Modeling Variability in the Software Product Line Architecture of Distributed Middleware Services

Liliana Dobrica
Faculty of Automation and Computers Science
University Politehnica of Bucharest
Spl. Independentei, 313, Sect. 6, Bucharest, Romania

Eila Niemelä
VTT Technical Research Centre of Finland
P. O. Box 1100 FIN-90571
Oulu, Finland

Abstract - *The product line is defined as a middleware services framework that includes several products. The products realize different functionality by using various modern software technologies of spontaneous networks. UML provides the means to use specific variation mechanisms to describe hierarchical systems. However,it does not support a description of variation, as it is required for service architecture. UML built-in extension mechanisms refine its specification. This paper presents the extensions of the UML for representing variations in the software product line architecture of middleware services. Architecture design produces descriptions at two abstraction levels from multiple viewpoints. The modeling of service architectures benefits from a more familiar and widely used notation that improves stakeholders' understanding of the architectural artifacts. A standard based notation also enables more extensive tool support for manipulating architecture models.*

Keywords: software architecture, product line, variability, middleware, service.

1 Introduction

In the software product-line development the product-line architecture (PLA) is the main tangible element shared by all the product members. Architectural components cover functionality common for the products in the product-line and support the variability required for the various products. Due to increasing complexity in the middleware services, their architecture specification requires a more explicit approach. It becomes important to understand how to express variability to indicate locations in design for which different kinds of modifications, omissions and extensions are permitted, expected or required.

One of the most important aspects of PLAs is variation among products, defined as variation in space [4] [10]. In this paper, we introduce UML extensions for the design of this kind of variability. The main goal is to specify modeling constructs that deal with variability and represent a profile of the extended UML concepts intended primarily for use in distributed middleware services PLAs. We apply the new constructs on a case study. The case study validates the UML notation extensions for architectural models that are designed with a proprietary method, QADA®[1] [17].

In the case study we model the architecture of a software framework for distributed middleware services (DisMis) that includes the features, commons and variants, implemented in several related products. Because we consider the PLA as a framework we can derive the products in a more flexible manner and with a higher reusability degree. Our approach is based on analysis of different sources of variation. Moreover, the PLA model exploits not only styles and design patterns [13][5], but also separation of concerns, intensional (vs. extensional) and locality criteria [11]. By separating different aspects in distinct views we manage complexity. Intensional and locality criteria facilitate the decision about the information that goes into PLA and what implementation specific is.

There is a difference between our case study and the others surveyed in the literature [14] [16]. While other studies brings to front experiences from industry that promotes the current state of practice, we identify and apply concepts and principles about PLA modeling as a state of the art. We contribute in this way in theory development of the domain. The middleware definition as "a variety of distributed computing services and application development supporting environments that operate between the application logic and underlying system" [6] is similar to software product line definition that is "a set of products sharing a common, managed set of features that satisfy the specific needs of a particular mission" [7]. So, building the architecture of a framework for distributed middleware services is equivalent to modeling a PLA. Distributed computing services represent the shared common, managed set of features that satisfy the specific distribution needs.

The remainder of the paper is organized as follows. First we discuss about concepts related to service

[1] Registered Trademark of VTT Technical Research Centre of Finland

architecture description and variability. Our focus is on different mechanisms and sources of variability in service architecture design. Next section introduces our original ideas about modeling variability by UML extensions for service architecture description. Then we apply our ideas in a case study. Finally, in the last section we present some discussion and concluding remarks.

2 Background

2.1 Service architecture description

Modern distributed systems are software-intensive systems that embody service architecture and provide a variety of services for their users. Services are constructed by a set of software components, which are "*units of composition with contractually specified interfaces and explicit context dependencies"* [20].

A service is the capability of an entity (the server) to perform, upon request of another entity (the client), an act that can be perceived and exploited by the client. Service architectures have two abstraction levels: conceptual and concrete [17]. Conceptual means abstract, i.e. delayed design decisions concerning, e.g. technologies to be selected or details in functionality, whereas the concrete abstraction level illustrates the realization of conceptual architecture. Architecture design produces descriptions at both abstraction levels from four viewpoints: structural, behavior, deployment and development.

