We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

3

Transformational Variability Modeling Approach to Configurable Business System Application

Marcel Fouda Ndjodo[1] and Amougou Ngoumou[2]
[1]*University of Yaounde I*
[2]*University of Douala*
Cameroon

1. Introduction

For more than ten years now, adaptation of software systems has become a major challenge for the software engineering community which has proposed different reference architectures and systematic approaches to address this challenging research topic. In the literature, the concept of adaptability is very broad and has many unclear and inconsistent definitions, with many closed-related types of non functional requirements such as flexibility, evolvability, transformability, reusability, robustness, configurability, etc. (see (Subramanian & Chung, 2001a) for a sample of representative definitions). This broad nature of adaptability makes it critical in practice since one of the problems in dealing with it is to give a clear and non ambiguous definition of adaptation and adaptability.

This contribution is based on the intuitive definition of N. Subramanian and L. Chung who consider that "adaptation means change in the system to accommodate change in its environment", and "adaptability is the extent to which a system adapts to change in its environment" (Subramanian & Chung, 2001a, 2001b). Since we also agree with them that, "software architecture should itself be adaptable for the final software system to be adaptable", this chapter, which is a continuation of our earlier work on business component semantics (BCS) extension and transformation of feature-oriented models (Fouda & Amougou, 2009, 2010), describes an engineering approach to support adaptation at architectural level of enterprise systems.

The term enterprise system (ES) came into fashion somewhat recently, but the concept behind it has been subject to academic discussion for a long time now and has evolved from an historic development in Business, Computer Science, and Information Systems. Over the last years, ES have evolved to comprehensive IT-supported business solutions that presumptively support and enhance organizations in their operations. Often times, ES refer to the larger set of all large organization-wide packaged applications with a process orientation. They have to be configured to suit the requirements of an organization (alignment with organizational requirements). In order to facilitate the alignment process, most ES solutions provide reference models that describe the functionality and structure of the system. But, research shows that reference models still are only of limited use to the configuration process. According to M. Rosemanna and W.M.P. van der Aalst (Rosemanna

& van der Aalst, 2003), this is mainly due to a lack of conceptual support for configuration in the underlying modeling language. Following this line of argumentation, they have defined a language and a process for the design and usage of *configurable reference models* in a model-driven approach towards ES configuration (Recker et al., 2006).

Computer-based systems built using ES are types of information systems (IS) which, according to Jeffrey L. Whitten and al. (Whitten et al., 2001), are intrinsically linked to an organization (also referred to, hereafter, as an enterprise) because "an IS is an arrangement of persons, data, procedures and technology tools which interact to insure the collection, processing, storage and the diffusion of essential information to the life of an organization". Since each enterprise must be adapted permanently to the evolution of its environment, information systems are therefore intrinsically dynamic due to the fact that any adaptation of an enterprise to its moving environment triggers an information system change whose aim is to adapt the IS to its new environment.

A change in an IS, is any observable mutation and/or evolution of one or many of its building blocks: people, data, processes or interface (Whitten et al., 2001; Zachman, 1987). We qualify as "major" any change that results in a larger deviation of the information system definition. While robustness (i.e. the ability to tolerate some deviations in the environment) can be added to a software system at the design or even implementation stage, adaptability (i.e. the ability to adapt to larger deviations in the environment) cannot be added at such late stages. Adaptability can be enforced only if it is considered at the architecture development stage (Subramanian & Chung, 2001b). We go further in that direction by considering the IS architecture development stage, i.e. the enterprise process modeling, should be the initial stage where adaptability is taken in consideration.

Enterprise modeling (Bernus, 2003; Fox & Gruninger, 1998; Lankhorst, 2004; Vernadat, 2002) is a critical building block to establishing an agile, robust enterprise architecture that keeps pace with the fast moving business. It is the first building block in aligning the IT initiative with the business objectives. The aim of an enterprise model, named here "*business system architecture*" (BSA), is to bring together business operations and IT. The BSA serves as the foundation, framework and guidepost necessary to understand the enterprise and its environment.

The aim of this chapter is to propose and illustrate a reusable business component-based approach to develop BSAs with an innate potential to evolve and adapt to new requirements. To be more concise, the chapter's contribution is two-fold: First, it introduces an adaptable BSA modeling framework covering an architecture description language which formalizes the FORM engineering assets (Kang et al., 2002, 2003; Lee et al., 2000) as reusable business components (Ramadour & Cauvet, 2002) which provide domain knowledge reusable during IS engineering and a generic abstract model for adaptable business architectures. Second a transformational (Rotenstreich, 1992) engineering process for adaptable BSA design and use is given.

Our approach is an integrated system product line approach, like PLUSS+ (Eriksson et al., 2010), in the sense that it extends traditional systems engineering by incorporating ideas from software product line (SPL) engineering. It integrates a product line method managing variability with a software engineering methodology. It is based on the traditional *domain engineering–application engineering* view of software product line development (van der

Linden et al., 2007; Weiss & Lai, 1999). SPL engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization (Pohl et al., 2005). Developing applications using platforms means to plan proactively for reuse, to build reusable parts, and to reuse what has been built for reuse. Building applications for mass customization means employing the concept of managed variability, i.e. the commonalities and differences in the applications (in terms of requirements, architecture, components, and test artifacts) of the product line have to be modeled.

The chapter is organized as follows: section 2 presents the modeling framework, section 3 then outlines the associated transformational engineering process, and section 4 concludes the chapter.

2. Modeling approach

This section outlines a generic conceptual framework for adaptable BSA. This framework is generic in the sense that it is not dependent on a specific modeling technique or method. However, a requirement for the application of our engineering process is that the engineering method used throughout the process must manage *variability* (Kang et al., 2010) in order to facilitate the derivation of model variants from the initial model. The main idea is to give *reusable business component* (Ramadour & Cauvet, 2002) semantics (rBCS) to the assets of a domain-specific architecture design method *M* by providing for each structure the context in which it can be reused. The resulting method, named *M/BCS* (read "M with business component semantics"), produces *M*-adaptable domain-specific architectures.

The model for adaptable BSA specification given in this section is based on a well established method in the product line engineering research community: the feature-oriented reuse method (FORM) (Kang et al., 1998).

