A UML based methodology to ease the modeling of a set of related systems

Firas Alhalabi, Mathieu Maranzana, Jean-Louis Sourrouille

A UML based methodology to ease the modeling of a set of related systems

Firas Alhalabi, Mathieu Maranzana, Jean-Louis Sourrouille
Université de Lyon
INSA-Lyon, LIESP, Bat. B. Pascal, F-69621, Villeurbanne Cedex, France
{Firas.Alhalabi, Mathieu.Maranzana, Jean-Louis.Sourrouille}@insa-lyon.fr

Abstract

Despite progress in model engineering, modeling large distributed systems is still a long and complex task. This paper outlines a methodology based on UML to make the modeling of a set of related systems simpler. A generic component-based framework specifies the commonality and variability of these systems at high-level of abstraction both from structural and behavioral viewpoint. Then, models of specific systems are derived from the coarse-grained generic framework. A case study in the field of QoS management systems illustrates this approach.

1. Introduction

Modeling large-scale distributed systems is complex and requires a great knowledge of the application domain. Modeling a second system in the same application domain is much easier. Further, having available knowledge about an *application domain* (*domain* in short) helps software development, and avoids starting from scratch. This paper proposes (i) an approach to model a domain at a high abstraction level based on UML 2.1 [5]; (ii) a methodology to aid the development of specific systems in this domain.

The aim is close to works such as, *Component-Based Software Engineering* (CBSE [4]), and *Software Product Lines* (SPL [2]). In these approaches, the consistency of component assembly should be checked either manually or automatically, while in our approach the framework itself enforces such constraints.

The basic concepts of the proposed solution are framework, component and model centric development. Component-based system design and framework-based development are well-known software engineering practices that reduce system complexity by separating concerns, and facilitate software reuse. The need of abstract models that do not depend on specific applications is in line with current trends in software development change from code-centric to model-centric [8].

The rest of the paper is organized as follows: section 2 describes component models in standard UML, and defines required UML extensions. Section 3 proposes our methodology aiming to facilitate the development of specific system models from a generic framework in a given domain. To illustrate our approach, section 4 presents a case study in the domain of *QoS Management System* (QMS), while section 5 discusses related works. Finally, section 6 concludes these works.

2. UML Notations

2.1. Definitions

According to the UML superstructure document, "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment". A component is a black-box container from an external point of view, but its internal structure can be described, particularly using subcomponents. The internal structure describes roles and multiplicities of subcomponents as well as their connectors. A multiplicity greater than one specifies that a subcomponent, viewed as a type, plays various roles in the context of the composite component (C plays the roles c_1 and c_2 in Figure 1).

2.2. UML Extensions

As a general purpose language, UML does not supply domain specific concepts. Fortunately, it includes mechanisms to add new concepts and to extend the language. UML profiles are interchangeable models gathering together related language extensions. To enforce the semantics of the concepts, UML supplies a language (OCL [5]) to specify constraints that apply to models, thus restricting the use of model elements to keep only meaningful expressions. The following sections define our stereotypes and constraints to make UML more suitable for component-based modeling.

2.2.1. Component. A UML component specifies a type and can be realized in different manners. The realization of a component is independent of the realization of its underlying subcomponents. For example in Figure 1, the realization of the component *A* does not depend on the realization of its subcomponents *B* and *C*. Finally, a component can be implemented by a set of classes or even external non-detailed components.

The internal structure of a component shows how the subcomponents carry out the exported services. Component interfaces specify signatures of public features such as operations and signals. In Figure 1, components interact with each other using assembly connectors through provided and required interfaces associated with ports. A component can be substituted for a component providing equivalent services based on interface compatibility: in UML, substitutability is characterized by a dependency relationship with the stereotype «*substitute*».

2.2.2. Component Inheritance. *Component* is a subtype of *Class* in the UML metamodel, but UML does not redefine the *Generalization* relationship for components, hence the relationship between classes applies to components. However, classes and components are different from several points of view. To describe our framework, we decided to make explicit some definitions and restrictions:

- Component *Realization* in the UML sense is always described using an arrow with a dotted line, and never using nested components to avoid confusing with subcomponent containment;
- Within a component, subcomponents are of type *Property* in the UML metamodel. In principle, when the containing component is created, instances of properties are created, but instance creation of a *Component* has no precise semantics. To cope with this problem, we assume that instance creation occurs only for properties of type *Class*;
- Components intercommunicate only through ports owning required and/or provided interfaces. This constraint avoids direct connection via non-typed connectors.