Table 1. Service architecture descriptions.

Level of abstraction	Views	Components	Relationships
Conceptual	Structural (decomposition model only)	System Subsystem Leaf	*Passes-data-to* «data» *Passes-control-to* «control» *Uses* «uses»
	Behavior (collaboration model only)	Service components	Ordered sequence of actions
	Deployment (allocation model only)	Deployment Node Deployment Unit	*Is-allocated-to*
Concrete	Structural (hierarchical structure diagram only)	Subsystem capsules Component capsules at level 1..n	Concrete interfaces with ports, connectors and protocols
	Behavior	State diagram Message sequence diagram	Ports, in and out signals Interactions between capsules
	Deployment	Node, device	Connection

The structural view is concerned with the composition of software components, whereas the behavior view takes the dynamics into consideration. The deployment view refers to the allocation of software components to various computing environments. Variation in space is an integral part of the first three views, contrary to the development view that represents the categorization and management of domains, technologies and work allocation. Entities of the first three views are summarized in the Table 1.

2.2 Variability

A variability mechanism is a wide range of generalization and specialization techniques. Jacobson [15] defines the following variability mechanisms: inheritance, uses, extensions, parameterization, configuration and generation. *Inheritance* is used to create subtypes or subclasses that specialize abstract types or classes at their variation point. Use case inheritance mechanism is for *uses*. *Extensions* are particular type-like attachments that can be used to express variant in use case and object components. *Parameterization* is used for types and classes using templates, frames and macros. *Configuration* variation points are used to declaratively or procedurally connect optional or alternative components and variants into complete configurations. *Generation* provides derived components and various relationships from languages and templates. When it is more suitable to select one mechanism over another and what are the consequences of a particular mechanism are questions that receive an answer in the paper of Svahnberg et al. [19]. The authors establish the factors that need to be considered for selecting an appropriate mechanism for implementing variability and identify two major mechanisms, configuration management and design patterns. The most commonly used design patterns are discussed in detail in [13] and [5]. Nevertheless, variation is difficult to model in architectural descriptions. A PLA may include a set of different alternatives for dealing with variation among products. Capturing these alternatives in various views facilitates the designer constructing a product to have the potential solutions to choose from. Identifying and separating sources of variation is a systematization of concerns regarding variability management.

Several sources of variation are introduced in [2]. There is v*ariation in data,* where a particular data structure may vary from one product to another. V*ariation in function* implies that a particular function may exist in some products and not in others. V*ariation in control flow* means that a particular pattern of interaction may vary from one product to another. V*ariation in technology* suggests that the platform (OS, hardware, dependence on middleware, user interface, run-time system for programming language) may vary in exactly the same mode as the function. V*ariation in quality goals* exists if particular quality goals are important for a product and v*ariation in environment* when the style in which a product interacts with its environment varies.

Since the focus of this paper is on how to describe variability on the architecture level, we will try to deal with variability that is visible in each view. In practice, analysis

of these sources is useful when the variability is architecturally relevant.

3 Modeling variability by UML extensions

UML provides the means to use specific variation mechanisms to describe hierarchical systems [3]. However, the standard does not support a description of variation, as it is required for service architecture. UML supports the refinement of its specification through three built-in extension mechanisms: constraints, tagged values and stereotypes. Tabular forms for specifying the new refinements need to be organized (Fig. 1). Stereotype tables columns identify stereotype name, the base class of the stereotype that matches a class or subclass in the UML metamodel, the direct parent of the stereotype being defined, an informal description with possible explanatory comments and constraints associated with the stereotype. Finally, the notation of the stereotype is specified.