2.1 Business component semantics

We use the model for conceptual business components specification of P. Ramadour and C. Cauvet (Ramadour & Cauvet, 2002) to define reusable domain-specific architecture assets. In this model, a business component integrates both reusable knowledge (object structures) and contextual knowledge guiding the reuse of the component. The context of a structure specifies specific requirements (set of constraints) accomplished by the structure and therefore indicates the suitable situation(s) in which a structure can be reused. The three levels of contextual constraints (business goals, business processes and business rules) considered by the model to specify conceptual business components clearly indicate that the conceptual business components of Ramadour and Cauvet are closely-related to enterprise process models assets (Bernus, 2003; Fox & Gruninger, 1998; Lankhorst, 2004; Vernadat, 2002). In this model, each business component has three constituents: a *name*, a *descriptor* and a *realization*:

- Descriptors explain when and why use components. A descriptor has an *intention* and a *context*. The intention is the expression of the generic modeling problem. The term "generic" here means that this problem does not refer to the context in which it is

supposed to be solved. The context of a business component details its main business activity (*domain*) in terms of atomic and non atomic sub activities (*process*) and explains the choice of one alternative and not the other (*common*, *optional*, *variabilities*).

- Realizations provide solutions to the modeling problems expressed in the descriptor sections. Solutions are the reusable part of the business component; they may have adaptation points that are parameters whose values are fixed at the reuse moment.

We use the formal language Z to formalize this business component model in order to allow a rigorous study of its properties. Due to space constraints, this model cannot be given here. Figure 1, gives the specification skeleton, where $\mathbb{F} A$ denotes the set of finite subsets of A and *Class* is the set of classes of objects (as used in the object-oriented terminology). The detailed specification is given in (Fouda & Amougou, 2009).

BusinessComponent = =[*name:* ***Text;*** *descriptor:****Descriptor;*** *realization:* ***Realization*** /]
Descriptor = = [*intention :* ***Intention*** ; *context :* ***Context*** /]
Intention = = [*action:* ***EnginneeringActivity****; target:* ***Interest*** /]
Context = = [*domain :* ***Domain*** ; *process :* $\mathbb{F}$ ***Context*** /]
Realization = = [*solution:* ***Solution****; adaptationpoints :* ***AdaptationPoints*** /]
Interest = ***Domain*** / ***BusinessObjects***
Domain = = [*action:* ***BusinessActivity****; target :* ***BusinessObjects*** /]
BusinessActivity == [*common:* ***BusinessActivity*** ;
optional: $\mathbb{F}$ ***BusinessActivity****;*
variabilities: $\mathbb{F}$ $\mathbb{F}$ ***BusinessActivity***
atomic: ***Boolean*** /]
BusinessObjects = = $\mathbb{F}$***Class***

Fig. 1. A formal specification of a business component

The types of solutions depend on the types of the business components. A solution can be a system decomposition, an activity organization or an object description, or anything else depending on the intention of the component. If this intention is to implement an activity of a product line engineering method (e.g. feature analysis), then the type of the solution is necessarily a kind of asset produced by the method (e.g. a feature model).

The BCS approach for adaptable business system architecture, which is advocated here, is a way to envelop assets of a product line engineering method with a domain knowledge layer. This layer, which indicates the purpose intended by the asset and the constraints it solve, provides the context in which it can be reused. It formally defines the extent to which the asset adapts to change in its environment. This additional layer is in fact an "adaptability information layer".

2.2 Business architecture description language

FORM/BCS architecture description language is specified through the description of its four main concepts: feature business components, subsystem architecture business components, process architecture business components, module business components and adaptable system architectures.

2.2.1 Feature business component

In FORM, a feature model of a domain gives the "intention" of that domain in terms of generic features which literally marks a distinct service, operation or function visible by users and application developers of the domain. FORM/BCS specifies a feature model of a domain as a business reusable component of that domain which captures the commonalities and differences of applications in that domain in terms of features (Figure 2).

The type *Feature* specifies business activities. A business activity is caused by an event which is applied to a target set of objects. Features have a generalization (in the sense of object-oriented analysis) and decomposition. A feature's decomposition gives the set of its common (sub) features which indicate reuse opportunity, the set of its optional (sub) features and the set of its groups of alternate (sub) features.

FeatureBusinessComponent = = *[name :**Name** ;*
*descriptor:**Descriptor**;*
*realization: **Realization***
fbc:FeatureBusinessComponent,
solution(realization(fbc)) ***Feature***
Adaptationpoints(realization(fbc)) *(Feature* × ***Feature)]***

Feature = =*[activity: **BusinessActivity** ;*
*objects: **BusinessObjectst** ;*
decomposition:[common: ***Feature****; optional:* ***Feature****;*
variabilities: ***Feature****]*
generalization: ***Feature*** *]*

Fig. 2. The feature business component model

2.2.2 Subsystem architecture business component

A subsystem architecture business component (Figure 3) describes a system in terms of abstract high level subsystems and the relationships between them. Graphically, the solution of a subsystem architecture business component is represented as a symmetric boolean matrix in which rows and columns represent the different subsystems of the business component and the values of the matrix indicate the existence of links between these subsystems.

SubSystemBusinessComponent = =
*[name: **Name**; descriptor: **Descriptor**; realization: **Realization***
ssbc: SubSystemBusinessComponent ,
solution(realization(ssbc)) ***SubsystemArchitecture***
adapationpoints(realization(ssbc)) ***(SubSystem*** × ***SubSystem)]***

SubsystemArchitecture = =
[subsystems: ***SubSystem*** *; Links:* ***(Subsystem SubSystem)*** *]*
SubSystem = ***Feature***

Fig. 3. The subsystem business component model

2.2.3 Process architecture business component

A process architecture business component represents a concurrency structure in terms of concurrent business activities to which functional elements are allocated; the deployment architecture shows an allocation of business activities to resources (Figure 4).

The type *ProcessArchitecture* specifies process architectures. A process architecture is a set of business activities (tasks) and classes of objects (data). Each business activity operates on a class of objects (data accesses) and business activities exchange messages between them in the form of actions call or with the environment (null).

ProcessBusinessComponent = =
[name: ***Name****; descriptor:****Descriptor****; realization:* ***Realization***
pbc:ProcessBusinessComponent,
solution(realization(pbc)) ***ProcessArchitecture***
adaptationpoints(realization(pbc))
*(****BusinessActivity****×* ***BusinessActivity****)]*

ProcessArchitecture = =
[tasks: ***BusinessActivity*** *;*
data : ***Class****;*
dataaccess: [name: ***Name****; access:* ***BusinessActivity****×****Class*** *]*
messages: [name: ***Name****; call:* ***(BusinessActivity*** *{null}) ×*
(BusinessActivity *{null})]]*

Fig. 4. The process business component model

2.2.4 Module business component

Module business components are refinements of process business architecture components. A module business component may be associated with a set of relevant features. Also, alternative features may be implemented as a template module or a higher level module with an interface that could hide all the different alternatives (Figure 5).