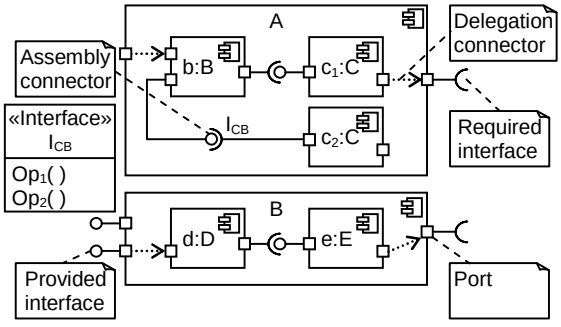

Figure 1. Internal structure of components *A* and *B*.

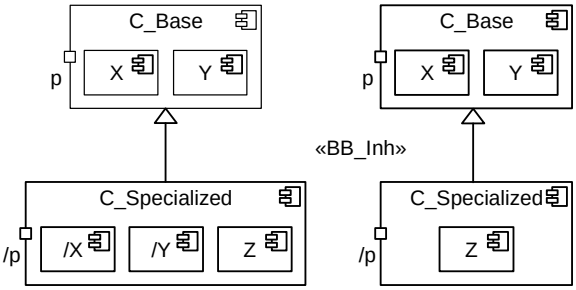

Figure 2. Inheritance (left) and «*BB_Inh*» (right).

According to the *Generalization* relationship, components inherit ports from their ancestors. Port inheritance ensures that any component is substitutable for its parent from the connections' point of view. When descendants redefine ports, interface conformance between general and specific ports should be enforced to ensure substitutability from the invoked services' point of view. The internal structure of visible subcomponents, i.e., not private, is inherited. In essence, a component is a black box whose obvious public elements are only ports and interfaces. To consider these ideas, we define two stereotypes:

(i) The *«BB_Inh»* stereotype (Figure 2) extends the *Generalization* metaclass and prevents a component to inherit subcomponents from its ancestors. This constraint restricts the inherited members for *Component* elements. The internal structure of the ancestor is ignored.

(ii) The *«P2P_Inh»* stereotype (Figure 3) extends the *Generalization* metaclass by making port interfaces precise. The redefinition of port *p* by *p_Spec* implies rules that are similar to pre/postconditions. Regarding an interface as a contract:

- *p_Spec* cannot require more than *p*, hence the required interface of *p_Spec* is included into the required interface of *p*, e.g., *I_Base_Required* inherits from *I_Specialized_ Required*;

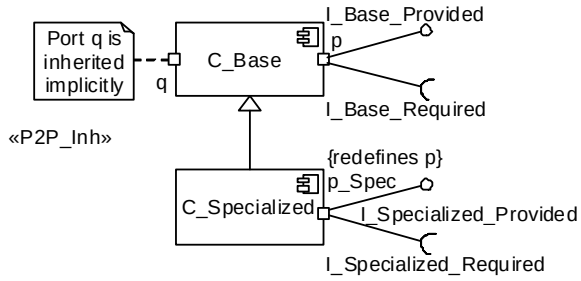

Figure 3. «*P2P_Inh*» stereotype.

– *p_Spec* should provide at least as much as *p*, hence the provided interface of *p_Spec* inherits from the provided interface of *p*, e.g., *I_Specialized_Provided* inherits from *I_Base_Provided*.

Another way to enforce this constraint would be to use the dependency relationship with the «*substitute*» stereotype, but this requires describing matching ports, i.e., the substitution of component C_1 with ports p_1 and p_2 for C_2 with ports q_1 and q_2 requires mapping each q_i to a p_j. In our proposal, port matching is clearly shown by the redefinition.

3. Proposed Approach

Our approach takes into account both structural and behavioral aspects. For the sake of place, we focus on structure only.

