $ADLComponent_{OW,\ def}$ is defined as follows:

$ADLComponent_{OW,\ def}(p)$:= Component instance c, where

$c.name$:= $p.name$

$c.type$:= $ADLComponent_{OW,\ type}(p.type)$

$c.cardinality$:= $p.cardinality$. ■

Similarly, part definitions where the part type is a subtype of ADLCONNECTOR are mapped into connectors. The definition is omitted.

Further, part definitions where the part type is a subtype of ADLREPRESENTATION are mapped to representations. The mapping is similar to the mapping of subtypes of ADLSYSTEM into Wright systems, and is defined formally in terms of the function $ADLRepresentation_{OW,\ def}$.

Definition 5-64 Part definition p in Configuration model M such that $p.type$ is a subtype of ADLREPRESENTATION is mapped into Representation $ADLRepresentation_{OW,\ def}(s, M)$. The function $ADLRepresentation_{OW,\ def}$ is defined as follows:

$ADLRepresentation_{OW}(p, M)$:= Representation r, where

$r.name$:= $p.name$

$r.comp\_types$:= $ADLComponent_{OW, type}(p.type.components)$

$t.conn\_types$:= $ADLConnector_{OW, type}(p.type.components)$

$r.components$:= $ADLComponent_{OW, def}(p.type.parts)$

$r.connectors$:= $ADLConnector_{OW, def}(p.type.parts)$

$r.attachments$:= $connections_{OW}(M, p)$. ■

Port Definitions

Port definitions where the port type is a subtype of ADLPORT or ADLROLE are mapped into ports and roles in Wright, respectively. The mapping to ports is explicated in the following definition by the function $ADLPort_{OW, def}$.

Definition 5-65 Port definition p such that $p.type$ is a subtype of ADLPORT is mapped into Port $ADLPort_{OW, def}(p)$. The function $ADLPort_{OW, def}$ is defined as follows:

$ADLPort_{OW, def}(p)$:= Port r, where

$r.name$:= $p.name$. ■

Constraints

Those constraints that imply a connection between two individuals, of which one corresponds to a port and the other to a role in Wright, are mapped into attachments. The function $constraint_{OA}$ defined in Definition 5-36 is applicable to Wright as such. Therefore, the definition of the function is not repeated here.

Similarly, the mapping for other type of constraints is similar to what was seen in Armani in Definition 5-36.

Configuration Models without Subtype of ADLSystem

Configuration models that contain no subtypes of ADLSYSTEM are mapped into styles. Formally, the mapping is carried out by the function $model_{OA}$.

Definition 5-66 Configuration model m such that m contains no subtypes of ADLSYSTEM is mapped into Style $model_{OA}(m)$. The function $model_{OA}$ is defined as follows:

$model_{OA}(m)$:= Style s, where

$s.components$:= $ADLComponent_{OW, type}(m.components)$

$s.connectors$:= $ADLConnector_{OW, type}(m.components)$

s.constraints := $constraint_{OA}$(*m,* GLOBAL*)*. ■

The definition is similar to Definition 5-37, where similar configuration models were mapped into Armani styles. The observation made in conjunction with the above-mentioned definition about the use of the identifier GLOBAL when applying the function $constraint_{OA}$ applies here as well.

5.5 Mapping between Koala and the Ontology

This section contains three subsections. In the first subsection, a mapping is defined from Koala to the configuration ontology. Ontological constraints are defined in the second subsection, and a mapping from the ontology to Koala in the third.

5.5.1 Mapping Koala to the Ontology

Overview of the Mapping

Similarly as above for Armani and Wright, an overview of the mapping is given before proceeding into the detailed descriptions of how individual language elements are mapped.

First, Koala components are mapped into components in the configuration ontology: in more detail, component types are mapped into component types, and components defined within compound components are mapped into part definitions.

Second, interfaces are mapped into ports: interface types are mapped into port types, and interfaces declared in components are mapped into port definitions.

As can be recalled, Koala configurations are component types that have no interfaces on their boundary. Therefore, the natural way to handle Koala configurations is to map into them into a certain kind of component type in the ontology.

Finally, modules are mapped into a special type of component. The mapping is motivated by the fact that component of the configuration ontology is a concept that can capture many relevant aspects of modules in Koala.

The order in which the mappings are presented is different for Koala than for Armani and Wright. This is due to the fact that the component and interface types are distinct from configurations using them. Therefore, component and interface types are given a mapping first, and a mapping for configurations after these.

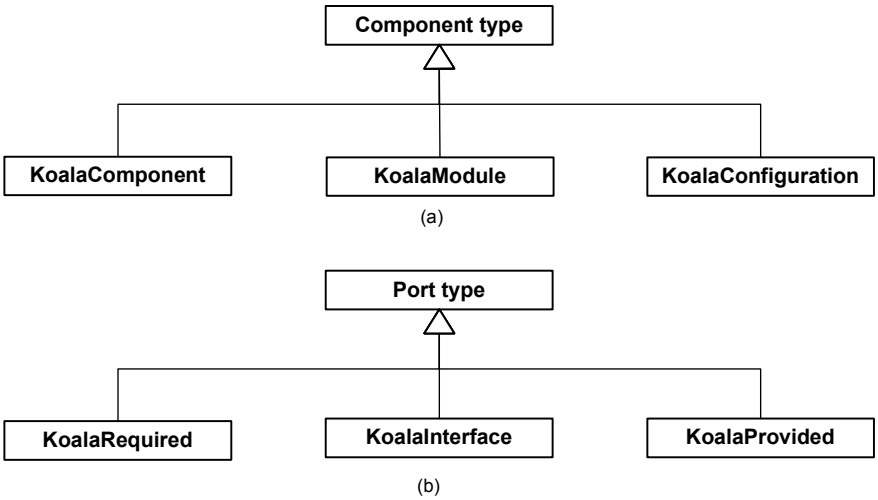


Figure 15 Component and port types used in mapping Koala. (a) The taxonomy of component types. (b) The taxonomy of port types.

General Definitions

As for Armani and Wright, special component and port types are defined to serve as supertypes for types corresponding to different entities in Koala. Figure 15 (a) depicts the taxonomy for component types, and Figure 15 (b) for port types.

Figure 15 (a) demonstrates that three subtypes are derived from Component type. First, KOALACOMPONENT is will serve as the counterpart for ordinary Koala components, i.e., not configurations; these are handled with KOALACONFIGURATION. Finally, KOALAMODULE is the type that will be the supertype for component types corresponding to modules in Koala.

Especially worth noticing in Figure 15 (b) is that for interfaces, three different types are defined: KOALAINTERFACE serves as the supertype for types corresponding to interface types of Koala. KOALAREQUIRED and KOALAPROVIDED, in turn, are used to distinguish required and provided versions of each KOALAINTERFACE from each other.

Further, the following compatibility definitions are made to enable modelling bindings, i.e., to allow port individuals corresponding to required and provided interfaces being bound, and to disallow port individuals corresponding to a pair of required or provided interfaces from being bound.

Definition 5-67 KOALAREQUIRED is compatible with KOALAPROVIDED. KOALAPROVIDED is compatible with KOALAREQUIRED.

Component Types

Component types that are not configurations are mapped into dependent component types. Formally, the mapping is carried out by the function $componentType_{KO}$ defined below.

Definition 5-68 Component type *c* is mapped into Component type $componentType_{KO}(c)$. The function $componentType_{KO}$ is defined as follows:

$componentType_{KO}(c)$:= Component type *t*, where

t.name	:= *c.name*
t.abstraction	:= CONCRETE
t.dependency	:= DEPENDENT
t.parts	:= $containedComponent_{KO}(c.components)$
t.ports	:= $interfaceDef_{KO}(c.interfaces)$
t.types	:= { KOALACOMPONENT }

Further, for each Binding *b* in *c.bindings*, and each Interface *i* such that *i* = *b.required* and *i.direction* = REQUIRED, $interfaceDef_{KO}(i)$ is modified as follows:

constraints := $interfaceDef_{KO}(i).constraints \cup \{ binding_{KO}(b) \}$.

M is modified as follows:

components := $M.components \cup componentType_{KO}(c)$

In the above definition, the components contained in *c* type are mapped into parts of *t* by using the function $containedComponent_{KO}$ defined below. Similarly, the interfaces defined by *c* are mapped into port definitions of *t* by the function $interfaceDef_{KO}$, likewise defined below.

Example. Figure 16 (a) depicts that `CClient`, `CServer`, and `CSub` have been mapped into subtypes of KOALACOMPONENT. ■

Interface Types

Each interface type in Koala is mapped into three port types. First, an abstract subtype of KOALAINTERFACE is created. Then, two additional types are created by multiple inheritance from the abstract subtype of KOALAINTERFACE and KOALAREQUIRED. The latter two types are concrete. Formally, the mapping is carried out by the functions $interfaceType_{KO}$, $interfaceReq_{KO}$, and $interfacePro_{KO}$.