ModuleBusinessComponent = =
[name: ***Name*** *; descriptor:****Descriptor****; realization:* ***Realization***
mbc:ModuleBusinessComponent,
solution(realization(mbc)) ***Module***
adaptationpoints(realization(mbc)) *(****Module****×* ***Module****)]*
Module = = *[pseudonym :* ***Name*** *;*
parameters: ***Parameter****;*
description: [task: ***BusinessActivity****;*
included: ***Module****;*
external: ***Module*** *]*
specification: ***PseudoCode*** *]*

Fig. 5. The module business component model

A business module has a name, a list of parameters, a code in a pseudo language and a description which defines the task done by the module and the modules required for its execution, some of them are included in the module and some others are external.

2.2.5 Adaptable system architecture

FORM-based adaptable business architectures (Figure 6) have four perspectives or views:

- The *service view*, which is a set of feature business components (the functional perspectives), provides the solution for the analysis of the service provided by a business organization.
- The *system view*, which is a set of subsystem business components (the structural perspectives), gives the solution for the decomposition of a business organization.
- The *process view*, which is a set of process business components (the procedural perspectives), provides the solution for the description of the processes of a business organization.
- The *logical view*, which is a set of module business components (the logical perspectives), gives the solution for the specification of application modules associated to sub processes or tasks of a business organization.

The reusable business components defining adaptable system architectures can be stored in a database which can be requested using engineering by reuse operators developed by P. Ramadour (Ramadour, 2001): search, selection, adaptation, and composition operators.

AdaptableArchitecture = = *[service View:* ***FunctionalPerspective****;*
systemView : ***StructuralPerspective****;*
processView: ***ProceduralPerspective****;*
logicalView: ***LogicalPerspective*** *];*
FunctionalPerspective = = *FeatureBusinessComponent;*
StructuralPerspective = = *SubsystemBusinessComponent;*
ProceduralPerspective = = *ProcessBusinessComponent;*
LogicalPerspective= = *ModuleBusinessComponent*

Fig. 6. The Adaptable business architecture model

3. Adaptable architectures engineering

In this section, we describe a system engineering methodology for the production and use of adaptable business architectures. Systems engineering focuses on stakeholder needs and the required functionality early in the development cycle to synthesize an overall system design that captures those requirements from a total life-cycle perspective. Our approach is an integrated system product line approach, like PLUSS+ (Eriksson et al., 2010), in the sense that it extends traditional systems engineering by incorporating ideas from software product line engineering. It integrates a product line method managing variability with a software engineering methodology. It is based on the traditional *domain engineering–application engineering* view of software product line development (van der Linden et al., 2007; Weiss & Lai, 1999).

The purpose of domain engineering is to develop a product line's reusable core assets to provide a production capability for products (Northrop, 2002) and the purpose of application engineering is to generate new systems utilizing the assets developed by domain engineering. We refer to the domain engineering activities of our methodology as *horizontal engineering process* and the application engineering activities as *vertical engineering process* to

indicate that the purpose of application engineering is to refine business architectures at more low levels of abstraction (Figure 7).

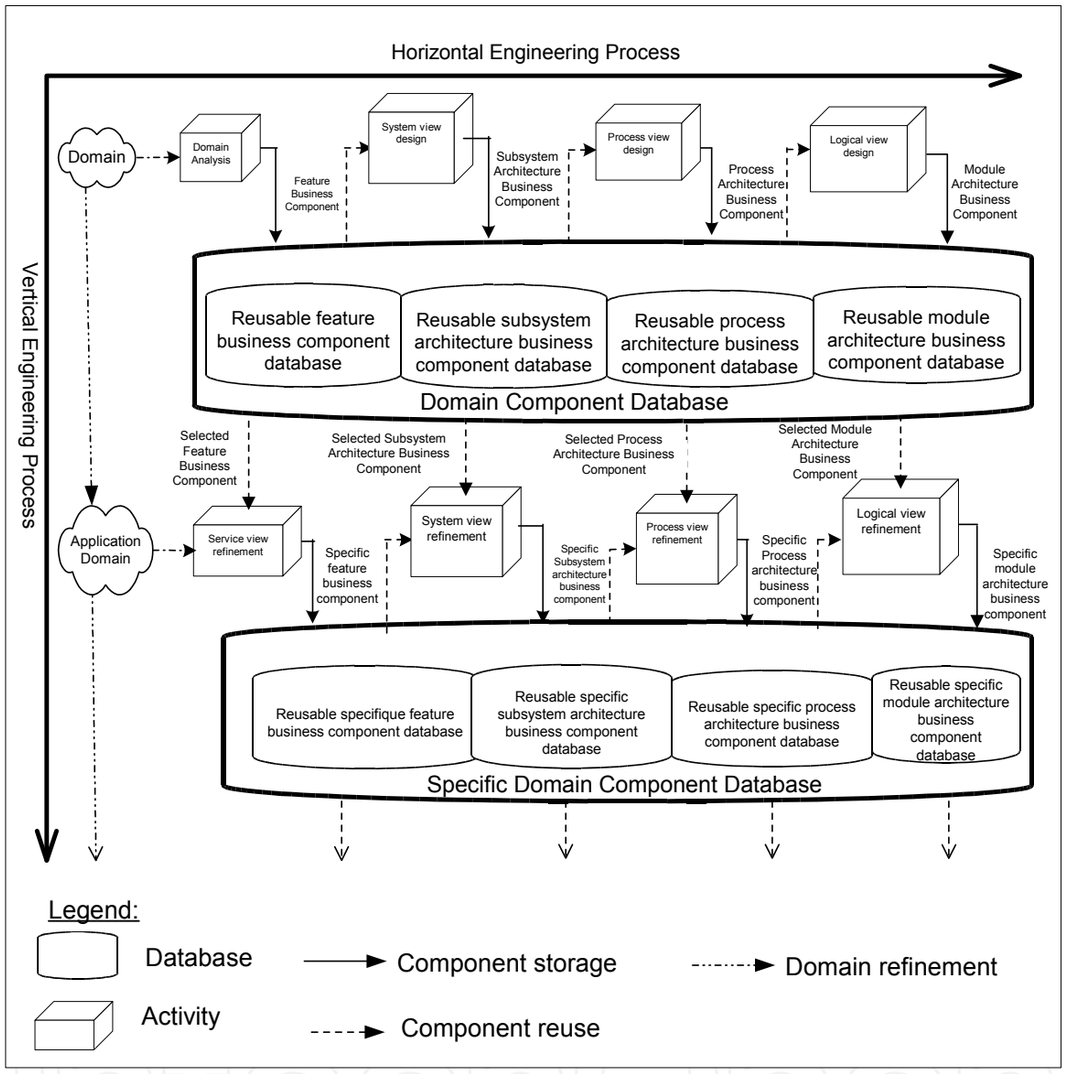


Fig. 7. The adaptable business architecture engineering process

The horizontal process, which corresponds to the "engineering for reuse" approach, gives the possibility to analyze a product line domain and develop adaptable architectures of that domain. These abstract reusable models can be refined ("engineering by reuse" approach) by the vertical engineering process in order to derive the specific business components of an application domain, which is to configure a suitable application from domain engineering.