3.1. Process description

The proposed approach is in line with *Model Driven Development* (MDD) [5]. The main idea of MDD is to separate conceptual concerns described in *Platform Independent Model* (PIM), from implementation-specific concerns described in *Platform Specific Model* (PSM). PIMs can be successively refined, but when a *Platform Model* (PM) is added to the refined PIM, it becomes a PSM. Our approach first defines a generic model describing a family of systems in a given domain. Then, specific models are derived from the generic model. Generic and specific models both are platform independent models. Ideally, after transforming the most detailed PIM into suitable PSMs, the code of applications is automatically generated. Figure 4 summarizes the proposed development process distinguishing two main roles: the *architect*, who is assumed to be an expert in the given domain, and *developers* who derive the generic model to design specific models. A UML expert may assist the architect in defining profiles. The development process is outlined as follows (numbers refer to Figure 4):

(1) The architect creates a UML profile called *Generic Profile* (a) that defines all the concepts related to a family of systems in a domain. These concepts are described by stereotypes, which are user-defined metaclasses, and constraints.

(2) The architect designs a generic component-based architecture that captures commonality and main variability for a family of systems in a domain. The architecture includes components and their roles, connectors, ports, interfaces and constraints. The architect also provides generic sequence diagrams to describe dynamic aspects, i.e., interactions and communications between components.

(3) The architect describes derivation constraints in a Derivation Profile (b).

(4) From the generic model, each developer builds his/her specific model through transformations. Several transformations are common to all areas and others are area-specific. The latter are done manually because they are not reusable. The former are defined apart from any area description into a *Common Profile* through a set of stereotypes that mark elements and specify transformations to be executed. For instance, model elements marked with stereotype *«Remove»* will be removed from the model during transformation. Developers start from the generic model, mark elements with stereotypes of the common profile, and then execute model mapping by using a model transformation language. Finally, they manually transform the specific model. The derived model should conform to constraints (a), (b) and (d). During this step, developers may define Specific Profiles, and check constraints (c). Moreover, they will have to ensure model consistency.

(5) As in usual development processes, specific models may be refined.

(6) The components of specific models are still specifications. At implementation time, they will be instantiated to represent context-dependant implementations (PSM).

(7) Finally, the code skeleton is automatically

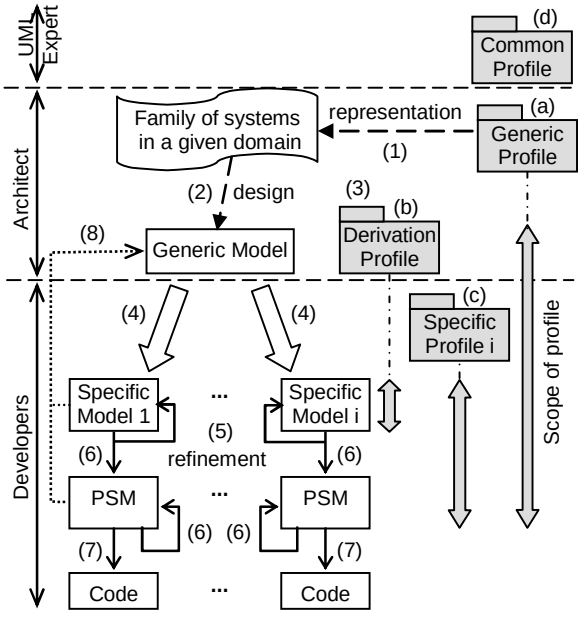

Figure 4. Outline of our process.

generated and developers fill in methods. At detailed level, descriptions are specific, but should meet higher-level constraints such as interfaces.

(8) According to the experience gained during modeling, developers may ask the architect to enhance the generic model.

3.2. Constraints

Within our development process, we distinguish three types of constraints (letters refer to Figure 4). These constraints are specified either in natural language or preferably in OCL to allow automatic checks:

(a) *Generic Constraints* are defined by the architect, and apply to all the models (scope Figure 4**)**. They guide the use of the domain notions expressed as components, ports and connectors. For instance, in a *Client/Server* pattern, a generic constraint may reduce the licit connections: "*each Client must be associated with at least one Server"*.

(b) *Derivation Constraints* apply to any derived specific model, ensuring it remains licit after derivation. For instance, to complete the above generic constraint, the architect may add "*any Server should have at least one Client"*. This derivation constraint forbids a cut off *Server*, while the previous generic constraint did not.