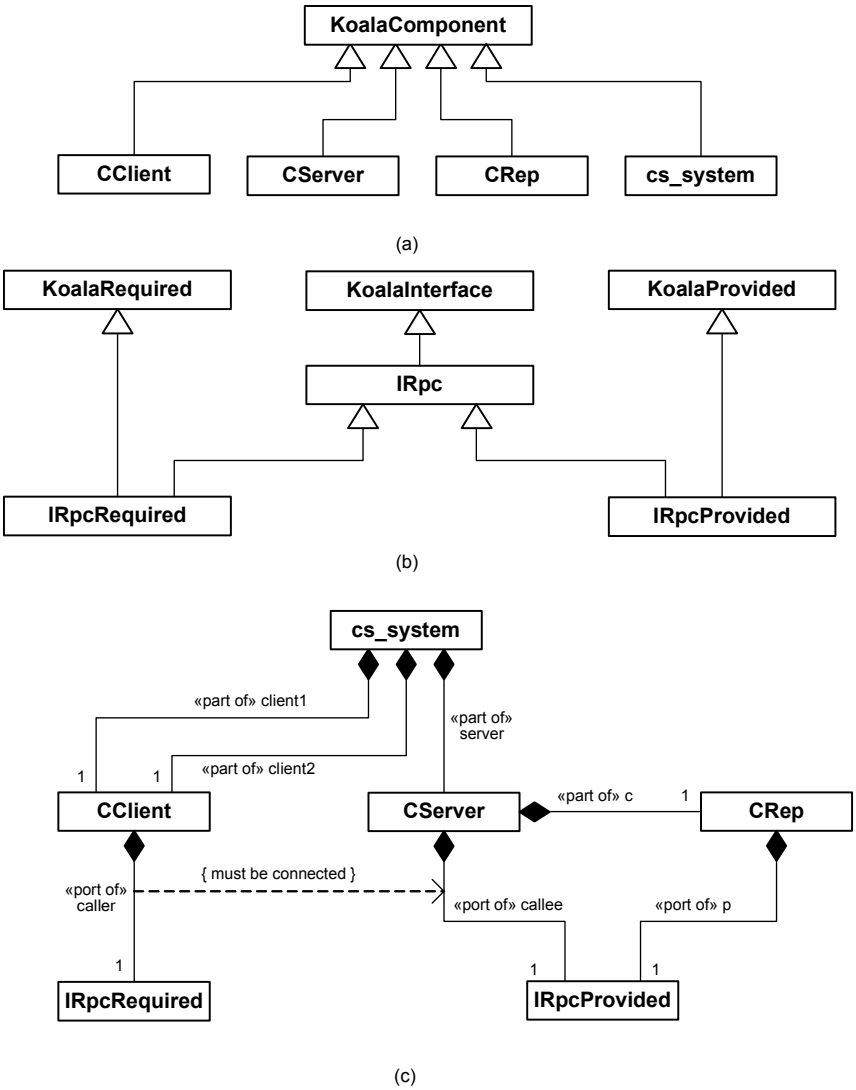


Figure 16 Mapping the client-server system from Koala to the configuration ontology. (a) The taxonomy of component types. (b) The taxonomy of port types. (c) The part and port structure in the configuration model.

Definition 5-69 Interface type *t* is mapped into Port type $interfaceType_{KO}(t)$, Port type $interfaceReq_{KO}(t)$, and Port type $interfacePro_{KO}(t)$. The function $interfaceType_{KO}$ is defined as follows:

$interfaceType_{KO}(t)$:= Port type *p*, where

p.name := *p.name*

p.abstraction := ABSTRACT

p.types := { KOALAINTERFACE }

The function $interfaceReq_{KO}$ is defined as follows:

$interfaceReq_{KO}(t)$:= $interfaceType_{KO}(t)$ modified with

name := $interfaceType_{KO}(t).name$ || 'Required'

abstraction := CONCRETE

types := { KOALAINTERFACE, KOALAREQUIRED }

The function $interfacePro_{KO}$ is defined as follows:

$interfacePro_{KO}(t)$:= $interfaceType_{KO}(t)$ modified with

name := $interfaceType_{KO}(t).name$ || 'Provided'

abstraction := CONCRETE

types := { KOALAINTERFACE, KOALAPROVIDED }

M is modified as follows:

M.ports := $M.ports \cup interfaceType_{KO}(t) \cup interfaceReq_{KO}(t) \cup interfacePro_{KO}(t)$. ■

Above, the operator '||' is the catenation of strings. E.g. 'IRpc' || 'Required' evaluates to 'IRpcRequired'.

Example. Figure 16 (a) illustrates the types created based on the `IRpc` interface type in the sample system. `IRpc` is an abstract type derived from `KoalaInterface`. `IRpcRequired` inherits from `KoalaRequired` and `IRpc`; `IRpcProvided`, in turn, inherits from `KoalaProvided` and `IRpc`.

Component Definitions

The obvious mapping for definitions of contained components is to represent them as part definitions in the compound component types. Formally, this is defined in terms of the function $containedComponent_{KO}$.

Definition 5-70 Component definition *c* is mapped into Part definition $containedComponent_{KO}(c)$:

$containedComponent_{KO}(c)$:= Part definition *d*, where

d.name := *c.name*

d.type := *c.type*

d.cardinality := 1. ■

Example. Figure 16 (c) depicts part definitions resulting from mapping contained components. Type `CClient` is a part of `cs_system` by the names `client1` and `client2`; `CServer` is part of `cs_system` by the name `server`; finally, `CRep` is a part of `Cserver` by the name `c`. ■

Interface Definitions

Interfaces defined by components are mapped into port definitions by using the function $interfaceDef_{KO}$.

Definition 5-71 Interface definition d is mapped into Port definition $interfaceDef_{KO}(d)$. The function $interfaceDef_{KO}$ is defined as follows:

$interfaceDef_{KO}(d)$:= Port definition p, where

$p.name$:= $d.name$

$p.type$:= $interfaceReq_{KO}(p)$, if $d.direction$ = REQUIRED; otherwise $interfacePro_{KO}(p)$

$p.cardinality$:= 1, if $d.direction$ = REQUIRED; otherwise ∞. ■

It can be seen from the above definition that the mappings for required and provided interfaces differ from each other in two respects. First, the interface type is naturally different, as was explained above when defining the mapping for interface types. Second, the cardinality is different for definitions corresponding to required and provided interfaces. This difference is due to the binding rules of Koala. A required interface must be bound to a single provided interface, but a provided interface can be bound to any number of required interfaces.

Example. Figure 16 (c) depicts port definitions resulting from mapping required and provided interfaces in component types: type `IRpcRequired` is related through a port definition to component type `CClient`; and type `IRpcProvided` is related to `CServer` and `CSub`.

Bindings

As attachments in Armani and Koala, bindings between required and provided interfaces are mapped into constraints. These constraints are then placed in the port definitions.

It should be noted that only bindings between independent components are given a mapping. This is due to the fact that binding between independent components and between a contained and a compound component are semantically different, although the same concept, namely binding, is used for modelling both types of bindings. The difference between the two forms of bindings is reflected e.g. in the fact that bindings between independent interfaces are always between a provided and a required interface,

whereas bindings between an interface in a contained and in a compound component is always between a pair of required or provided interfaces.

Formally, bindings are mapped by the function $binding_{KO}$ defined in the definition below.

Definition 5-72 Binding *b* is mapped into Constraint $binding_{KO}(b)$. The function $binding_{KO}$ is defined as follows:

$binding_{KO}(b)$:= Constraint *c*, where

c.expression := $interfaceDef_{KO}(b.required)$ and $interfaceDef_{KO}(b.provided)$ must be connected. ■

Modules

Module is a construct for binding a required interface with a number of provided interfaces. They are mapped into special type of component, namely KOALAMODULE, using function $module_{KO}$.

Definition 5-73 Module *m* is mapped into Component type $module_{KO}(m)$. The function $module_{KO}$ is defined as follows:

$module_{KO}(m)$:= Component type *t*, where

t.abstraction := CONCRETE

t.dependency := DEPENDENT

t.ports := $interfaceDef_{KO}(m.required \cup m.provided)$

t.types := { KOALAMODULE }

M is modified as follows:

M.components := $M.components \cup module_{KO}(m)$. ■

Configurations

Configurations in Koala are component types that define no interfaces themselves. Thereby, it is natural to map configurations into independent subtypes of KOALACONFIGURATION. The mapping is carried out by the function $configuration_{KO}$ that is defined below.

Definition 5-74 Configuration *c* is mapped into Configuration model *M* and Component type $configuration_{KO}(c)$. Initially, *M* contains the knowledge about the component and interface types resulting from the above definitions for mapping component and

interface types, and the component and port types from Figure 15 and the compatibility definitions from Definition 5-67.

configuration$_{KO}$*(c)* := Component type *t*, where

t.name := *c.name*

t.abstraction := CONCRETE

t.dependency := INDEPENDENT

t.parts := *containedComponent*$_{KO}$*(c.components)*

t.types := { KOALACONFIGURATION }

Further, for each Binding *b* in *c.bindings*, and each Interface *i* such that *i* = *b.required* and *i.direction* = REQUIRED, *interfaceDef*$_{KO}$*(i)* is modified as follows:

constraints := *interfaceDef*$_{KO}$*(i).constraints* ∪ { *binding*$_{KO}$*(b)* }.

M is modified as follows:

components := *M.components* ∪ *configuration*$_{KO}$*(c)*. ■

Example. Figure 16 depicts the client-server system mapped from Koala to the configuration ontology. Figure 16 (a) illustrates the taxonomy of component types, and Figure 16 (b) that of port types. Figure 16 (c) presents the part and port definitions of component types graphically.