3.1 Horizontal engineering process

The horizontal engineering process has been done in (Fouda & Amougou, 2010). It is a transformational method (Partsch, 1992; Rotenstreich, 1992), based on a set of provably semantics-preserving derivation rules called constructors.

The aim of a constructor is to transform, according to a formal rule called the construction schema, a kind of architectural artifact (the input type of the constructor) to another kind of architectural artifact (the output type of the constructor) by preserving all the system properties incorporated in any given input. The definition of a constructor therefore has three parts: the specification of the input type, the specification of the output type and, the specification of the construction rule which defines, through a construction schema and a set of semantics rules, how to build a semantics preserving output from a given input.

The horizontal engineering process has three constructors: the *system view constructor* which supports the system view design activity, the *process view constructor* which supports the process view design activity and the *logical view constructor* which support the logical view design activity. The service view of a system is the starting point of the process; this view is therefore obtained by applying any relevant requirement analysis technique.

3.1.1 System views design

The purpose of the system view design activity is to derive system views from service views. This activity is carried by the total function *SVC,* the system view constructor, whose purpose is to construct structural perspectives from functional perspectives of organizations. Figure 8, which intensively uses the adaptable business architecture model defined in section 2, specifies the system view constructor. In that figure, any text inside /* */ is a comment which explains the formal notation.

Input: A functional perspective *fp* of an organization.
Output: A structural perspective *SVC(fp)* of the organization.
Construction schema: *SVC (fp) = (SVC.name(fp), SVC.descriptor(fp), SVC.realization(fp))*
Semantics rules: 1. *SVC.name(fp) = text* /* *text* is any text used by the designer to name the reusable business component modeling the structural perspective */ 2. *SVC.descriptor(fp) = (SVC.intention(fp), SVC.context(fp))* 2.1. *SVC.intention(fp)* = <$(decompose)_{ACTION}$ $(domain(descriptor(fp))_{TARGET}$>, /* The intention of the structural perspective *SVC(fp)* is to decompose the business domain of *fp* */ 2.2. *SVC.context(fp) = context(descriptor(fp))* /* The context of *SVC (fp)* is the same as the context of *fp* */ 3. *SVC.realization(fp) = (SVC.solution(fp), SVC.adaptation_points(fp))* 3.1. *SVC.solution(fp) = (SVC.subsystems(fp), SVC.links(fp))* *3.1.1. SVC.subsystems(fp)* is the partition of the solution of the realization of *fp* defined as follows: *3.1.1.1.* $SVC.subsystems(fp) \subseteq \mathbb{F}\,\mathbb{F}\,Feature$ *3.1.1.2.* $\cup(F \in SVC.subsystems(fp)) = decomposition(solution(realization(fp)))$ *3.1.1.3.* $\forall F1, F2 \in SVC.subsystems(fp)\ ,\ F1 \neq F2 \Rightarrow F1 \cap F2 = \emptyset)$ *3.1.1.4.* $\forall F \in SVC.subsystems(fp),\ \forall f \in Feature\ ,\ \forall g \in Feature,$ $((f \in F \wedge g \in F) \Leftrightarrow$ $(\exists h \in F\ /\ (objects(f) \cap objects(h) \neq \emptyset) \wedge (objects(g) \cap objects(h) \neq \emptyset)))$.

3.1.2. $SVC.links(fp)= \{(F,G) \in SVC.subsystems(fp) \times SVC.subsystems(fp) / \exists (f,g) \in F \times G \bullet decomposition(f) \cap decomposition(g) \neq \emptyset\}$

3.2. $SVC.adaptation\_points(fp) = \{(ss, subsystemrealizations(ss)) \bullet ss \in SVC.subsystems(fp) \wedge ss \cap adaptation\_points(realization(fp)) \neq \emptyset\}$

3.2.1. $subsystemrealizations(ss) = \{ss':Subsystem \bullet \forall f \in ss, \exists ! g \in ss' / g \in featurerealizations(f) \wedge \forall g \in ss', \exists f \in ss / g \in featurerealizations(f)\}$

3.2.2. $featurerealizations(f) = \{ g:Feature \bullet common(f) \subseteq common(g) \wedge \forall V \in variabilities(f), (\exists ! h \in common(g) \bullet h \in V) \wedge optional(g) \subseteq optional(f)\}$

Fig. 8. The system view construction rule

3.1.2 Process view design

The purpose of the process view design activity is to derive process views from system views of organizations. This activity is carried by the total function *PVC*, the process view constructor, whose purpose is to construct procedural perspectives from structural perspectives of organizations. Figure 9 defines the process view constructor.

Input: A structural perspective *sp* of an organization.

Output: A set of procedural perspectives *PVC(sp)* of the organization.