Tabular form of a **Stereotype** definition
- Stereotype: Leaf
- Base Class: Subsystem
- Parent: Architectural element
- Description: ...
- Constraints: None or self.isMandatory=true
- Tags: None

Notation: A UML package stereotyped as «leaf»

Tabular form of a **Constraint** definition
- Constraint: isMandatory
- Stereotype: Leaf
- Type: UML::Datatypes::Boolean
- Description: Indicates that the Leaf is Mandatory

Tabular form of a **Tag** definition
- Tag: isDynamic
- Stereotype: Capsule
- Type: UML::Datatypes::Boolean
- Description: Identifies if the associated capsule class may be created and destroyed dynamically.

Fig.1. Examples of stereotypes, constraints and tag definitions.

We present in the following the main ideas about variability that can be realized in each view for service architecture description.

3.1 Conceptual

Conceptual structural view. Variation in this view is divided into internal variation (within Leaf components) and structural variation (between Leaf/ Subsystem components). Structural variation has to offer the possibility of preventing automatic selection of all Leaf or Subsystem components that are bind in a System during product derivation. We consider that a Leaf or a Subsystem could be stereotyped in:

- «mandatoryLeaf» or «mandatorySubsystem»
- «alternativeLeaf» or «alternativeSubsystem»
- «optionalAlternativeLeaf» or «optionalAlternativeSubsystem»
- «optionalLeaf» or «optionalSubsystem».

In the case of «alternative» or «optionalAlternative» variability of a Leaf or Subsystem, the inclusion of a letter "A" or "B", etc., at the bottom of the UML package symbol points to the product requiring that specific architectural element (Fig.2). Some of the constraints that govern variability modeling cannot be expressed by the UML metamodel. They concern the following:

- If a «mandatorySubsystem» only consists of «optionalLeaf» components, at least one of them must be selected during the derivation process; otherwise, a «Subsystem» that only consists of «optionalLeaf» components must be an «optionalSubsystem».
- Two «alternativeLeaf» or «alternativeSubsystem» components of different products are exclusive, meaning that only one can be selected for a product. The product is specified at the bottom of the notation.
- There should be no relationships between *alternative* or *optionalAlternative* components; they belong to different products. All relations to an *optional* component must also be optional.

The relationships of the structural view are appropriately stereotyped: «control», «data», «uses», «control (opt)», «data (opt)», «uses (opt)», «control (optAlt)», «data (optAlt)», «uses (optAlt)» (Fig.2).

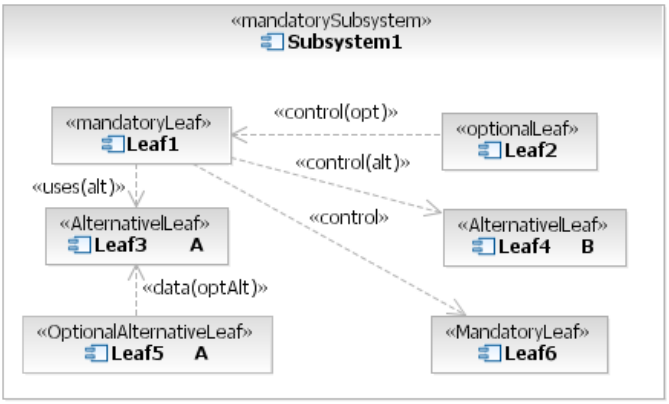

Fig. 2. Variation in conceptual structural view

We define *internal variation* only for Leaf components (Fig.3). A Leaf component is on the lowest level of a structure and it models functional requirements variable for different products.

Fig.3.Internal variation of a «mandatoryLeaf» component

The internal variation is designated by a ● symbol. Although the symbol is not included in the UML standard, Jacobson [15] and later Webber [22] introduced it for variation points. The UML tag syntax `vp<<m|o><VariationName>>|<<a|oa><VariationName><ProductId>>` shows the parts of an internal variation

so that the reuser can build a product. Mandatory (m) or optional (o) functionality (*VariationName*) of a Leaf component is specified in the tag syntax. In the case of alternative (*a*) or optionalAlternative (*oa*) the product identifier (*ProductId*) is also specified.