(c) *Specific Constraints* are defined by the developer, and apply to the derived and refined models, including PSMs. For instance, to achieve the chosen example, a specific constraint may be written: "*each Client is associated with exactly one Server"*. This last constraint defines the expected peer-to-peer connection in the refined model.

3.3. Model Derivation and Refinement

The architect specifies the semantics of the area by describing its concepts and their allowed interactions. Most of these semantic constraints are implicitly described within the generic model. In addition to generic and derivation constraints, these constraints specify rules that ensure the consistency of component assembly. Moreover, automatic transformations enforce component composition rules. Developers should check constraints that are not expressed in OCL. Anyway, we assume that the developer uses an *Integrated Development Environment* handling profiles and constraints, performing model transformations from a high-level description, and providing an easy way to describe and check constraints.

4. Case Study

Our team has a good experience in the development of QMS, which explains the choice of the QMS field to illustrate our approach. Beside our works, e.g., DCBL [9] and PMQDA [10], other solutions have been proposed, e.g., Quartz [7]. Each system has a specific model depending on its particular requirements. This section first defines a generic component-based framework aiming to aid the designer in the domain of QMS. Then, a specific model for the PMQDA distributed QMS is derived from the generic framework. In addition, we discuss the problems that we encountered regarding the derivation process.

4.1. Generic Profile Specification

The *Generic Profile* defines notions (vocabulary) as stereotypes that mainly extend the UML *Component* metaclass. The notions that we identified in the QoS management domain are for instance manager, admission, monitoring and mapping. In Figure 5, stereotypes mark components to specify their meaning and to ensure traceability after derivation. The semantics of the domain induces constraints that are attached to these stereotypes. For instance, in Figure 5a "a *LocalManager* (*LM*) cannot be connected with another *LM"* or "the root of the managers hierarchy along the *supervise* connector should be a *GlobalManager* (*GM)"*.

Our generic stereotypes «*BB_Inh*» and «*P2P_Inh*» and their constraints are included into the *Generic Profile* not to multiply profiles since they are common to all models.

4.2. Generic QMS Framework Design

4.2.1. Architecture. Figure 5a depicts the architecture of a generic framework for QMS (some connectors are omitted to simplify). The top-level composite component *G_Manager* controls the QMS. A G*_GlobalManager (G_GM)* controls one or more *G_ LocalManager*s (*G_LM)* through the *supervise* connector.

G_Adaptation and *G_Policy* (Figure 5b) set application behavior to improve the overall QoS of the system. Thanks to the «*P2P_Inh*» stereotype, *G_MixedPolicy* and *G_SimplePolicy* both inherit the ports and interfaces of *G_Policy*. *G_MixedPolicy* may include simple or mixed policies. Policies are for instance *G_Planning*.

For each application, the component *G_Local-Application* implements the business logic merged with

the execution template required for adaptation service. According to the execution context, the *G_Manager* sends orders to change application behavior, hence to tune their resource consumption to fit the current context. The *G_Loader* launches *G_LocalApplication* using *G_OS* services, possibly after negotiation and adaptation of running applications. The *G_OS* component, included in all QMS, allows components to use both standard operating system services and specific layers. For further details from the QoS point of view, the reader may refer to [1].

Components' names in the generic framework do not matter. Stereotype tagging is the only meaningful information to identify components during model transformation. Hence, in the sequel, component «*X*» means a component marked with the stereotype «*X*» or inheriting from a component marked with «*X*».

4.2.2. Comments. Since the general architecture aims to apply to any QMS, the components are rather abstract and limited to commonality. Thus, developers will adapt the generic model to build the specific system model that deals with their own requirements. For instance, the generic model assumes that when an application is rejected at admission time due to a lack of resources, a negotiation process is initiated to find a new operating point, which leads to several message exchanges between applications and managers (sequence diagrams are not shown). When the specific system does not supply negotiation, the component **«***Negotiation***»** has to be removed from the derived model and consistency should be checked. On the other hand, since admission is a basic service for QMS, the component **«***Admission***»** appears in any QMS.