Construction schema:

$PVC(sp) = \{(PVC.name(p), PVC.descriptor(p), PVC.realization(p)) \bullet p \in process(sp)\}$

Semantics rules:

1. $PVC.name(p) = text.$
   /* *text* is any text used by the designer to name the reusable business component modeling the procedural perspective */
2. $PVC.descriptor(p) = (PVC.intention(p), PVC.context(p))$
   - 2.1. $PVC.intention(p) = (describe)_{ACTION}\ (p)_{TARGET}$
     /* The intention of the process architecture built from $p \in process(sp)$ is to describe p */
   - 2.2. $PVC.context(p) = (domain(sp), \{p\})$
     /* The business activity of the process architecture constructed from the process $p \in process(sp)$ is the same as the main activity of *sp* and it has only one sub activity p */
3. $PVC.realization(p) = (PVC.solution(p), PVC.adaptation\_points(p))$
   - 3.1. $PVC.solution(p) = (PVC.tasks(p), PVC.datas(p), PVC.dataaccess(p), PVC.messages(p))$
     - 3.1.1. $PVC.tasks(p) = decomposition(action(domain(p)))$
       /* Tasks of the process architecture constructed from $p \in process(sp)$ are obtained by decomposing the action of the domain of p */
     - 3.1.2. $PVC.data(p) = target(domain(p))$
       /* The data of the process architecture constructed from $p \in process(sp)$ are the business objects of the target of the domain of p */
     - 3.1.3. $PVC.dataaccess(p) = \{(t, c) \in decomposition(action(domain(p))) \times target(domain(p)) / decomposition(t) \cap operations(c) \neq \emptyset\}$
       /* The task *t* of the process architecture constructed from $p \in process(sp)$ can operate on a class of object *c* only if some subtasks of *t* are operations

of the class *c* */

3.1.4. *PVC.messages(p) = {(t1, t2) ∈ (decomposition (action(domain(p))))² / decomposition (t1) ∩ decomposition (t2) ≠∅}*
/* Two tasks *t1* and t2 of the process architecture constructed from *p ∈ process (sp)* can exchange messages only if some subtasks of *t1* are subtasks *t2* */

3.1.5. *PVC.adaptation_points(p)) = {(t1, A) • t1 ∈ PVC.tasks(p) ∧ A = {t2 : BusinessActivity • common(t1) ⊆ common(t2) ∧(∀ V ∈ variabilities(t1), ∃ ! g ∈ common(t2) • g ∈ V) ∧ optional(t1) ⊆ optional(t2)} ∧ #A > 1}*
/* Adaptation points of the process architecture constructed from *p* in *process(sp)* are tasks of the process architecture for which we have more than one realization */

Fig. 9. The process view construction rule

3.1.3 Logical view design

The purpose of the logical view design activity is to derive logical views from process views. This activity is carried by the total function *LVC*, the logical view constructor, whose purpose is to derive logical perspectives from procedural perspectives of organizations. Figure 10 defines the logical view constructor.

Input: A procedural perspective *pp* of an organization.

Output: A set of logical perspectives *LVC(pp)* of the organization.

Construction schema:

LVC.descriptor(t), LVC.realization(t)) • t ∈ process(p), p ∈ process(pp)}

Semantics rules:

1. *LVC.name (t) = text*
   /* *text* is any text used by the designer to name the reusable business component modeling the logical perspective */
2. *LVC.descriptor(t) = (LVC.intention(t), LVC.context(t))*
   2.1. *LVC.intention(t) = <(implement)$_{ACTION}$ (t)$_{TARGET}$>*
   /* The intention of the module architecture built from a task *t ∈ process(p)* and *p ∈ process(pp)*, is to implement *t* */
   2.2. *LVC.context(t) = (domain(pp), {t})*
3. *LVC.realization(t) = (LVC.solution(t), LVC.adaptation_points(t))*
   3.1. *LVC.solution(t) = (LVC. pseudonym (t), LVC.parameters(t), LVC.task(t), LVC.included(t),), LVC.external(t), LVC.specification(t))*
   3.1.1. *LVC. pseudonym (t) = text'*
   /* *text'* is any text used by the designer to name the solution of the module architecture component constructed from the task *t ∈ process(p)*, for any *p ∈ process(pp)* */
   3.1.2. *LVC.parameters(t)* is a set of business objects of the domain of *t.*
   3.1.3. *LVC.task(t) = action(domain(t))*
   /* The task of the solution of the module architecture constructed from

the context *t* is the action of the domain of *t* */

3.1.4. *LVC.included(t) = {m:Module • task(m) ∈ decomposition(action (domain(t))) ∧ specification(m) ≠ "" }*
/* Modules included in the module architecture constructed from the context *t* are modules for which the task is a subtask of *t* and the specification is not empty*/

3.1.5. *LVC.external(t) = {m:Module • task(m) ∈ decomposition(action (domain(t))) ∧ specification(m) = "" }*
/* External modules in the module architecture constructed from the context *t* are module for which the task is a subtask of *t* and the specification is empty */

3.1.6. *LVC.specification(t) = {specification(m) • m ∈ LVC.included(t) ∪ LVC.required(t)}*
/* The specification of the module architecture constructed from the context *t* is the set of specifications of subtasks of t */

3.2. *LVC.adaptation_points(t) = {(m1, A) • m1∈ LVC.included(t) ∧ A = {m2 : Module • common(task(m1)) ⊆ common(task(m2)) ∧ (∀ V ∈ variabilities(task(m1)), ∃ ! g ∈ common(task(m2)) • g ∈ V) ∧ optional(task(m1)) ⊆ optional(task(2))} ∧ #A > 1}*
/* Adaptation points of the module architecture of a context *t* are modules included in the module architecture of *t* for which we have more than one realization */

Fig. 10. The logical view construction rule

3.2 Vertical engineering process

The purpose of the vertical engineering process is to generate new systems utilizing the assets developed by horizontal engineering. Its ultimate goal is to configure a suitable business application from domain engineering. It refines architectural assets of a domain to low level assets of an application domain of that domain. This engineering process is also a transformational method based on a set of provably semantics-preserving refinement rules called refiners. The process has four refiners: the service view refiner which supports the service view refinement activity, the system view refiner which supports the system view refinement activity, the process view refiner which support the process view refinement activity and the logical view refiner which supports the logical view refinement activity (see Figure 7).

3.2.1 Service view refinement

The purpose of the service view refinement activity is to derive a service view of an application domain of a domain from the service view of that domain. This activity is carried by a total function *FMR,* the functional model refiner, defines in Figure 12, which refines feature business components, i.e. functional perspectives, of a domain to specific business components of an application domain by using decompositions of non atomic services of input feature business components. A service view refinement is triggered by a decomposition of an abstract service of the service view. Any decomposition defines an application domain since it specifies a specific manner to implement the service. Decompositions define how abstract (common and optional) services of domains are implemented in application domains.