Conceptual behavior view. This view is mapped directly onto a hierarchy of UML collaboration diagrams. The elements are roles/instances of the Subsystem stereotypes defined in the conceptual structural view. Variable parts of a collaboration or interaction diagram can be represented with dashed lines or alternative branches. Optional messages between ServiceComponents use dashed lines with solid arrowheads. Collaboration diagrams describe each operation that is part of the requirements specification. Similar to the structural view, *alternative* and *optionalAlternative* ServiceComponents may be represented in this view. An identifier of the specific product that requires a particular interaction should be introduced and represented in the diagram.

Conceptual deployment view. In UML a deployment diagram shows the structure of the nodes on which the components are deployed. The elements related to a deployment diagram are Node and Component. DeploymentNode for service architecture is a UML Node that represents a processing platform for various services. The notation used for DeploymentNode is a Node stereotyped as «DeploymentNode». UML notation for Node (a 3-dimensional view of a cube) is appropriate for this architectural element. A DeploymentUnit is composed of one or more conceptual leaf components. Clustering is done according to a mutual requirement relationship between leafs. It cannot be split or deployed on more than one node. The stereotype, «deploymentUnit» is a specialization of the ArchitecturalElement stereotype and applies only to Subsystem, which is a subclass of Classifier in the UML metamodel. The other stereotypes «mandatory» ,«mandatoryActive», «mandatoryPassive», «optional» and «alternative» are specializations of the DeploymentUnit and also apply to Subsystem. *Exclude* is introduced as a new stereotype of UML association.

3.2 Concrete

Concrete structural view. The notation in this view includes a means to represent the decomposition of Capsule components. This feature allows step-by-step understanding of more and more details of the PLA. Decomposition is also used to show possible variations. A Capsule cannot only be decomposed into componentCapsules, but it can also be decomposed so that new functionality is revealed. Decomposition relationships exists between abstract components. Concrete components are obtained by specialization. The Capsules notation specifies a particular product (A) or a subset of products (B, C) at the bottom of the symbol. Looking top-down (Fig.4), the AbstractComponents encapsulated in the «TopCapsule» are decomposed into «subsystemCapsule» abstract components: CapsuleS1,.., CapsuleSN. Decomposition continues on «component1Capsule», « component2Capsule» and so on, if necessary. In each component, abstract functions of the corresponding sub-domains are collected, which are subsets of the parent abstract functions. For each product, each abstract component is specialized in a «concrete Component». This view may include indication of products or product sets, thus providing information about the reusability of each component.

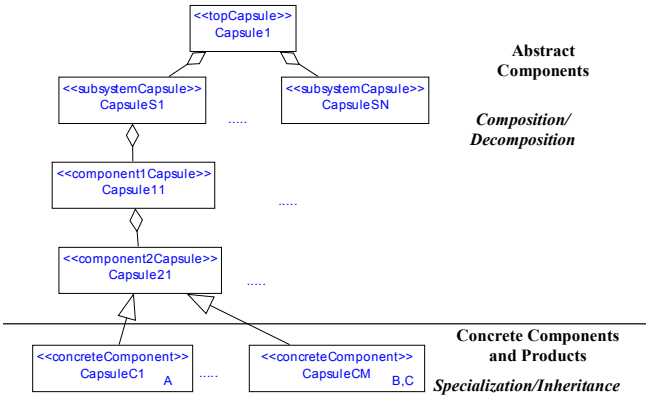

Fig.4 Variation in concrete structural view.

Concrete behavior view. This view is mostly modeled using two main diagrams: a state diagram and a message sequence diagram. This view describes how the system reacts in response to external stimuli. State diagrams are used with the concrete structural view's entities: capsules, ports and protocols. Standard UML state diagrams are recommended for modeling the behavior of capsules, which in combination with inheritance facilitates reuse. Variability is included in notation and state decomposition. As for notation, parts that are not needed in all products are represented with dashed lines (optional states) or a different filling pattern and Product_Id (alternative states). State decomposition is the other source for variants. The decomposition of a state may be shown by a small symbol in the top left corner of a state symbol.