To derive the component *G_Policy*, we distinguish simple from mixed policies. In the former case, the derivation process is a substitution of the component *G_Policy* for any single component inheriting from *G_SimplePolicy* (Figure 5b). In the latter case, *G_Policy* is replaced by the component *G_MixedPolicy* that may encompass components inheriting from *G_Policy* and supplying policies used in the specific system, e.g., *G_Learning* and *G_ExpertSystem* in DCBL [9]. In both cases, the substitution is performed using «*P2P_Inh*».

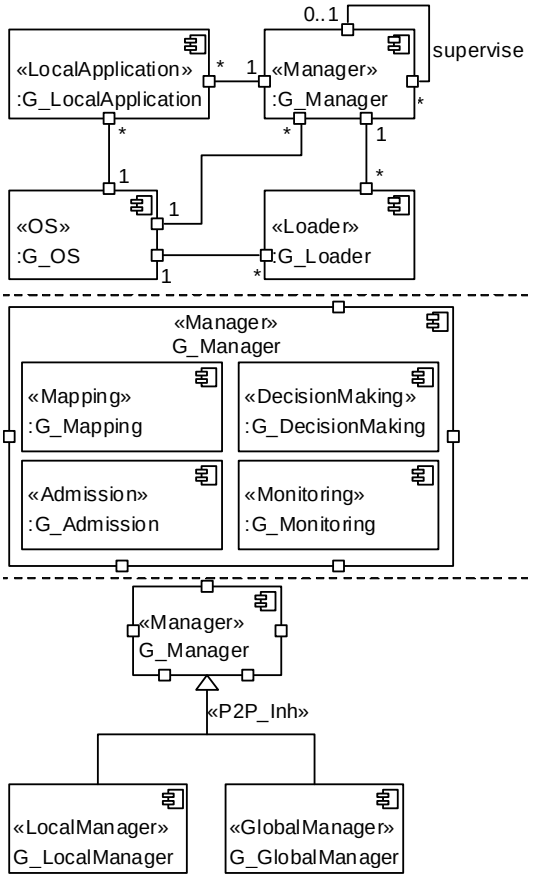

Figure 5a. Component-based generic architecture for QMS.

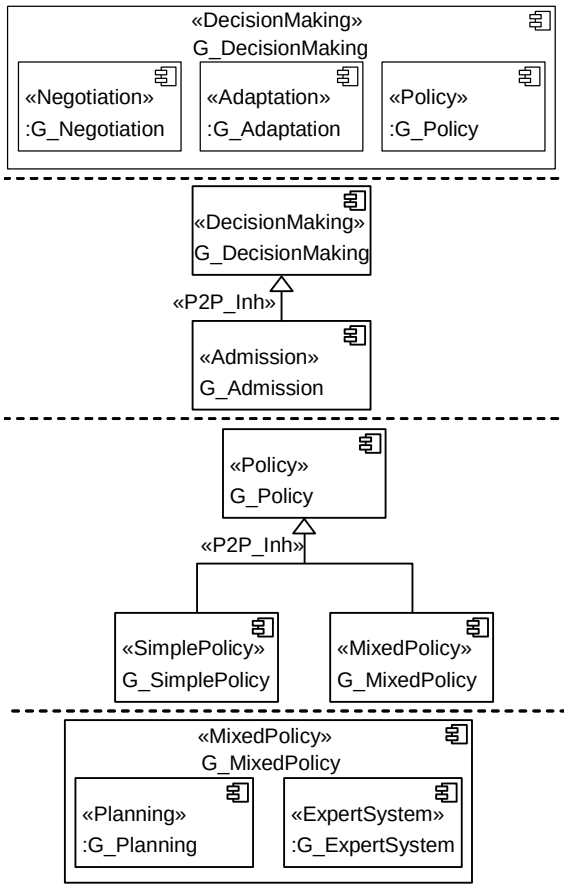

Figure 5b. Component-based generic architecture (cont'd).

4.2.3. Discussion. The nature of the derivation process depends on the modeling approach used to describe the domain, which explains why the choice of this approach was a central issue. A first way could be to define base components with the required properties, and then to derive specific components to inherit properties. This appealing solution is suitable for classes, but inheritance proves to be too strong a relationship for components.

In our approach, we start from a transformation of the generic model to keep only the needed components and relationships. This chosen approach, easy to understand by UML readers, seems the best compromise between derivation flexibility and effort to write the generic model and constraints.