Figure 11 shows an example of a decomposition of an abstract service (career management of state personnel governed by the general status of the public service or the labor code: C_{11}) of the Cameroon civil servant management information system which has been used as a case study for the method (Atsa et al., 2010). In this decomposition, the abstract service C_{11} is implemented only by four common actions; one of these actions (the recruitment process: C_{111}) is itself decomposed in four optional actions.

common(C_{11}) = { C_{111} = (recruit = [{}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}])$_{ACTION}$(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)$_{TARGET}$(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)$_{DETAIL}$

C_{112} = (advance)$_{ACTION}$ (applications, civil servants governed by the general status or the labor code, decisions) $_{TARGET}$

C_{113} = (liquidate) $_{ACTION}$ (applications, civil servants governed by the general status or the labor code, decisions) $_{TARGET}$

C_{114} = (transfer) $_{ACTION}$ (civil servants governed by the general status or the labor code, decisions) $_{TARGET}$ }

optional(C_{11}) = {}

variabilities(C_{11}) = {}

Fig. 11. Decomposition of the non atomic service C_{11}

The refinement of a service view (see Figure 12 for the formal definition) replaces the decomposed service by its decomposition and integrates the new variability constraints in the new model.

Input: - A functional perspective *fp* of an organization - A decomposition *D* of an abstract service *s* of *fp*
Output: A specific functional perspective *FMR(fp,D)* of an application domain
Construction schema: *FMR (fp,D)= (FMR.name(fp,D), FMR.descriptor(fp,D), FMR.realization(fp,D))*
Semantics rules: ***1.*** *FMR.name(fp,D) = name(fp)* 2. *FMR.descriptor(fp,D) = (FMR.intention(fp,D), FMR.context(fp,D))* *2.1. FMR.intention(fp,D) = intention(descriptor(fp))* *2.2. FMR.context(fp,D) = (FMR.domain(fp,D), FMR.process(fp,D))* *2.2.1. FMR.domain(fp,D) = domain(context(descriptor(fp)))* *2.2.2. FMR.process(fp,D) = {(f, ∅) • f ∈ D}* **if** *process(context(descriptor(fp))) = ∅* /* In this case, *s* is the main activity of *fp* and *D* defines its sub activities */ **and** *FMR.process(fp,D) = process(context(descriptor(fp)))* **if** *process(context(descriptor(fp))) ≠ ∅.* /* A refinement of a sub activity of *fp* doesn't change the context of *fp* */

3. $FMR.realization(fp,D) = (FMR.solution(fp,D), FMR.adaptation\_points(fp,D))$

3.1. $FMR.solution(fp,D) = solution(realization(fp)) \bullet decomposition(s) = D$

/* D is the decomposition of the service s in the solution of the new business component */

3.2. $FMR.adaptation\_points(fp,D) = adaptationpoints(realization(fp))$ if $(optional(D) = \varnothing \wedge variabilities(D) = \varnothing)$

/* The set of adaptation points of fp doesn't change if D decomposes s only in common sub services */

and

$FMR.adaptation\_points(fp,D) = adaptationpoints(realization(fp)) \cup \{(s, variants(s,D))\}$ if $optional(D) \neq \varnothing \vee variabilities(D) \neq \varnothing$

/* A new adaptation point based on the variants of s induced by the decomposition D is created if D defines optional or variable subservices of s */

Fig. 12. The service view refinement rule

Figure 14 shows the result of the refinement of the service view of the civil servant management information system (Figure 13) based on the decomposition given in Figure 11.

Name *:Functional model of the Cameroonian civil servant management information system*

Descriptor *:*

Intention *:* $(Analyze)_{ACTION}((manage)_{ACTION}(State\ personals\ and\ salaries)_{TARGET})_{TARGET}$

Context *:*

Domain : $C = (manage)_{ACTION}(career, salaries, training, network, mail, system)_{TARGET}$

Process : $C_1 = (manage)_{ACTION}(civil\ servants\ career)_{TARGET}$

$C_2 = (manage)_{ACTION}(salaries)_{TARGET}$

$C_3 = (manage)_{ACTION}(training)_{TARGET}$

$C_4 = (manage)_{ACTION}(attributions)_{TARGET}$

$C_5 = (manage)_{ACTION}(mail)_{TARGET}$

/* *sub-process of* C_1 */

$C_{11} = (manage)_{ACTION}(decisions, personnels\ governed\ by\ the\ general\ status\ or\ the\ labor\ code)_{TARGET}$

$C_{12} = (manage)_{ACTION}(decisions, magistrates)_{TARGET}$

$C_{13} = (manage)_{ACTION}(decisions, university\ lecturers)_{TARGET}$

$C_{14} = (manage)_{ACTION}(decisions, police\ officers)_{TARGET}$

$C_{15} = (transfer)_{ACTION}(decisions)_{TARGET}$

/* *sub-process of* C_2 */

$C_{21} = (transfer)_{ACTION}(decisions)_{TARGET}$

$C_{22} = (calculate)_{ACTION}(salaries)_{TARGET}$

$C_{23} = (manage)_{ACTION}(workstation)_{TARGET}$

$C_{24} = (manage)_{ACTION}(profiles, workstations)_{TARGET}$

$C_{25} = (manage)_{ACTION}(connections, workstations)_{TARGET}$

/* *sub-process of* C_4 */

$C_{41} = (manage)_{ACTION}(workstations)_{TARGET}$

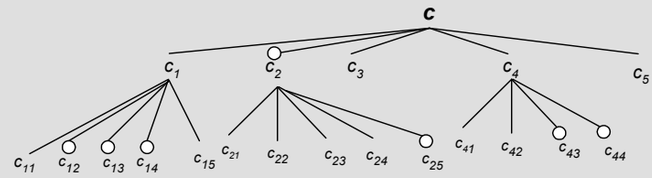
$C_{42} = (manage)_{ACTION}(profiles, workstations)_{TARGET}$

$C_{43} = (manage)_{ACTION}(connections, workstations)_{TARGET}$

$C_{44} = (manage)_{ACTION}(transactions, workstations)_{TARGET}$

Realization :

Solution :



Adaptation points : {

(c, {{c1, c3, c4, c5}, {c1, c2, c3, c4, c5}}),

(c1, {{c11, c15}, {c11, c15, c12}, {c11, c15, c13}, {c11, c15, c14}, {c11, c12, c13, c15}, {c11, c13, c14, c15}, {c11, c12, c14, c15} et {c11, c12, c13, c14, c15}}),

(c2, {{c21, c22, c23, c24}, {c21, c22, c23, c24, c25}}),

(c4, {{c41, c42}, {c41, c42, c43},{c41, c42, c44},{ c41, c42, c43, c44}})

}

Fig. 13. Service view of Cameroon civil servant management IS

. . .