Concrete deployment view. This view is mapped directly on the deployment diagram of UML. UML deployment diagrams are less well explained in the standard than other elements of UML. However, nodes - the UML elements which represent processing elements – are Classifiers in UML, which means that they can have instances, play roles in collaborations, realize interfaces, etc. They can also contain instances of components.

4 Case study

4.1 Description and analysis

The starting point of our exercise represents descriptions of core classes implemented for the four products that focus on distribution of middleware services [21]. The products are: a dynamic distributed platform (DDP), a Bluetooth connectivity component for Java

(BCC), a Jini service framework (JSF), and a video camera demonstration (VCD) for JSF. DDP, is a framework for distributed applications. DDP performs all that is required for the task of distribution, to connect pieces of distributed applications that may reach each other over various means of communications. BCC is an example of a component that can establish connectivity through new media, protocols and connectivity methods. BCC is used in applications that employ spontaneous wireless communications. The JSF, is described as a) It creates a set of extensible classes that would automatically perform the most important functions required from a Jini service or service user. b) It transfers legacy client-server applications into Jini services with minimal changes, and c) It creates a new Jini service that has distinct types of services for a service user and a service session. The VCD system represents a distributed application that proves the functionality of JSF. The provided service is a live video stream from a camera. The service user needs all necessary code from the service for viewing the video stream. The differences between these products express variation in our PLA.

In the following we will analyze each source of variation that brings variants for DisMiS PLA. *Variation in function:* DDP, JSF and VCD have the same functional requirements, i.e. distribution of applications. BCC requires a different communication protocol. *Variation in data:* We distinguish VCD that uses multimedia streams. *Variation in control flow:* We identified a proxy pattern in DDP, JSF and VCD, asynchronous operations in BCC and, a lookup service in JSF. *Variation in technology:* Jini technology, RMI and TCP/IP are necessary in JSF and VCD, Nokia DTL1 connectivity card and Windows OS are required by BCC, and JPG format for streams is used in VCD. *Variation in quality goals:* Interoperability, scalability, adaptability, and fidelity are few of the quality characteristics that could vary for this framework. Reusability, maintainability, modifiability, portability and extensibility may vary when evolution is of concern. *Variation in environment:* A particular component of our framework may be invoked from either C++ or Java. The invocation mechanism may vary from one product to another. Java Media Framework is required by VCD.

4.2 Modeling DisMiS PLA variability

Designing the architecture of the DisMis framework gives us several challenges. Through the analysis of each product we discovered one by one the common abstract components and variant ones. We assumed DDP as a reference product because it is the most complete as PL functionality. Then we design our framework by reengineering. Results are presented in the following.

Conceptual level. The conceptual structural view of the framework represents a set of layers, *«subsystem»*-s that have assigned modules (Fig.5). Modules are *«leaf»*-s or other *«subsystem»*-s . For practical reasons we didn't represent the whole decomposition from high-level to lower-level/detailed in one diagram. This would become very difficult for reading by other stakeholders, users of this view. The level of description of conceptual components is appropriate when it reveals an understanding of the coarse-grain common and variable features included in each product member. From the point of view of a PL DisMiS has several particularities. Firstly, BCC is included in the PL as a single feature and is represented by a *«leaf»*. Thus, we get a variant introduced in a *«leaf»*. *CommunicationProtocol «leaf»* internal variation is specified as •*vpa alternative BluetoothProtocol*. Secondly, the re-design of the Discovery Service *«subsystem»* of DDP was necessary. Based on principles about separation of concerns and locality the components with communication features are gathered in the most appropriate *«subsystem»*. Thus, *Multicast Communication*, *DataResolver* and *UDPProtocol* are all configured in the *CommunicationServices «subsystem»*.