As can be seen in Figure 16 (a), the configuration type `cs_system` has been mapped into a subtype of KOALACONFIGURATION. ■

Example. In Figure 16 (c), it can be seen how the contained components are mapped into part definitions. E.g. `CClient` is the part type in two part definitions, named `client1` and `client2`. ■

Table 11summarises the above mappings. As can be seen, functions, the constituent parts of interfaces, and bindings between them were the only concept that could not be found a mapping.

5.5.2 Ontological Constraints

Similarly as in Armani and Wright, there is nothing in Koala corresponding to resources and functions.

Definition 5-75 There are no resource and no function types in the configuration model. ■

Table 11 Summary of mapping Koala to the configuration ontology

Koala		Configuration ontology
Entity	Properties	
Configuration		Independent subtype of KOALACONFIGURATION, configuration model
	Components	Part definitions of the independent subtype
	Binding	Connection constraints in port definitions
Component type		Concrete and dependent subtype of KOALACOMPONENT
	Interfaces	Port definition of the subtype of KOALACOMPONENT
	Components	Part definition of the subtype of KOALACOMPONENT
	Bindings	Connection constraint in a port definition
Interface type		Abstract subtype of KOALAINTERFACE, two concrete versions of the type corresponding to required and provided interfaces
	Functions	**No mapping**
Interface definition		Port definition of a subtype of KOALACOMPONENT
	Type	The abstract part of the port type in the port definition
	Direction	The concrete part of the port type in the port definition
	Necessity	Different values for cardinality in the port definition
Component (cont.)		Part definition in a subtype of KOALACOMPONENT
	Type	Part type of the part definition
Function binding		**No mapping**
Module		Concrete and dependent subtype of KOALAMODULE

The above constraint is set is due the fact that as in Armani and Wright, there is no obvious corresponding concept in Koala for functions and resources.

The taxonomy of types must be restricted.

Definition 5-76 There exists exactly one independent subtype of KOALACONFIGURATION. This type may not define ports. ■

The latter constraint in the above definition is due to the restriction that Koala configurations have no interfaces on their boundary.

Subtypes of KOALACOMPONENT may not be independent, as configurations are the only independent components in Koala.

Definition 5-77 All the subtypes of KOALACOMPONENT are dependent types. ■

Further, all the component types must be concrete subtypes of KOALACOMPONENT or KOALACONFIGURATION, or KOALAMODULE.

Definition 5-78 Each component type must be a direct and concrete subtype of KOALACOMPONENT, KOALACONFIGURATION, or KOALAMODULE, except for this type themselves. ■

Definition 5-79 Each abstract port type must be a direct subtype of KOALAINTERFACE, and may not define other subtypes beyond KOALAINTERFACE.

Definition 5-80 Each concrete port type, except for KOALAINTERFACE, KOALAREQUIRED, and KOALAPROVIDED, must be a direct subtype of KOALAINTERFACE, and either KOALAREQUIRED or KOALAPROVIDED. ■

Components in Koala can contain both other component and interfaces. In terms of the ontology, component types can have both part and port definitions. However, the set of possible part or port types must be restricted, along with the cardinalities.

Definition 5-81 For each Part definition *d*, *d.types* may contain only concrete component types and must be of size one, and *d.cardinality* must be exactly one.

Definition 5-82 For each Port definition *d*, *d.types* may contain only concrete port types and must be of size one.

As the two above definitions guarantee that the set of possible types in part and port definitions contains a single type, the type will be for simplicity referred by the name *type* (compare with Table 3). The next definition utilises this:

Definition 5-83 For each Port definition *d* such the cardinality of d must be:

'0..1' or '1', if *d.type* is a subtype of KOALAREQUIRED
'0 or infinite' or 'infinite', if *d.type* is a subtype of KOALAPROVIDED. ■

Above, the first possible cardinalities ('0..1', '0 or infinite') correspond to optional interface definitions, and the latter to mandatory (regular) interface definitions.

According to the binding rules of Koala, a non-optional required interface must be connected to a provided interface:

Definition 5-84 For each Port definition *d* such *d.type* is a subtype of KOALAREQUIRED exactly one port individual corresponding to *d* must be connected, if there are port individuals corresponding to the definition.

In the above definition, it is stated that at least one port individual corresponding to a port definition must be connected, if there are port individuals. The situation that there are no port individuals is possible, if the interface is optional and not present.

Definition 5-85 There may be no constraints in the configuration model except for those stating that certain ports must be connected.

The constraints in the above definitions capture the binding in Koala configurations and components. Other constraints are not meaningful, as there is nothing in Koala that would correspond to them.

5.5.3 Mapping the Ontology to Koala

Component Types

Subtypes of KOALACONFIGURATION are mapped into Koala configurations using the function $KoalaConfiguration_{OK}$.

Definition 5-86 Component type *t* in Configuration model *m* such that *t* is a subtype of KOALACONFIGURATION is mapped into Configuration $KoalaConfiguration_{OK}(t, m)$. The function $KoalaConfiguration_{OK}$ is defined as follows:

$KoalaConfiguration_{OK}(t, m)$:= Component type *c*, where

c.name := *t.name*

c.components := $KoalaComponent_{OK, def}(t.parts)$

c.bindings := $connections_{OK}(m, t)$. ■

The functions $KoalaComponent_{OK, def}$ is used for mapping the part definitions of *t* into contained components of *c*. Further, $connections_{OK}$ is used for mapping the connections defined between the ports of *t* into bindings in *c*.

The function $KoalaComponent_{OK, type}$ maps dependent component types in configuration models into component types in Koala.

Definition 5-87 Component type *t* in Configuration model *m* such that *t* is a subtype of KOALACOMPONENT is mapped into Component type $KoalaComponent_{OK, type}(t)$. The function $KoalaComponent_{OK, type}(t)$ is defined as follows:

$KoalaComponent_{OK, type}(t)$:= Component type *c*, where

c.name := *t.name*

c.interfaces := $KoalaInterface_{OK, def}(t.ports)$

c.components := $KoalaComponent_{OK,\ def}(t.parts)$

c.bindings := $connections_{OK}(m, t)$. ■

The above mappings for subtypes of KOALACONFIGURATION and KOALACOMPONENT resemble each other closely. The difference is subtypes of KOALACONFIGURATION have no port definitions, as they correspond to Koala configurations; these port definitions are mapped into interfaces defined by components using the function $KoalaInterface_{OK,\ def}$ specified below.

Finally, subtypes of KOALAMODULE are mapped into modules in Koala. The mapping is implemented by function $KoalaModule_{OK}$, and defined in the definition below.

Definition 5-88 Component type *t* such that *t* is a subtype of KOALAMODULE is mapped into Module $KoalaModule_{OK}(t)$. The function $KoalaModule_{OK}$ is defined as follows:

$KoalaModule_{OK}(t)$:= Module *m*, where

m.interfaces := $KoalaInterface_{OK,\ def}(t.ports)$. ■

Part Definitions

Part definitions correspond to contained components. They are mapped into Koala as follows using function $KoalaComponent_{OK,\ def}$.

Definition 5-89 Part definition *p* is mapped into Component definition $KoalaComponent_{OK,\ def}(p)$. The function $KoalaComponent_{OK,\ def}$ is defined as follows:

$KoalaComponent_{OK,\ def}(p)$:= Component instance *c*, where

c.name := *p.name*

c.type := $KoalaComponent_{OK,\ type}(p.type)$. ■

Port Types

Abstract port types in the configuration ontology are mapped into interface types in Koala. Formally, the mapping is carried out as follows by the function $KoalaInterface_{OK,\ type}$:

Definition 5-90 Abstract Port type *t* such that t is a subtype of KOALAINTERFACE is mapped into Interface type $KoalaInterface_{OK,\ type}(t)$. The function $KoalaInterface_{OK,\ type}$ is defined as follows:

$KoalaInterface_{OK,\ type}(p)$:= Interface type *t*, where

t.name := *t.name*. ■

It should be noticed that the above definition concerns only abstract port types; the other two port types created for each interface type in Koala (see Figure 16) become thus correctly ignored.

Port Definitions

Port definitions in configuration models are mapped into interface definitions in Koala component types. The mapping is carried out by the function $KoalaInterface_{OK,\ def}$.

Definition 5-91 Port definition *p* is mapped into Interface definiton $KoalaInterface_{OK,\ def}(p)$. The function $KoalaInterface_{OK,\ def}$ is defined as follows:

$portDef_{OK}(p)$:= Interface definition *r*, where

r.name := *p.name*

r.type := $KoalaInterface_{OK,\ type}(p.type)$

r.direction := REQUIRED, if KOALAREQUIRED is a supertype of *p.type*; otherwise PROVIDED. ■

Constraints

Only constraints implying that two port individuals must be connected can be meaningfully mapped into Koala: they are mapped into bindings by the function $connections_{OK}$.

Definition 5-92 The function $connections_{OK}(m,\ t)$ takes a Configuration model m and Componen type *t* as arguments and returns a set of Bindings *s*: *s* := { *b* | *b.first* = $KoalaInterface_{OK,\ def}(p)$ and *b.second* = $KoalaInterface_{OK,\ def}(r)$, *p* and *r* are Port definitions in *m* such that *p* and *r* are port definitions in parts of *t*, and m implies that *p* and *r* must be connected }. ■

Above, only the connection constraints corresponding to bindings within a single component type are returned. This is correct, as component types are mapped one at a time, and a configuration model can contain constraints corresponding to bindings located in multiple component types.

6 Discussion

This chapter discusses the construction presented in the previous chapter, and explicates answers to the research question stated in the introduction.