/ sub-process of C_{11} */*

C_{111} *= (recruit = [{}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}])*$_{ACTION}$*(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)*$_{TARGET}$*(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)*$_{DETAIL}$

C_{112} *= (advance)*$_{ACTION}$ *(applications, civil servants governed by the general status or the labor code, decisions)* $_{TARGET}$

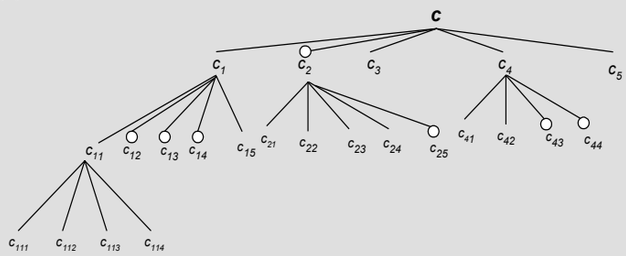
C_{113} *= (liquidate)* $_{ACTION}$ *(applications, civil servants governed by the general status or the labor code, decisions)* $_{TARGET}$

C_{114} *= (transfer)* $_{ACTION}$ *(civil servants governed by the general status or the labor code, decisions)* $_{TARGET}$

…

Realization :

Solution :



. . .

Fig. 14. A refined service view of Cameroon civil servant management IS

3.2.2 System view refinement

The purpose of the system view refinement activity is to derive a system view of an application domain of a domain from the system view of that domain. This activity is

carried by a total function *SMR*, the structural model refiner, defines in figure 16, which refines subsystem business components, i.e. structural perspectives, of a domain to specific business components of an application domain by using decompositions of non atomic services in subsystems of input subsystem business components. A system view refinement is triggered by a decomposition of an abstract service in a subsystem of the subsystem view. Any decomposition defines an application domain since it specifies a specific manner to implement the service. Decompositions define how abstract services of domains are implemented in application domains. Figure 15 shows an example of a decomposition of an abstract service (career management of state personnels: C_1) of the Cameroon civil servant management information system.

$common(C_1) = \{C_{11} = (manage)_{ACTION}(decisions, personnels\ governed\ by\ the\ general\ status\ or\ the\ labor\ code)_{TARGET}\}$

$optional(C_1) = \{ C_{12} = (manage)_{ACTION}(decisions, magistrates)_{TARGET},$
$C_{13} = (manage)_{ACTION}(decisions, univerties's\ lecturers)_{TARGET},$
$C_{14} = (manage)_{ACTION}(decisions, police\ officers)_{TARGET},$
$C_{15} = (transfer)_{ACTION}(decisions)_{TARGET}\}$
$variabilities(C_1) = \{\}$

Fig. 15. Decomposition of the non atomic service C_1

In this decomposition, the abstract service C_1 is implemented by one common action and four optional actions.

The refinement of a system view (see Figure 16 for the formal definition) replaces the decomposed service by its decomposition and integrates the new variability constraints in the new model.

Input:
- A structural perspective *sp* of an organization,
- A decomposition *D* of a non atomic service *c* in a subsystem *ss* of *sp*.

Output: A specific system view perspective *SMR(sp,D)* of an application domain

Construction schema:
SMR (sp,D)= (SMR .name(sp,D), SMR .descriptor(sp,D), SMR .realization(sp,D))

Semantics rules:

1. *SMR .name(sp,D) = name(sp)*

2. *SMR .descriptor(sp,D) = (SMR .intention(sp,D), SMR .context(sp,D))*
   - *2.1. SMR .intention(sp,D) = intention(descriptor(sp))*
   - *2.2. SMR .context(sp,D) = (SMR .domain(sp, D), SMR .process(sp,D))*
     - *2.2.1. SMR .domain(sp,D) = domain(context(descriptor(sp)))*
     - *2.2.2. SMR .process(sp,D) = {(f, ∅) • f ∈ D}*
       if *process(context(descriptor(sp))) = ∅*

       and

       SMR .process(sp,D) = process(context(descriptor(sp)))
       if *process(context(descriptor(sp))) ≠ ∅*

3. $SMR.realization(sp,D) = (SMR.solution(sp,D), SMR.adaptation\_points(sp,D))$

3.1. $SMR.solution(sp,D) = (SMR.subsystems(sp,D), SMR.links(sp,D))$

3.1.1. $SMR.subsystems(sp,D) = subsystems(solution(realization(sp)))$

/* A system view refinement of a system doesn't change its subsystems */

3.1.2. $SMR.links(sp,D) = links(solution(realization(sp))) \cup \{(ss, t), t \in subsystems(solution(realization(sp))) \bullet \exists g \in t, D \cap decomposition(g) \neq \varnothing\}$

/* A refinement of the subsystem *ss* triggered by the decomposition *D* creates a new link between *ss* and any other subsystem which is not disjoint with *D* */

3.3. $SMR.adaptation\_points(sp,D) = adaptationpoints(realization(sp))$ **if** $(optional(D) = \varnothing \wedge variabilities(D) = \varnothing)$

/* The set of adaptation points of *sp* doesn't change if *D* decomposes *c* only in common sub services */

and

$SMR.adaptation\_points(sp,D) = adaptationpoints(realization(sp)) \cup \{(ss, variants(ss,D))\}$ **if** $optional(D) \neq \varnothing \vee variabilities(D) \neq \varnothing$

/* A new adaptation point based on the variants of *ss* induced by the decomposition *D* is created if *D* defines optional or variable subservices of *c* */

Fig. 16. The system view refinement rule

Figure 18 shows the result of the refinement of the system view of the civil servant management information system (Figure 17) based on the decomposition of Figure 15.

Name *:Structural Model of the Cameroonian civil servant management information system*
Descriptor *:*
Intention *:*$(Decompose)_{ACTION}((manage)_{ACTION}(career, salaries, training, natwork, mail, system)_{TARGET})_{TARGET}$
Context *:*
Domain : $C = (manage)_{ACTION}(career, salaries, training, network, mail, system)_{TARGET}$
Process : $C_1 = (manage)_{ACTION}(career)_{TARGET}$
$C_2 = (manage)_{ACTION}(salaries)_{TARGET}$
$C_3 = (manage)_{ACTION}(training)_{TARGET}$
$C_4 = (manage)_{ACTION}(attributions)_{TARGET}$
$C_5 = (manage)_{ACTION}(mail)_{TARGET}$
/* sub-process of C2 */
$C_{21} = (transfer)_{ACTION}(decisions)_{TARGET}$
$C_{22} = (calculate)_{ACTION}(salaries)_{TARGET}$
$C_{23} = (manage)_{ACTION}(workstation)_{TARGET}$
$C_{24} = (manage)_{ACTION}(profiles, workstations)_{TARGET}$
$C_{25} = (manage)_{ACTION}(connections, workstations)_{TARGET}$
/* sub-process of C4 */
$C_{41} = (manage)_{ACTION}(workstations)_{TARGET}$
$C_{42} = (manage)_{ACTION}(profiles, workstations)_{TARGET}$
$C_{43} = (manage)_{ACTION}(connections, workstations)_{TARGET}$
$C_{44} = (manage)_{ACTION}(transactions, workstations)_{TARGET}$
…