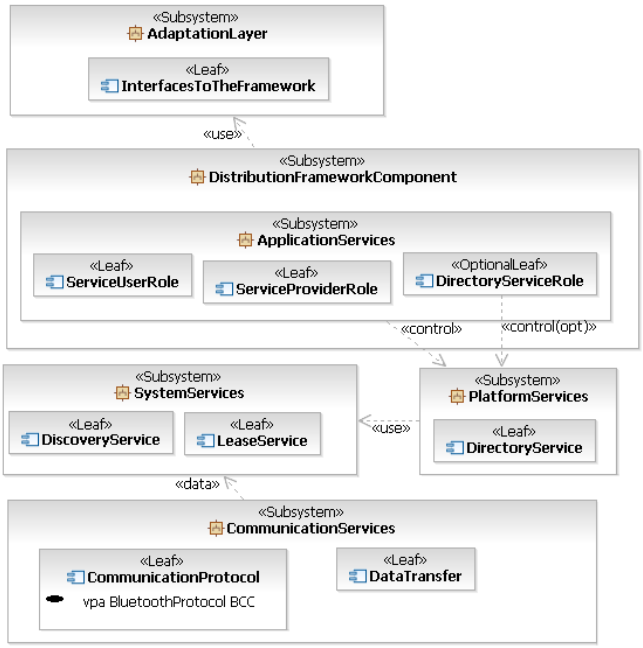

Fig. 5. Conceptual structural view

The conceptual behavior view of the DisMis framework specifies the dynamic actions the system produces and participates as well as their ordering and synchronization and gives an understanding about the dynamic aspects of services. We analyzed the behavior of the conceptual components associated in JSF. We discovered that the presence of the *Lookup Service* in this diagram gives a dependency on Jini technology. This cannot totally suit on the concept of distributed services for a spontaneous environment. This dependency has been managed in DisMiS by applying a variability mechanism. The distributed system has to stay functional even though the directory service may be shut down. We introduced a conditional clause (parameterization). Thus, a branch in collaboration diagram (emphasized with solid arrows) on

the *active* condition gives a service possibility to communicate with another active service in the network.

The conceptual deployment view consists of DeploymentNode components and DeploymentUnit components allocated to nodes. UML stereotypes of different types of deployment units are necessary in this view. It is significant to note «mandatory», «mandatoryActive» and «mandatoryPassive» variants deployment units in nodes.

.