The first research question, which concerned the concepts of the ADLs and their comparison to those of the configuration ontology, has already been answered in Section '3.2 Architecture Description Languages', and Section '4 Comparison of Concepts of the ADLs with the Configuration Ontology'; see, e.g., Table 1 for details.

The first section summarises the mappings that could be found from the ADL concepts to the configuration ontology. Simultaneously, the section explicates and reiterates the answer to the second research question concerning the mapping between the ADLs and the configuration ontology. In short, the answer is that there is a partial mapping; full details of the mapping were given in Chapter '5 Synthesis'.

The second section, in turn, discusses the concepts of the ADLs that were not found a satisfactory mapping. Further, extensions to the ontology that would enable capturing the lacking concepts are suggested. The third research question is also answered in the subsection.

The concepts of the ontology that were not used for representing any concepts in the ADLs are discussed in the third section.

In the fourth section, the reliability of the results achieved in the thesis is evaluated. General discussion follows in the fifth section.

6.1 Successfully Mapped Concepts

6.1.1 Configurations, Systems

Armani and Wright systems and Koala configurations are mapped into specific kind of component types in the configuration ontology. Another alternative would have been to map these concepts into sets of independent component types; the component types would have corresponded to the components and connectors defined in the system or configuration. This approach was, however, not adopted because systems in Armani and Wright have identities, and in Armani properties as well. Therefore, the approach would have resulted in losing this class of information. In Koala, in turn, a configuration is

explicitly defined to be a kind of component; therefore, not mapping configurations into component types would have been hard to argue for.

6.1.2 Components and Connectors

Armani and Wright components and connectors, and Koala components are mapped into component types, and part definitions.

In more detail, Armani components were mapped into component types and part definitions. As was argued in conjunction with the associated definition (Definition 5-3), both the component type and the part definition are needed in the ontology to capture both information about the structure of a component and the fact that a component is always part of something.

Of course, the number of component types resulting from mapping an Armani design would be smaller in some cases, if all components declaring a single type were mapped into a single component type and multiple part definitions using the same type. However, this approach would have other drawbacks: most significantly, components declaring no types, a single type, and multiple types would have had to been handled differently: creating a new component type is in any case mandatory for design elements declaring no types or multiple types, as every individual in the configuration ontology must be directly of exactly one type. Uniform handling of all components is considered more important than minimizing the number of resulting component types.

In Wright and Koala type information about components is separated from the information about instances. Therefore, the above problem of an extensive number of component types is not a problem when mapping from these languages, as component types are mapped into component types, and instantiations into part definitions or constraints requiring that the components types get instantiated in configurations.

Of course, what was said about components above applies mostly likewise to connectors. However, mapping connectors of Armani and Wright to component types is not in perfect accordance with the semantics of component in the configuration ontology: connectors are not typically distinguishable entities in software systems. This cannot, however, cannot be avoided, as there are no concepts in the configuration ontology besides component to which connectors could be reasonably mapped.

6.1.3 Ports, Roles, and Interfaces

Ports and roles in Armani and Wright, and interfaces in Koala are mapped into port types and definitions in the configuration ontology.

Much of the discussion presented for components and connectors in the above section could be repeated as such for ports and roles in Armani and Wright. That is, the mapping for ports and roles is mostly obvious in the sense that there seem to be no viable alternative mappings. However, the number of resulting port types in the configuration model is somewhat excessive. In this case, both Armani and Wright have this condition: although Wright distinguishes between component and connector types and instances, there is no corresponding mechanism for ports and roles, except the interface type construct. Consequently, each port and role defined in a component in Armani and Component in Wright is mapped into a port type and a port definition.

As ports and roles in Wright define no properties beyond name and process, of which the process is not mapped, it could be argued that all the port definitions corresponding to ports in Wright could use the same type. I.e., no individual types would be defined for Wright ports. However, this approach is not adopted, as it would have de-emphasised the fact that not all ports can be attached to any role in Wright; with the selected approach, the knowledge about which ports can be substituted for which roles can be encoded as compatibility definitions of the corresponding port types.

Koala does not suffer from the excessive number of port types, as interface types and instances of them in components are separated from each other.

6.1.4 Topology: Attachments and Bindings

Attachments between ports and roles in Armani and Wright are mapped into connection constraints in port definitions in the configuration ontology corresponding to ports in the two ADLs. Bindings between independent interfaces in Koala are mapped into connection constraints as well.

Mapping attachments in Armani to connection constraints in the configuration ontology brings forward the issue that instances of Armani were mapped into types in the configuration ontology. As argued in Section '5.3.1 Mapping Basic Concepts of Armani to the Ontology', this is the simplest way to map instances into the configuration ontology. However, in a sense the natural counterpart for Armani instances and the

relations between them in the ontology would be individuals and the relations between them: i.e., components in Armani correspond to component individuals in the ontology; compositional structure corresponds to HAS PART-relation; and attachments to CONNECTED TO-relation. Therefore, when mapping Armani systems into configuration models, component types, part definitions and connection constraints are included in the model to enforce that the configurations based on the model contain individuals that are related to each other in the way described above: e.g. a connection constraint essentially guarantees that in a complete configuration, the port individuals involved in the constraints are in the CONNECTED TO-relation with each other.

6.1.5 Taxonomy

Of the ADLs, Armani is the one to define taxonomy between types. There, taxonomy is defined both for design element, i.e., component, connector, port, and role, types, and styles.

Design element types of Armani are mapped into abstract component and port types. The taxonomy between design element types is mapped into taxonomy between the resulting, abstract component and connector types. Further, the fact that a design element declares a type is captured by making the concrete type of the design element a subtype of abstract types corresponding to the types declared by the desing element.

Mapping design element types into abstract types can be questioned, as mapping design element types into concrete types would as well have been possible. However, there are some reasons speaking against such an approach, as discussed when in conjunction with the definition for mapping component types from Armani to the ontology (Definition 5-10). First, following this approach would have resulted in ambiguities when mapping from ontology to Armani: if all the types were concrete, there would be no way of distinguishing types corresponding to design elements and design elements types. Second, there would be no use in making these types concrete, as all the design elements are created a concrete type of their own when mapping them to the ontology. That a new component type is created for each design element when mapping in the ontology was justified by uniformity above in Section '6.1.2 Components and Connectors'.

6.1.6 Compositional Structure

Systems and Representations

The compositional structure of systems and representations is mapped into part definitions in the configuration ontology. The mapping is intuitive and correct, as part definitions are the mechanism for modelling compositional structure in the ontology.

However, mapping connectors into part definitions of component types corresponding to systems is not in perfect accordance with the semantics of part definition in the ontology. This disaccordance could have been avoided by making connectors independent components instead of parts of other components. Albeit viable, this approach was not adopted, as it was considered more important that the resulting configuration models reflect the structure of ADL systems than that the use of the concepts of the configuration ontology matches their intended use.

The above design decision that connectors are mapped into part definitions is related to a more general issue: when modelling software products with the configuration ontology, how closely should the guidelines set for modelling traditional products (see (Tiihonen et al. 1998)) be followed when modelling software products. Intuitively, the most of the design guidelines are likely to apply to software products as well. However, stubbornly following the guidelines without properly analysing their applicability to software systems is hardly reasonable.

Components and Connectors

Defining compositional structure for components and connectors is possible in all the studied ADLs: In Armani, components and connectors can define a number of representations; in Wright, the computation of a component and the glue of a connector can be replaced with a system; in Koala components can contain other components. Compositional structure is mapped into compositional structure of component types, manifested by part definitions. However, no satisfactory mapping for the bindings between the ports and roles in compound and containing components could be found, although these bindings exist in all the ADLs studied. This issue is discussed in more detail below, in Section '6.2.5 Bindings between Compound and Contained Components'.

Some discussion about the mapping of representations in Armani and Wright is in place. Namely, as can be recalled from the mappings (Definition 5-9 and Definition 5-43), representations were mapped into component types, and the components and connectors in the representation were mapped, in addition to types, part definitions in these component types. An alternative mapping would have been to map the components and connectors directly into parts of the component or connector containing the representation. Following this approach would have resulted in avoiding one layer in the decomposition hierarchy of the components. However, as argued in '5.3.1 Mapping Basic Concepts of Armani to the Ontology', the adopted mapping creates a symmetry between mappings for independent and representing systems, and allows mapping the names and properties of representations into the configuration ontology. Further, unlike the alternative approach, the adopted approach allows mapping multiple representations for a component or connector.

6.2 Unmapped Concepts and Potential Extensions

This section discusses the features of the ADLs that could not be captured with the configuration ontology. These features include heuristic constraints, design analyses and interface types, behaviour modelling, function biding, binding the interfaces in contained component with those in compound components, and constraint languages. Further, extensions that would allow accommodating some of these features in the ontology are outlined. This section also answers the third research question.

6.2.1 Heuristic Constraints

Heuristic constraints are a part of Armani, in which they can be defined in style definitions. The distinguishing factor between heuristic constraints and ordinary constraints, termed design constraints in Armani and just constraints in the ontology, is the fact that heuristic constraints are merely suggestions of what might be true for a Armani system following a style or a configuration corresponding to a configuration model.

Default values of attributes are a special case of heuristic constraints. On the conceptual level, default values are an important special case of heuristic constraints. In this special case, the heuristic constraint simply states that a certain attribute has a specific value. A default value is typically interpreted as to hold if not otherwise explicitly specified. This

special case demonstrates two important aspects of heuristic constraints: first, they take the same form as ordinary constraints; second, they must be given an *interpretation* that defines their semantics.