4.3. Derivation Constraints Specification

The derivation constraints must be satisfied for the first specific model derived from the generic model. Examples of such constraints are:

- One manager must remain in any derived model, either *LM* or *GM* in case of single node;
- At least one manager must include a component «*Policy*»;
- Since admission is a basic service in QMS, at least one component «*Admission*» must remain in any QMS. «*Mapping*» may be removed.

4.4. Derivation and Refinement of the PMQDA Models

4.4.1. Specific Constraints. The developer defines and adds specific constraints to the PMQDA's specific profile, which applies to PMQDA models only. Examples of specific constraints (Figure 6):

- Components *LM* and *GM* do not include any component «*Negotiation*» because PMQDA does not consider negotiation mechanism;
- A model of the PMQDA distributed system includes a unique *GM*, and one *LM* on each node;
- A *LM* does not include any «*Policy*» component because resource management is fully centralized in the *GM*;
- Developers may refine generic stereotypes to add particular notions and constraints related to their specific requirements, for instance in Figure 6 «*SpecificLM*» refines *LM* to add a constraint ensuring that local managers are leaves regarding the *supervise* relationship.

4.4.2. PMQDA Architecture. The specific model of PMQDA in Figure 6 is derived from the generic model shown in Figure 5. Then generic and specific constraints are checked. For instance, when the multiplicity of a connector becomes 0 (from *PM_GM* to *PM_Loader* in the PMQDA model), the connected port is removed since it is useless. On the other hand, the port *p* that connects *PM_LM* to *PM_Loader* is kept. After derivation, the model encompasses elements as follows:

- *PM_GM*: this supervisor controls the execution of *PM_LocalApplication*s and applies the *PM_Planning* policy that schedules the use of the resources for all the concurrent applications. Thanks to *PM_Adaptation*, the *PM_GM* tunes the

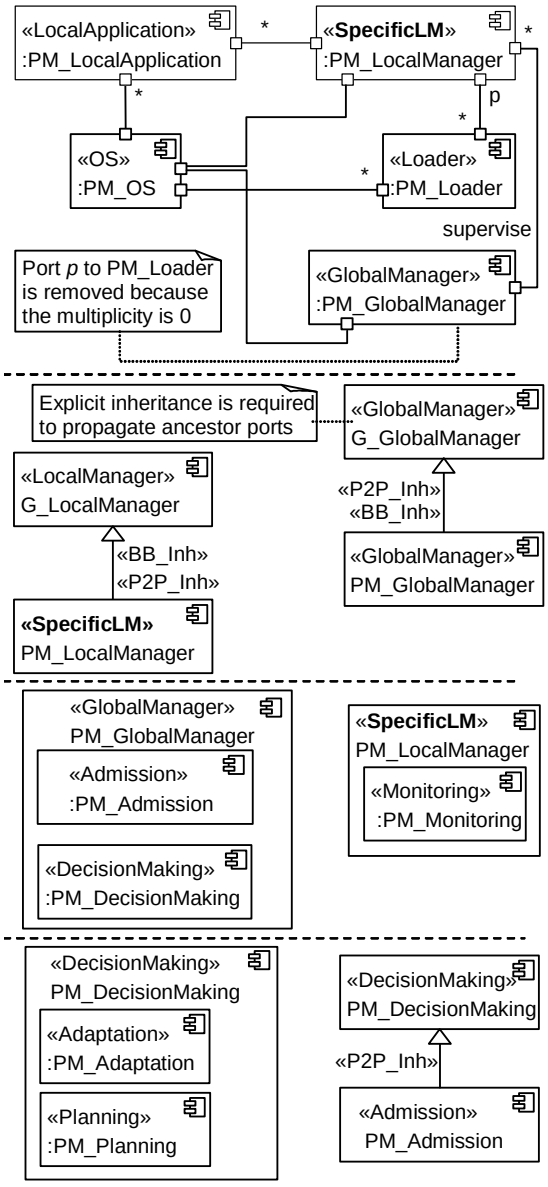

Figure **6**. The specific PMQDA model.

application behavior according to the choices carried out during the scheduling step. Since precise port inheritance is required, the *PM_Global-Manager* inherits ports from *G_GM* using *P2P_Inh.*

- *PM_LM*: on each node of the distributed system, it acts as an intermediary between *PM_Local-Application*s running on its node and the *PM_GM*.
- *PM_Loader*: during admission of a local application, the *PM_Loader* establishes a dialog with its *PM_LM* in order to obtain clearance to launch the *PM_LocalApplication*.
- *PM_LocalApplication*: it merges usual application business code with behavior adaptation services.
- *PM_Admission* has the same structure as *PM_DecisionMaking* because PMQDA executes the same operations for both admission and adaptation.