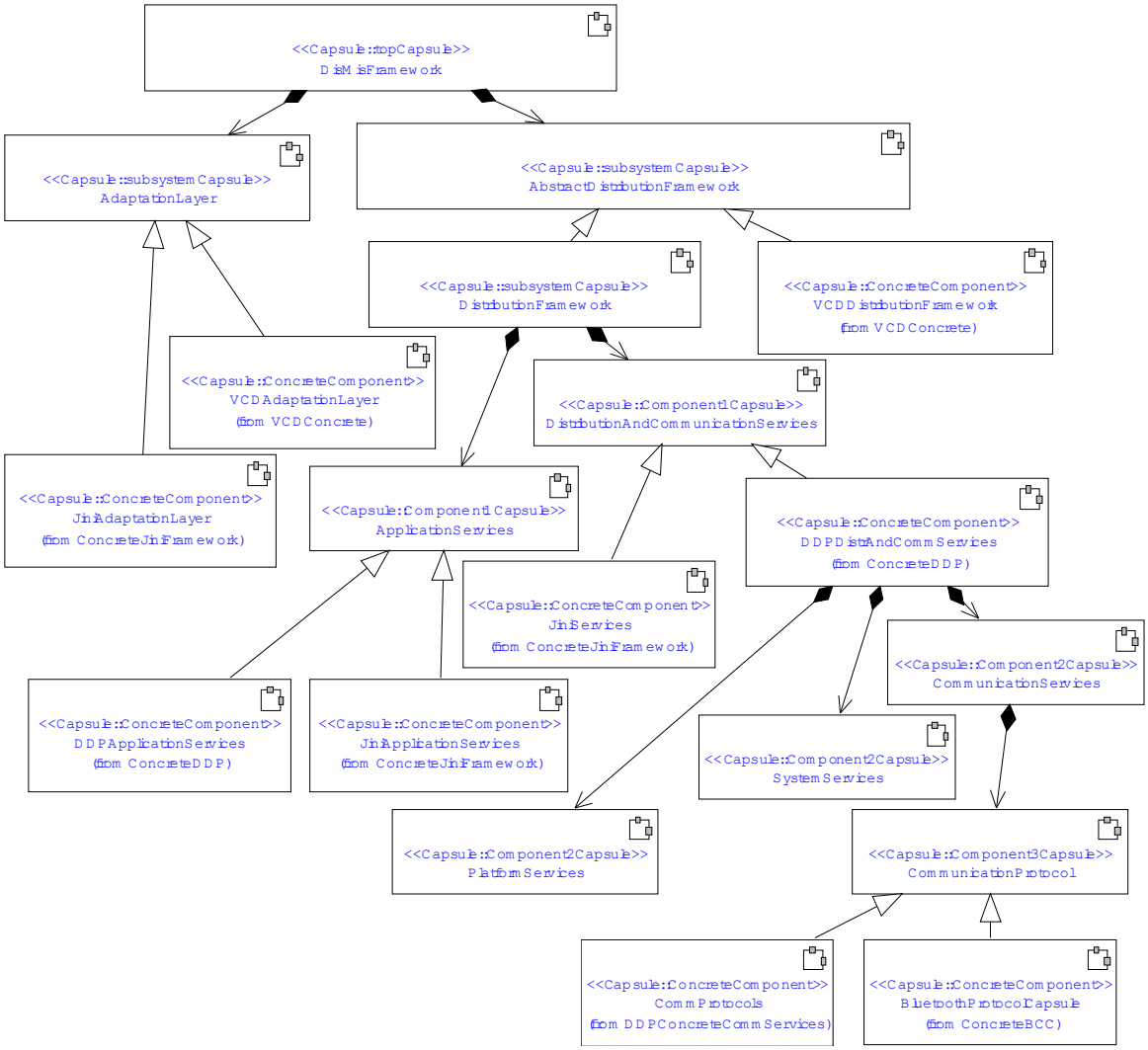

Fig. 6. Variation of products in the concrete structural view of DisMiS PLA.

Concrete level. Structural, behavior and deployment views have been developed for modeling architecture on this level. Due to the limited space, we discuss only about the variability modeling. Variability from various sources is represented in the concrete structural view by using architectural elements and relationships extended from the UML standard. Thus, components are subsystem capsules or component capsules at level 1..n, and relationships are concrete interfaces with ports, connectors and protocols.

Decomposition relationships describe the hierarchical structure of abstract components (Fig. 6). The vertical range of abstract components consists of several levels, from the *topCapsule* component (*DisMis Framework*) to *component3Capsule* level component *(Communication Protocol)*. Inheritance mechanism and specialization are used to obtain various concrete components. Products could be DDP, Jini, BCC or VCD and their acronyms are automatically included by the modeling tool in the name of each component.

5 Discussion and conclusions

This paper has described how UML can be extended to address the challenges of variability in space of software PLAs for middleware services. A new UML profile has been defined to be integrated in a systematic approach. UML extensibility mechanisms are used to express diagrammatic notations of each view of the architecture. Integrated use of a profile and a design method allow extensive and systematic design of software PLAs.

Other researchers have tried to use ADLs or extend UML for variability specification in PLA. Like us, they have introduced new symbols tagged to UML elements. In PRAISE [12] UML package represents a hot spot with the stereotype <<hot spot>> any collaboration is tagged with a variant with "variation point". Variability is also visible in the UML models with the variation points technique defined in SPLIT [8] providing information for a reuser to choose a variant. The mechanism of attaching attributes to each variation point, by using a class to represent it, defines the transformation to apply when doing a derivation. However, using this technique systematically requires development of specific scripts and programs to manage it, since it is not integrated in UML design tools. Webber [22] goes a step further and shows a reuser how to build a variant in VPM.. This study inspired us in extending UML notation.