Preferences or soft constraints are known in the configuration domain as well (see, e.g. (Junker 2001)). Therefore, extending the ontology with the notion of heuristic constraints has a justification also from the configuration point of view. A simple approach to extending the configuration ontology with heuristic constraints would be to define each constraint instance to be either invariant or heuristic. However, this would not be enough: heuristic constraints should be defined semantics as well, and it is not obvious what the semantics should be.

6.2.2 Design Analyses, Interface Types

Both design analyses in Armani and interface types in Wright are characterised by the fact that they bear whatsoever no meaning as such, but only in some specific context: design analyses are expressions in the Armani Constraint Language that can be used in forming constraints, and interface types are CSP process description that can be used in place of process definitions in element type declarations.

The ontology could be extended to cover design analyses simply by defining the ontology an appropriate constraint language that includes a feature corresponding to design analyses. Basically, interface types could be incorporated to the ontology similarly: define a mechanism for modelling behaviour that includes interface types. However, as will be next discussed, extending the ontology with behaviour modelling is not as simple as defining a constraint language.

6.2.3 Modelling Behaviour

Of the studied ADLs, Wright has a strong emphasis on modelling behaviour: CSP is used as a method for describing the behaviour of different entities. Further, tools processing CSP can be used to perform analysis on the local compatibility of ports and roles, and on system-wide properties.

In a generic solution to introducing behaviour modelling in the configuration ontology, behavioural knowledge would be integrated with other kind of configuration knowledge. Wright would suggest including behaviour descriptions in components and their ports. Of course, simply adding behaviour knowledge to component and port types would not

be a complete solution to the problem: the semantics of behavioural knowledge should be defined as well. Further, there are many other approaches to behaviour modelling besides CSP: e.g. statecharts is an alternative method to model the behaviour of software (Kühn 2000). Thereby, coming up with a generic solution to behaviour modelling is would require carefully analysing the different techniques for modelling behaviour.

On the other hand, behaviour modelling methods could be used in conjunction with the configuration ontology without integrating any specific method of into the configuration ontology. E.g., the compatibility definitions between port types in a configuration model could by produced by a tool analysing the CSP processes in ports and roles in Wright. Further, a ready configuration resulting from a configuration process could be transformed back to Wright and tested for some desired properties, again using tools processing CSP. Of course, these kinds of approaches are no substitute for the generic solution described in the above paragraph: e.g., the approach suggested above requires that the resulting configurations are in most cases valid; otherwise a multitude of configurations should be generated in order to discover a valid one.

If it is not clear how to model behaviour in the ontology, it is likewise uncertain if the ontology should model behaviour at all. That Koala is an ADL without behaviour modelling in industrial use speaks against the usefulness and necessity of modelling behaviour.

6.2.4 Function Binding

Function binding is a problematic issue from the configuration ontology point of view: In the ontology, connections are defined between ports, and ports can have no compositional structure. Therefore, there seems to be no possibility to model functions of Koala interfaces, or more importantly, the possibility to connect either interfaces or functions.

As both interfaces and functions can be directly bound to each other and interfaces are composed of functions, in a sense natural approach would be to allow compositional structure for ports in a way similar to components. In this approach, port types would correspond to both interfaces and functions. Then, types corresponding to functions could be defined to be parts of types corresponding to interfaces. Formally, the Port type

interfaceType$_{KO}$*(t)* (Definition 5-69) resulting from mapping Interface type *t* would be supplemented with port definitions for the functions as specified in the following definition:

Definition 6-1 Function *f* is mapped into Port definition *function*$_{KO}$*(f)*. The function *function*$_{KO}$ is defined as follows:

function$_{KO}$*(f)* := Port definition *d*, where

p.name := *f.name*

p.types := { KOALAFUNCTION }. ■

The KOALAFUNCTION type occurring in the above definitions would be defined as a concrete port type. It is worth noticing that the individual functions are not mapped into ports with different types. This is, of course, not the only possibility. Other alternatives would be to create types based on the signatures of functions, and creating types based on the names of the functions. Both of these approaches would limit the possibilities of binding functions, which would in principle be a good thing, as all functions surely cannot be reasonably bound with each other. Unfortunately, neither of these schemes for creating different types for functions would result in the desired effect: first, intuitively, functions with the same name can be incompatible, and functions with different names compatible with each other; second, it is not necessary to bind the functions with a one-to-one matching between the parameters, but some parameter conversions and similar adjustments are possible. Thereby, both types based on names and function signatures would have to be supplemented with complex compatibility definitions. This would not be reasonable, as Koala seems to assume that the engineer has the knowledge on what functions may be bound which each other.

Of course, the semantics of connections between ports representing interfaces and functions in Koala should be defined. In Koala, a connection between two interfaces implies a connection between all the constituent functions of the interfaces; this could be expressed as an ontological constraint in the ontology.

Definition 6-2 A connection between port two individuals of KOALAINTERFACE implies connections between all the ports of the individuals of the two individuals. ■

Still another issue is the depth of the port-of hierarchy. In the case of Koala, an ontological constraint should be set to restrict that only subtypes of KOALAINTERFACE can be defined ports.

Definition 6-3 Only ports types that are subtypes of KOALAINTERFACE may define ports, and the ports defined must be of type KOALAFUNCTION. ■

All in all, introducing compositional structure for ports would seem to enable modelling the compositional structure of interfaces in Koala. However, the compositional structure of ports would result in a problem in the ontology. Namely, the compatibility of port types would not be obvious: if two types were defined to be compatible, would their subtypes be likewise compatible? In the current ontology, subtypes of compatible port types are compatible as well. The compositional structure of ports would induce problems, as two subtypes of a given type could define different sets of ports. Consequently, either the compatibility between types would not extend to subtypes, or there could not be an implication similar to that of Definition 6-2. In the first case, the current ontology would need to be changed; in the second case, the intuitive semantics of the compositional structure would not hold.

On the other hand, the possibility of using the existing mechanism of the ontology for capturing functions could be studied: after all, as a small number of concepts is one of the criteria for a good ontology (Gruber 2002), the possibility of reasonably using existing concepts should be studied before adding new ones.

It appears that the desired effects could be achieved by mapping interfaces to components and functions to ports. Formally, the mapping would be carried out as follows:

Definition 6-4 Interface type *t* is mapped into Component type $interfaceType_{KO}(t)$. The function $interfaceType_{KO}(t)$ is defined as follows:

$interfaceType_{KO}(t)$:= Component type *c*, where

c.name	:= *t.name*
c.abstraction	:= ABSTRACT
c.dependency	:= DEPENDENT
c.ports	:= $function_{KO}(t.functions)$
c.types	:= { KOALAINTERFACE }.

The function $function_{KO}$ is the same function as defined in Definition 6-1.

KOALAINTERFACE is an abstract and dependent component type that includes a Port definition *d*:

d.name	:= "Default"

d.types := { KOALAFUNCTION }

d.cardinality := 1

d.constraints :=

{

both ports in the connection have the name "Default"

and

type of both ports is a subtype of the same subtype of KOALAINTERFACE

and

other ports in the component are connected with a port with the same name

}

M is modified as follows:

components := $M.components \cup interfaceType_{KO}(t) \cup interfaceReq_{KO}(t) \cup interfacePro_{KO}(t)$. ■

Functions $interfaceReq_{KO}$ and $interfacePro_{KO}$ are defined in a similar manner as in **Definition 5-69**.

The port named DEFAULT in the above definition is used for capturing the fact that a connection between interfaces in Koala implies that the functions of the interfaces are likewise connected. In more detail, the third part of the connection constraint in the port definition guarantees this.

Definition 6-5 Interface definition *d* is mapped into Part definition $interfaceDef_{KO}(d)$. The function $interfaceDef_{KO}$ is defined as follows:

$interfaceDef_{KO}(d)$:= Part definition *p*, where

p.name := *d.name*

p.type := $interfaceReq_{KO}(p)$, if *d.direction* = REQUIRED; otherwise $interfacePro_{KO}(p)$

p.cardinality := 1, if *d.direction* = REQUIRED; otherwise INFINITE. ■

The above definition does not account for optional interfaces. For a discussion on how optionality affects cardinality, see Section '5.5.1 Mapping Koala to the Ontology'.

Compatibility definitions and constraints should be defined to guarantee the right connections between functions and interfaces. These definitions are, however, omitted for brevity.

6.2.5 Bindings between Compound and Contained Components

All the ADLs analysed in this thesis have a mechanism for relating the interfaces in entities on different levels but same branch of composition hierarchy. In Koala, the mechanism used is the same as the mechanism used for specifying bindings between interfaces in independent components. The terminology used in this section has been introduced in Section '4.1 Key Concepts and the Relations between Them'.

In Armani and Wright, bindings and connections are strictly different concepts. Conceptually speaking, there are two different relations between interfaces in the ADLs. This introduces a problem when mapping between these ADLs and the configuration ontology: the only possible relation between port individuals is the CONNECTED TO-relation. Above, attachments between interfaces were already mapped into connections between ports. Therefore, mapping bindings to same concepts would result in an inelegant construction, as it would imply using the same concepts for representing two strictly different phenomena. Further, the approach would lead to major complications in port type definitions, compatibility definitions, and port definitions. Therefore, the approach of mapping bindings to connections is abandoned out of hand.