5. Related Works

Numerous works aim to aid the development of systems in a given domain. They tackle the problem at different abstraction levels, and their building blocks range from general to very specific. A coarse-grained classification could distinguish CBSE [4], and in a way SPL [2]. All these approaches aim to provide domain elements and composition operators to build systems in a given domain. Unlike these approaches, our framework enforces composition rules by limiting the licit expressions.

SPLs concerns are close to our works, but the aim is to represent products of market segments with possibly low coupling, maybe extending UML with a profile for variations, while our approach deals with concepts and semantics of a user domain with high cohesion. In our approach, the derivation of specific models is partially done manually because the variability is unknown a priori. In SPLs, variability is precisely described, which allows automating derivation from the choices of elements to be kept [2].

Component based development requires checking component assembly. Component models such as EJB [3] consider components as building blocks without defining a precise composition language. To overcome this weakness, some works propose to insert “glue” between components. For instance, [6] uses design by contract to connect components. Our approach applies at a high level of abstraction only: "glue" is no required since connections are implicitly predefined in the generic model.

6. Conclusion

This paper proposes a methodology to ease the development of a family of specific systems in a given domain. Our approach, based on standard UML, does not require specific training or tools besides UML. A generic coarse-grained model captures the relevant knowledge and commonality of the domain. The architect aided by an expert of the domain, maybe expert himself/herself, builds this model once. Then application developers derive specific models from the generic one while the framework ensures architecture consistency and soundness. Their burden is reduced since they share the architect’s domain knowledge and the resulting general architecture.

Our approach is illustrated through a case study in the domain of QoS Management System (PMQDA). This case study shows that our approach is suitable in a complex domain. The abstraction level of the description remains reasonably high to focus on commonality. However, to start from a framework is a great advantage compared with a development from scratch.

References

[1] F. Alhalabi, P. Vienne, M. Maranzana, and J-L. Sourrouille, Code Generation from the Description of QoS-Aware Applications. IEEE ICTTA’06, Vol 2: p. 3216-3221, 2006.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, and D. Muthig, Component-based Product Line Engineering with UML. ISBN 0-201-73791-4 506 pages, 2001.

[3] H-G. Min, J-Y. Lee, S-A. Kim, and S-D. Kim, An Effective Method to Design CBD Components in EJB. SERA'06, p. 49-56, 2006.

[4] I. Gorton, G. T. Heinemann, I. Crnkovic, and H-W. Schmidt, Component-Based Software Engineering, ISBN 978-3-540-35628-8 pages, 2006.

[5] OMG: Object Management Group, UML 2.1.1 Superstructure. document forma/07-02-03., 2007.

[6] H-W. Schmidt, I-D. Peake, J. Xie, I. Thomas, J-B. Kramer, A. Fay, and P. Bort, Modelling Predictable Component-Based Distributed Control Architectures. WORDS'03, p. 339-347, 2003.

[7] F. Siqueira, and V. Cahill, A QoS Architecture for Open Systems. IEEE ICDCS'00, p. 197-204, 2000.

[8] A. Solberg, J. Oldevik, and J-Ø Aagedal, A Framework for QoS-Aware Model Transformation, Using a Pattern-Based Approach. CoopIS, DOA, and ODBASE: p. 1190-1207, 2004.

[9] P. Vienne, and J-L Sourrouille, A framework for Dynamic Control of Behavior based on Learning. ACM ESEC/FSE WITSE'03, V 1, p. 44-47, 2003.

[10]P. Vienne, J-L. Sourrouille, and M. Maranzana, Modeling Distributed Applications for QoS Management. Software Engineering and Middleware, LNCS 3437, Springer Verlag: p. 170-184, 2005.