Explicit modeling of the similarities and variations among members of the product lines by using the UML notation is allowed using various views. We identified KobrA [1] and a view integration approach described in [14]. All these methods have some similarities with our approach, but none of them provides PLA extensions that could be used to describe all kinds of variations possible in PLA. PLA has been prepared for change by studying various sources of variability that are visible in architecture description. Concrete components were defined for the prime reason of encapsulating variabilities, while abstract components were provided with the optimal balance generic/specific based on the different sources of variation analyzed for each product. Recently, our approach support representation of variation in functionality, data, control, technology and environment. Variation in quality goals is partly supported. Our future focus is on representing execution qualities, such as reliability and availability, in architectural models. Execution qualities are important in service architectures that our model is intended for.

6 References

[1] Atkinson C., J. Bayer, D. Muthig, "Component-based Product Line Development: The KobrA Approach", SPLC1, Kluwer Academic Publishers, pp. 289-310, 2000.

[2] F. Bachmann, L. Bass, "Managing Variability in Software Architectures", ACM SIGSOFTSoft. Eng. Notes, vol. 26, pp. 126-132, 2001.

[3] Booch, G., Rumbaugh, J., Jacobson, I., "The Unified Modeling Language User Guide", Addison-Wesley, 2004, 2nd Ed.

[4] Bosch J., "Design&Use of Software Architecture – Adopting and Evolving a Product Line Approach", Addison-Wesley, 2000.

[5] Buschmann F., Jäkel C., Meunier R., Rohnert H., Stahl M., "Pattern-oriented Software Architecture – A System of Patterns", John Willey&Sons, 1996.

[6] Charles J.,"Middleware moves to the forefront", Computer, vol. 22, pp. 52-, 1999.

[7] P. Clements, L. Northrop, "Software Product Lines - Practices and Patterns", Addison-Wesley, 2002.

[8] Coriat M., J. Jourdan, F. Boisbourdin, "The SPLIT Method", SPLC1, Kluwer Academic, pp. 147-166, 2000.

[9] Dobrica, L., Niemelä, E., "Using UML Notation Extensions to Model Variability in Product-line Architecture", Procs. of ICSE'03 Wshp. On SVM, Oregon, , 2003, pp. 8-13.

[10] Dobrica L., Niemelä E., "A strategy for analyzing product line software architectures", VTT Publications, 2000.

[11] Eden A.H., Kazman R., "Architecture, Design, Implementation", ICSE 2003.

[12] El Kaim, W., Cherki, S., Josset, P., Paris, F., "Domain Analysis and Product-Line Scoping", Procs. of SPL: Economics, Architectures, and Implications, June 2000.

[13] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-oriented Software", Addison-Wesley , Reading MA, 1995.

[14] Gomaa H, M. Gianturco, "Domain modeling for WWW based on Software Product Lines with UML", ICSR-7, LNCS 2319, pp. 78-99, Springer-Verlag 2002.

[15] Jacobson, I., Griss, M., Jonsson, P., "Software Reuse-Architecture, Process and Organization for Business Success". ACM Press, New York, NY, 1997.

[16] Jaring M., Bosch J., "Representing Variability in Software Product Lines: A Case Study", SPLC2, 2002.

[17] Matinlassi, M., Niemelä, E. Dobrica, L. "Quality-driven Architecture Design and quality Analysis method", VTT Publications 456, Espoo, Finland, 2002

[18] Purhonen, A., Niemelä, E., Matinlassi,, M.. "Viewpoints of DSP software and service architectures". JSS, Vol. 69.

[19] Svahnberg M., J van Gurp, J. Bosch, "A taxonomy of Variability Realization Techniques", Blekinge Institute of Technology, Sweden 1103-1581, 2002.

[20] Szyperski C. "Component Software –Beyond Object – Oriented Programming", Pearson Ed., Harlow UK, 1997.

[21] Vaskivuo T., "Software architecture for decentralised distribution services in spontaneous networks", Espoo 2003, VTT Publications 490.

[22] Webber D. and H. Gomaa, "Modeling variability with the variation point model", ICSR-7, LNCS 2319, pp.109-122, Springer-Verlag Berlin Heidelberg 2002.