After ruling out the approach of representing bindings as connections, there are no concepts left in the ontology that can be used to capture the semantics behind bindings. Therefore, the only alternative left for accommodating bindings in the ontology is to extend the ontology with an appropriate concept. In its most simple form, this concept would be a relation between port individuals that would specify the pairs of port individuals that correspond to each other. This relation would be called BOUND TO. In essence, the relation would be similar to the CONNECTED TO-relation. Of course, it must be possible to specify compatibilities and connection constraints with respect to the BOUND TO-relation in a similar manner as with respect to the connected-to relation.

A construct corresponding to compatibility definitions must be defined, as it would seem unreasonable to allow unrestricted possibilities for binding port individuals. Further, port definitions should be decorated with binding constraints, again in parallel with connections.

With the extensions outlined above, bindings could be mapped into the ontology in a similar manner as attachments. However, in the ADLs bindings affect the possibilities of making attachments. The most explicit example is Koala: e.g. a required interface

must be either connected to another required interface in the compound component, or to a provided interface in a peer component. The corresponding constraint in the ontology would be to that a port is not allowed to be in the BOUND TO- and CONNECTED-TO-relations simultaneously.

Based on the approach outlined above, adding a notion for bindings into the ontology would not seem to result in unacceptable damages in the elegance and clarity of the ontology. However, some specifications concerning port individuals and the two relations should be added. First, it should be specified that KOALAREQUIRED and KOALAPROVIDED are compatible with themselves with respect to the BOUND TO-relation; KOALAPROVIDED and KOALAREQUIRED would still be compatible with each other with respect to the CONNECTED TO-relation, as specified in Definition 5-67. Additionally, a constraint stating that a port of type KOALAREQUIRED must be in either CONNECTED TO- or BOUND TO-relation with a port of type KOALAPROVIDED should be added.

Besides the approach outlined above, i.e., adding a new relation between ports, an alternative worth considering would be to introduce connection types to the ontology. In more detail, connection types would be organised in a taxonomy in a manner similar to component types. Allowing attribute definitions in connection types would enhance their expressive power; whether this expressive power would have any use is another question. The natural way of relating connection types to ports would be to define the CONNECTED TO-relation between two port individuals and a connection type individual.

If connection types were added to the ontology, attachments and bindings could then be mapped connections using different connection types. Connections constraints and compatibility definitions could have the option to discriminate between different connection types. In a sense, connection types and connectors in the ADLs would be parallel concepts: both components and connection types would allow describing the connections between components. However, unlike connectors, connection types would not define interfaces. Further, connections between more than two ports would still not be possible.

Of course, the need to model both attachments and bindings is not as such a sufficient motivation for adding connection types to the ontology. In general, the possibility of specifying different kinds of connections between ports could be useful for physical products as well. An example could be computer networks. In this domain, cables connecting network nodes have characteristics that affect their use: e.g. UTP cable could

be a component type, with different categories as subtypes; further, the UTP cable type could define length as an attribute. In order to model these aspects with the current ontology, cables would have to be modelled as components. This is quite all right, as cables do adhere to the definition of component in the ontology: a distinguishable entity in a product. On the other hand, modelling network nodes and cables with different entities would have intuitive appeal.

Further, it could be argued that connection types are not needed, as connectors in the ADLs could adequately be mapped into components in the configuration ontology, and components can thus be used to model different types of interactions between components. This is a perfectly valid argument. On the other hand, it could be argued that components corresponding to connectors do not conform to the definition of component in the configuration ontology, and that the components are not parts of components corresponding to systems in the same sense as components corresponding to components. Connection types would not suffer from the above-mentioned problems.

6.2.6 Constraint Language

Both Armani and Wright specify in detail what kind of constraints are possible: Armani includes a constraint language, Armani Constraint Language, and form of constraints and the entities that can be referred in them is well specified in Wright. In the configuration ontology it is merely assumed that a constraint language is available.

The lack of constraint language in the configuration ontology leads to a number of problems. From the point of view of this thesis, the most severe are that it is impossible to say anything about the relative expressive power of the constraint languages of the ADLs and the constraints of the ontology. In detail, it is not possible to specify which constraints in the ADLs can be mapped into the configuration ontology, and vice versa.

There is no reason why the constraint language of the ontology should not be formally defined. Those of Armani and Wright could serve as starting points for formalising the language for the ontology. However, specifying the language is outside the scope of this thesis.

6.3 The Extra Expressive Power of the Ontology

Some concepts were not needed when mapping from the ADLs to the ontology. This chapter discusses these concepts and the aspects of software they could be used to capture.

6.3.1 Variability in Part and Port Definitions

In the configuration ontology, a part definition specifies the part name, a set of possible part types, and a cardinality. Port definition specifies additionally a set of connection constraints. When mapping between the ADLs and the ontology, ontological constraints were set to restrict the size of the set of possible part types to one and the cardinality to specific values. The constraints were needed, as none of the studied ADLs allows introducing any variability to the type of a design element.

A reason for the above-observed difference can be seen in how compositional structure is specified in the ADLs and in the configuration ontology: in the ADLs, compositional structure is specified in terms of the design elements that some other design element contains; in the configuration ontology, on the other hand, part definitions are holes or roles that the individuals of a number of types can fill in. In other words, structure is specified in terms of concrete components in the ADLs, and abstract parts in the configuration ontology.

The same that was said in the above paragraph about components and parts applies to ports as well, with the difference that ports specify the connection interfaces, not compositional structure, of component types.

Considering variability, multiple possible types correspond to alternative parts and ports. Optional parts and ports is a form of variability that can be achieved by varying the cardinality.

In Armani and Wright, the effect of alternative parts and ports can be achieved with styles supplemented with appropriate constraints. The constraint corresponding to alternative parts would state that exactly one component type in a set of component types is instantiated with a certain name. Optional parts, in turn, could be expressed by specifying a constraint that enumerates the components that may exist in a system of the style; in this model, other constraints would enforce the existence of mandatory parts. In conclusion, the effects that can be achieved by modifying cardinality in the

configuration ontology can be achieved with constraints in Armani and Wright. Thus, cardinalities do not provide the ontology extra expressive power compared with the two ADLs, but they do provide a more intuitive and more intelligible way of modelling.

6.3.2 Contexts

Contexts in the configuration ontology are a mechanism for restricting pieces of configuration knowledge to certain "parts" of a configuration. A context is defined to be a set of component individuals in a configuration. The ontology includes three special types of context:

- *Component type set context*, defined as a set of individuals in a configuration that are of the component types in a given set of component types.
- *Mereological context*, defined as consisting of a component individual and the set of component individuals that are its parts.
- *Topological context*, defined as consisting of a component individual and the set of component individuals that are connected through port individuals to it.

If available in the ADLs, contexts could be used to restrict the effect of constraints to certain components and connectors. Of course, the effect could be achieved simply by enumerating the components the constraint concerns. In the configuration ontology, contexts are of most utility when used with resources, the concept to be discussed next.

6.3.3 Resources

Resources are used in the ontology to model the production and use of entities, or the flow of such entities from one component individual to another.

Arguably, resources are an important aspect of software systems as well: especially in mobile devices, resources such as disk space, memory, and computing power cannot be assumed to be in ample supply. However, resources seem not to have been considered important in the software engineering domain in the sense resources are defined in the configuration ontology.

6.3.4 Functions

In the ontology, functions are used for modelling non-technical aspects of products. Examples of functions include high performance of a car, and the ability to play music of a computer. Functions are linked into technical concepts through implementation constraints. E.g. the ability to play music could be implemented by a suitable piece of software, a sound card, and a speaker.

Although the studied ADLs include no concepts corresponding to functions, they would certainly be useful in modelling software. In fact, feature models (see, e.g., (Czarnecki et al. 2000)) are used to model functional aspects of software systems among technical aspects. Thus, there is a need for function modelling in the software domain, and studying how functions of the configuration ontology could be utilised in modelling software is a promising research avenue.

6.3.5 Constraint Sets

As the name suggests, constrain sets are sets of constraints. Constraint sets allow considering the correctness of a configuration from different points of view.

6.4 Evaluation

This section presents an evaluation of the results achieved in this thesis. First, the mappings between the ADLs and the configuration ontology are evaluated for conformance to the criteria laid down in the introduction. Second, the reliability of the results achieved is evaluated.

6.4.1 Conformance of the Mappings to the Criteria

In this section, the mappings between each ADL and the configuration ontology are evaluated with respect to the criteria laid down for the mappings in Section '1.2 Research Problem and Goals'.

In general, the mappings satisfy the criteria well. First, by inspecting the mappings, it is easy to see that they are specified in a sufficient level of detail that they can be used as a specification for implementation. Second, there is no explosion in size, as each entity is mapped into at most two entities in the target domain. Third, the mappings are unambiguous, as there are no entities with more than one mapping specified.

Fourth, the mappings are mostly elegant: the mappings are predominantly simple and straightforward. However, as discussed various subsections of Section '6.2 Unmapped Concepts and Potential Extensions', there are some semantic mismatches in the mappings: e.g. the definition of connector in Armani and Wright does not correspond to the definition of component in the configuration ontology.

Fifth, the mappings are inherently not modular, as entities in the ADLs and the configuration ontology are organised in hierarchies: systems and configurations are the roots of decomposition hierarchies; types in Armani and the configuration ontology are organised in taxonomies. Thereby, mapping e.g. systems in Armani and Wright is not possible without mapping the components and connectors contained in it. On the other hand, the mappings are roughly as modular as possible: there are no other dependencies between mappings than those dictated by the hierarchies between concepts.

Finally, the mappings have been designed for being easily understandable. Of course, the reader has the ultimate right to decide this issue.

6.4.2 Reliability

Concerning the reliability of the answers to the research questions, the critical issues are: how representative of the entire class of ADLs the three exemplars are; is the assumption that ADLs are a suitable approach for modelling software product lines valid; and how representative of the class of conceptualisation of configuration knowledge the configuration ontology by Soininen et al. is.

Starting from the lattest question, it was already stated in Section '2.2 Conceptualisations of Configuration Knowledge', the configuration ontology synthesises earlier approaches to conceptualising configuration knowledge. Additionally, the ontology is close to another conceptualisation. Consequently, the ontology by Soininen et al. was found to represent well different methods for conceptualising configuration knowledge.

Thus, the focus must be shifted to the ADLs. The representativeness of the ADLs is best argued by the special characteristics of each of the three ADLs.

As discussed in Section '3.2.1 Armani', Armani is based on an earlier ADL called Acme that was designed to incorporate the semantics core of other ADLs, and in a sense serve as an ontology of ADLs.

Wright is an ADL that is often cited in the literature. In itself, Wright has a number of interesting properties. Most importantly, Wright models behaviour and does this using a formalism called CSP. Due to these reasons, Wright was considered to represent a number of important characteristics of the class of ADLs.

Whereas Armani and Wright are predominantly academic endeavours, Koala is in industrial use in a real company, namely Philips Consumer Electronics. Strictly speaking, Koala is not an ADL but a component model. The mechanisms for modelling structure in Koala are essentially those of another ADL, namely Darwin (Magee et al. 1995).

In conclusion, the three studied ADLs can be considered to be representative of the class of ADLs. First, Armani is designed to contain the structural core of other ADLs. Second, both academic and commercial aspects are well represented: Armani and Wright guarantee the former, and Koala the latter. Third, behaviour modelling and the high degree of formalism represent the variety in ADLs. Thereby, the answers found to the research questions can be considered reliable.

However, the fact that the three ADLs are representative of the entire class of ADLs is not sufficient to guarantee that modelling software product lines with ADLs would be practically feasible. In addition, the assumption that ADLs are a suitable approach for modelling software must be valid to guarantee the feasibility. The assumption is intuitively true: software architecture is arguably the principal level of designing, modelling, and managing product lines. However, there is not much evidence that ADLs would be used in companies with software product lines. In contrast, (Bosch 2000) argues that ADLs are not used for describing the architecture of software product lines, and (Smolander et al. 2002) further states that UML is often seen as a capable of modelling architecture; consequently, there is not necessarily a large demand for ADLs in companies.

6.5 General Discussion

6.5.1 Topology in the ADLs and in the Ontology

Perhaps the most remarkable single factor differentiating the disciplines studied in this thesis is the way the model topology: Armani and Wright consider connectors as first class entities for this purpose, whereas Koala and the configuration ontology include no notion corresponding to connectors.

What then is the reason for this disagreement in architectural connection? At least a partial reason for the importance of connectors in Acme and Wright can be found in assumptions underlying them and the software engineering domain in general: reusing existing components has been, and still is, a major issue in the software engineering domain. 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 (see, e.g., (Mehta et al. 2000)). Therefore, it could be argued that connectors have been introduced in ADLs as a vehicle for connecting heterogeneous components.

In Koala, the situation is different: the language operates on component repository consisting of homogenous components. Based on the published documentation of Koala (Ommering et al. 2000; Ommering 2002), all the components follow a single interaction standard, namely procedure calls. Hence, the absence of heterogeneous components can be seen as a reason for the exclusion of explicit connectors from the language.

However, binding components is not free of problems in Koala either: as stated in Section '3.2.3 Koala', all the bindings between interfaces are not between entire interfaces. In addition, modules can be used to create considerably complex interactions between interfaces; examples include function binding and switches. Comparing the role of modules in Koala with the definition of connector in Armani and Wright reveals that modules could well be called connectors: they are defined to represent interactions and mediate the communication and coordination activities among components. In fact, the most recent publication on Koala uses the word connector when referring to the different possibilities of binding interfaces (Ommering 2002). Thereby, Koala is after all not free from connectors, although connectors are not first-class entities in Koala.

The manner in which the configuration ontology models topology stems from earlier research. In (Mittal et al. 1989), intensional connection points are introduced in order to prevent arbitrary connections between components. The reason for ruling out arbitrary connections between components is to make the configuration task, i.e., finding one or more configurations satisfying a given set of requirements, well defined. Further, connections between components are important in computers and telecom switches that are both domains where product configuration has been applied.

6.5.2 Modelling Products with Embedded Software

As mentioned in the introduction, much of software is not designed to run on general purpose computing devices, but as embedded software in specialised devices. Further, software is used as a source of variability in many cases: physically identical products are made different through customising the software. Consequently, a need for modelling simultaneously both the physical product and the software embedded in it arises.

Given that configuration models can be used to describe both the physical structure of a product and the software embedded in it, additional benefits could be gained by combining the two models: in addition to modelling the physical product and the software, the models could include the dependencies between the two models. Thereby, both the physical and software aspect of the product could be configured simultaneously, and the resulting configuration is guaranteed to be valid with respect to the physical structure, the embedded software, and their interactions.

Of course, when modelling both physical and software components, distinguishing these two types of components from one other is important. The distinction can be achieved simply be creating two subtypes of Component type, namely PHYSICAL COMPONENT and SOFTWARE COMPONENT. These types, in turn, serve as the supertypes for physical component types and software component types. E.g. ADLCOMPONENT and the other component types specific to configuration models describing Armani and Wright designs would now be derived from SOFTWARE COMPONENT, not directly from COMPONENT TYPE as presented previously. The suggested taxonomy for components is depicted in Figure 17.

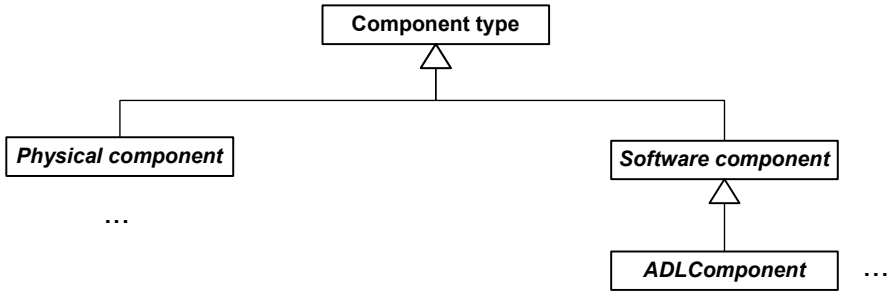


Figure 17 Taxonomy of component types for modelling products with embedded software.

7 Comparison with Previous Work

This section compares the work done in this thesis with related previous work.

Research closely related to this thesis has been conducted earlier. However, there seems to no research where the concepts of software architecture description would have been compared to those of configuration modelling. Thereby, comparing the concepts of ADLs with those of the configuration ontology is the main contribution of this thesis.

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 (2001). They have not, however, studied the concepts of ADLs in detail or proposed any mapping from these concepts to those of configuration modelling domain.

van der Hoek et al (1998; 1999), on the other hand, have developed a conceptualisation, or a basic architectural skeleton, as they call it, of ADLs that accommodates *evolution*, *variability*, and *optionality*. Out of these, evolution is out of the scope of this thesis and is not further discussed. But variability and optionality in ADLs are closely related to this thesis. van der Hoek et al. define variability as the ability of a single software system being able to provide multiple, alternative ways of achieving the same functionality. A numerical optimisation system is presented as an example of a system with variability: depending on the user needs, the system may be either fast and imprecise, or slow and precise; both alternatives are, however, incorporated into the system. The decisions about which alternative is used is done based on a boolean condition: if the condition for an alternative is true, the alternative is used. It is required that exactly one of the conditions in the alternatives is true at a time. Optionality, on the other hand, is defined as system having one or more additional parts that each may or must not be incorporated in the system. Continuing the numerical optimisation example, the ability to gather statistics can optionally be incorporated into a system. Whether to include a part in a configuration or not is decided by a boolean condition.

Although apparently dealing with similar issues, the work done by van der Hoek et al. is in many ways different from the work presented in this thesis. First, this thesis strives to find a mapping from software domain to the configuration domain, using ADLs as the conceptualisation of the software domain; their approach, on the other hand, is based on developing a conceptualisation of ADLs with features unknown in the current ADLs

from scratch. Second, variability as defined by van der Hoek is not really variability from the configuration point of view: their variability concerns the behaviour of a systems instance, not variability in what kind of system is produced during a configuration process. Third, from the configuration point of view, the notion of optionality as defined by van der Hoek et al. a very limited form of variability. In conclusion, the approach followed and the underlying goals are different in the work by van der Hoek and this thesis.

(Syrjänen 2000) 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 (2000). Basically, their approach could be used for creating configurators for software products as well. Their approach is, however, different from the one followed in this thesis: theirs is based on presenting configuration knowledge in UML, while the approach in this thesis is based on modelling software with the concepts of product configuration.

In (Kühn 2000), Kühn has presented an approach to software configuration based on structure and behaviour. In the approach, *domain objects* are decorated with properties and organised in taxonomic and compositional hierarchies. Statecharts, a method similar to finite state machines, are used for specifying the behaviour of modules.

Compared with the configuration ontology, the domain objects compare to components, and the taxonomy and composition hierarchy to taxonomy of component types and structure, respectively. Thereby, the concepts he uses for modelling software is subset of the concepts available in the configuration ontology, and of those of ADLs as well: Kühn's approach does not incorporate topology. On the other hand, the approach does model behaviour, unlike the configuration ontology, Armani and Koala. In conclusion, his approach is close to Wright in that the approach involves both structure and behaviour.

8 Future Work

The results achieved in this thesis are preliminary for at least two reasons.

First, as stated in the introduction, the goal underlying this work is to apply the research results achieved in the domain of configurable products to software product families as well. The results achieved in the thesis seem to bring this goal a step closer, but only a small step.

The results achieved so far should be concretised in the form of a language purported to model architecture of software product lines. The language should contain at least the concepts that were found relevant modelling software in this study. Further, the language should be defined formal semantics in terms of some suitable formalism in order to give it precise semantics and to allow analysing its properties and building support tools on it.

Of course, to be of practical value, the language's ability to model software product lines should be verified through case studies with real products in companies. The case studies will help to check if the set of concepts captured by the language is sufficient and minimal in the sense that it covers all the architecturally relevant aspects of the studied software product line and includes no unnecessary ones. Especially, the question whether to extend the ontology with the concepts that are included in at least one of the ADLs, but not in the configuration ontology, can be better answered based on empirical studies in real companies. Further, for the purpose of being of practical use, the language should be defined as an extension of some existing method currently used by software engineers; UML is a strong candidate for the method.

A new version of the modelling language should be built based on the results of the case study. As no modelling language without a supporting toolset is likely to ever be of practical value, a tool supporting the modelling task and configuration tasks should be designed and implemented. In more detail, the tool should support all the tasks related to using software product lines: from constructing a model of the family to producing individual software systems.

Finally, the validity of the second language version and the tool supporting it should be empirically tested and formally analysed. The former would be done through case studies in at least two companies with real software products and. The latter, in turn,

would be accomplished through analysing computational complexity of the tasks needed to support the concept of configurable software product families.

Of course, there are other important issues related to software product families than those addressed in the above-described research path.

As an example, the assumption that software product lines would be static is hardly viable. On the contrary, it is most likely that both the architecture of software product lines and the assets used for composing the systems will constantly evolve. The work by van der Hoek et al. (1998; 1999) and Männistö (2000) support this argument. Therefore, methods for handling the evolution of software product lines should be developed. Further, the methods for modelling software product lines in general and constructing instances of them should be integrated with those for handling the evolution.

9 Conclusions

In the beginning of this thesis, two domains were considered: the domain of configurable products, and the domain of software product lines. The goal of the thesis has been to study whether the concepts used for describing configurable products could be used for describing the architecture of software product lines as well. As there was no generally approved conceptualisation of software product lines, the concepts of three ADLs were analysed in detail to form a picture of the concepts used for modelling the structure of software systems. These concepts were then compared to a conceptualisation of configuration knowledge.

The comparison indicated that there are natural counterparts and close correspondences in the configuration ontology for the main elements of the studied ADLs. For instance, both share the notion of components. Furthermore, compositional structure, systems formed of connected components and constraints are phenomena present in both disciplines. In addition, for most of other concepts, satisfactory mappings to the ontology could be found; e.g. connectors could be mapped to a subtype of component. The mappings were defined in enough detail in order for it to serve as a specification for implementation. Hence, it was established that the concepts of the configuration ontology can be used for modelling software products.

However, capturing some aspects of ADLs would seem 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 (Kühn 2000). 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.

Although the results achieved in this thesis are based on detailed analysis and mappings, they subject to the validity of the assumptions made. Foremost, it was assumed that ADLs are a modelling method capable of capturing relevant aspects of the architectures of software product lines.

References

Allen, R. *A Formal Approach to Software Architecture*. Ph.D. thesis, Carnegie Mellon University. 1997.

Allen, R., Garlan, D. "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6(3): pp. 213-249. 1997.

Arnold, K. *The Java Programming Language*. Addison-Wesley, Boston (MA). 2000.

Bass, L., Clements, P. C. and Kazman, R. *Software Architecture in Practice*. Addison-Wesley, Boston (MA). 1999.

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

Clements, P. C. and Northrop, L. *Software Product Lines - Practices and Patterns*. Addison-Wesley, Boston (MA). 2001.

Czarnecki, K. and Eisenecker, U. W. *Generative Programming*. Addison-Wesley, Boston (MA). 2000.

Felfernig, A., Friedrich, G. and Jannach, D. "UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems", *International Journal of Software Engineering and Knowledge Engineering*, vol. 10(4): pp. 449-469. 2000.

Felfernig, A., Friedrich, G. and Jannach, D. "Conceptual Modeling for Configuration of Mass-Customizable Products", *Artificial Intelligence in Engineering*, vol. 15(2): pp. 165-176. 2001.

Garlan, D. "Software Architecture," in *Encyclopedia of Software Engineering*, J. J. Marciniak, ed., John Wiley & Sons, New York. 2001

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

Garlan, D., Monroe, R. T. and Wile, D. "Acme: Architectural Description of Component-Based Systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, eds., Cambridge University Press, pp. 47-68. 2000

Gruber, T. R. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", *International Journal of Human-Computer Studies*, vol. 43(5): pp. 907-928. 2002.

Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs. 1985.

Junker, U. "Preference Programming for Configuration", in *Proceedings of the configuration workshop of IJCAI 2001*. 2001.

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

Magee, J., Dulay, N., Eisenbach, S. and Kramer, J. "Specifying Distributed Software Architectures", in *Proceedings of the Fifth European Software Engineering Conference (ESEC '95)*. 1995.

Medvidovic, N. and Taylor, R. M. "Separating Fact from Fiction in Software Architecture", in *Third International Software Architecture Workshop (ISAW-3) in conjunction with SIFSOFT'98 (FSE-6)*. 1998.

Medvidovic, N., Taylor, R. M. "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, vol. 26(1): pp. 70-93. 2000.

Mehta, N., Medvidovic, N. and Phadke, S. "Towards a Taxonomy of Software Connectors", in *Proceedings of the International Conference on Software Engineering (ICSE)*. 2000.

Microsoft Corporation. *The Component Object Model Specification.*1995

Mittal, S. and Frayman, F. "Towards a Generic Model of Configuration Tasks", in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI)*. 1989.

Monroe, R. T. *Capturing Software Architecture Design Expertise with Armani.* Technical report CMU-CS-98-163. 2001

Monroe, Robert T., Garlan, David, and Wile, David. *Acme Reference Manual.* 2002

Männistö, T. *A Conceptual Modelling Approach to Product Families and Their Evolution*. Ph.D. thesis, Helsinki University of Technology, Finland. 2000.

Männistö, T., Soininen, T. and Sulonen, R. "Product Configuration View to Software Product Families", in *Proceedings of the Tenth International Workshop on Software Configuration Management (SCM-10) of ICSE 2001*. 2001.

Object Management Group. *OMG Unified Modeling Language Specification.*2001

Ommering, R. v. "A Composable Software Architecture for Consumer Electronics Products", *Xootic Magazine*, vol. 7(3): pp. 37-47. 2000.

Ommering, R. v. "Configuration Management in Component Based Product Populations", in *Proceedings of Tenth International Workshop on Software Configuration Management (SCM-10) - An ICSE 2002 Workshop)*. 2001.

Ommering, R. v. "Building Product Populations with Software Components", in *Proceedings of the 24th International Conference on Software Engineering*. 2002.

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

Pine, B. J. I. *Mass Customization - The New Frontier in Business Competition*. Harvard Business Scholl Press, Boston (MA). 1993.

Shaw, M. and Garlan, D. *Software Architecture - Perspectives of an Emerging Discipline*. Prentice Hall. 1996.

Smolander, K. and Päivärinta, T. "Describing and Communicating Software Architecture in Practise: Observations on Stakeholders and Rationale", in *Proceedings of the Fourteenth International Conference on Advanced Information Systems Engineering (CAISE '02)*. Springer-Verlag. 2002.

Soininen, T., Tiihonen, J., Männistö, T. and Sulonen, R. "Towards a General Ontology of Configuration", *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 12(4): pp. 357-372. 1998.

Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading (MA). 1997.

Syrjänen, T. "Including Diagnostic Information in Configuration Models", in *Proceedings of the First International Conference on Computational Logic*. Springer-Verlag, Berlin. 2000.

Thiel, S., Hein, A. "Modeling and Using Product Line Variability in Automotive Systems", *IEEE Software*, vol. 19(4): pp. 66-72. 2002.

Tiihonen, J., Lehtonen, T., Soininen, T., Pulkkinen, A., Sulonen, R. and Riihihuhta, A. "Modeling Configurable Product Families", in *Proceedings of the 12th International Conference on Engineering Design (ICED'99)*. 1998.

Tiihonen, J. and Soininen, T. *Product configurators - information system support for configurable products*. Techical report TKO-B137. 1997

van der Hoek, A., Heimbigner, D. and Wolf, A. L. *Investigating the Applicability of Architecture Description in Configuration Management and Software Deployment*. Technical report CU-CS-862-98. 1998

van der Hoek, A., Heimbigner, D. and Wolf, A. L. *Capturing Architectural Configurability: Variants, Options, and Evolution*. Technical report CU-CS-895-99. 1999