Realization :
Solution:
Sub-systems: {$SS_1 = \{c_1\}$, $SS_2 = \{c_2\}$, $SS_3 = \{c_3\}$, $SS_4 = \{c_4\}$, $SS_5 = \{c_5\}$}
Links: {$SS_2 \leftrightarrow SS_5$}
Adaptation points: *{*
(SS_2, {{c_{21}, c_{22}, c_{23}, c_{24}}, {c_{21}, c_{22}, c_{23}, c_{24}, c_{25}}}),
(SS_4, {{c_{41}, c_{42}}, {c_{41}, c_{42}, c_{43}},{c_{41}, c_{42}, c_{44}},{ c_{41}, c_{42}, c_{43}, c_{44}}})
}

Fig. 17. A system view of Cameroon civil servant management IS

. . .
/* sous-processus de C_1 */
C_{11} = (manage)$_{ACTION}$(decisions, personals governed by the general status or the labor code)$_{TARGET}$
C_{12} = (manage)$_{ACTION}$(decisions, magistrates)$_{TARGET}$
C_{13} = (manage)$_{ACTION}$(decisions, universities's lecturers)$_{TARGET}$
C_{14} = (manage)$_{ACTION}$(decisions, police officers)$_{TARGET}$
C_{15} = (transfer)$_{ACTION}$(decisions)$_{TARGET}$
…
Liens: {$SS_1 \leftrightarrow SS_2$, $SS_2 \leftrightarrow SS_4$}
Adaptation points: *{*
(SS_1, {{c_{11}, c_{15}}, {c_{11}, c_{15}, c_{12}}, {c_{11}, c_{15}, c_{13}}, {c_{11}, c_{15}, c_{14}}, {c_{11}, c_{12}, c_{13}, c_{15}}, {c_{11}, c_{13}, c_{14}, c_{15}}, {c_{11}, c_{12}, c_{14}, c_{15}}, {c_{11}, c_{12}, c_{13}, c_{14}, c_{15}}}),
(SS_2, {{c_{21}, c_{22}, c_{23}, c_{24}}, {c_{21}, c_{22}, c_{23}, c_{24}, c_{25}}}),
(SS_4, {{c_{41}, c_{42}}, {c_{41}, c_{42}, c_{43}},{c_{41}, c_{42}, c_{44}},{ c_{41}, c_{42}, c_{43}, c_{44}}})
}

Fig. 18. A refined system view of Cameroon civil servant management IS

3.2.3 Process view refinement

The purpose of the process view refinement activity is to derive a process view of an application domain of a domain from the process view of that domain. This activity is carried by a total function *PMR,* the procedural model refiner, defines in Figure 20, which refines process business components of a domain to specific business components of an application domain by using decompositions of non atomic tasks of input process business components. A process view refinement is triggered by a decomposition of a task of the process view. Any decomposition defines an application domain since it indicates a specific manner to implement the task. Decompositions define how abstract tasks of domains are implemented in application domains. Figure 19 shows an example of a decomposition of an abstract task (recruitment of state personnels governed by the general status of the public service or the labor code: C_{111}, see Figure 11) of the Cameroon civil servant management information system.

recruit = [{}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}]]

Fig. 19. Decomposition of the non atomic service C_{111}

In this decomposition, the abstract task of C_{111} is implemented by four optional tasks.

The refinement of a process view (see Figure 20 for the formal definition) replaces the decomposed task by its decomposition and integrates the new variability constraints in the new model.

Input:
- A procedural perspective *pp* of an organization,
- A decomposition *D* of a non atomic task *t* of *pp*.

Output: A specific procedural perspective *PMR(pp,D)* of an application domain

Construction schema:
PMR (pp,D)= (PMR .name(pp,D), PMR .descriptor(pp,D), PMR .realization(pp,D))

Semantics rules:

1. *PMR .name(pp,D) = name(pp)*

2. *PMR .descriptor(pp,D) = descriptor(pp)*

3. *PMR .realization(pp, D) = (PMR .solution(pp,D), PMR . adaptation _points (pp,D))*

3.1. PMR .solution(pp, D) = (PMR .tasks(pp,D), PMR .datas(pp,D), PMR .datasaccess(pp,D), PMR .messages(pp,D))

3.1.1. PMR .tasks(pp,D) = tasks (solution(realization(pp)))

3.1.2. PMR .data(pp,D) = data(solution(realization(pp)))

3.1.3. PMR .dataaccess(pp,D) = dataaccess(solution(realization(pp))) \ {(t, c)• ∃ u ∈ tasks(solution(realization(pp))), (u, c) ∈ datasaccess(solution(realization(pp))) ∧ D ∩ decomposition (u) ≠ ∅}

3.1.4. PMR .messages(pp,D) = messages(solution(realization(pp))) ∪ {(t, u) • D ∩ decomposition (u) ≠ ∅}

3.2. PMR. Adaptation_ points (pp,D) = adaptationpoints (realization(pp)) **if** *(optional(D) = ∅ ∧ variabilities(D) = ∅)*

and

PMR .adaptation_points(pp,D) = adaptationpoints (realization(pbc)) ∪ {(t, variants(t,D))} **if** *optional(D) ≠ ∅ ∨ variabilities(D) ≠ ∅*

Fig. 20. The process view refinement rule

Figure 22 shows the result of the refinement of the process view of the civil servant management information system (Figure 21) based on the decomposition of Figure 19.

***Name**: Procedural model of career management of personals governed by the general status or the labor code in the Cameroonian civil servant management information system.*

***Descriptor** :*

***Intention** : (describe)$_{ACTION}$((manage = [{recruit, advance, liquidate, transfer}, {}, {}])$_{ACTION}$(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)$_{TARGET}$)$_{TARGET}$*

***Context** :*

Domain : C = (manage)$_{ACTION}$(career, salaries, training, network, mail, system)$_{TARGET}$

Process : C_{11} = (manage = [{recruit, advance, liquidate, transfer}, {},

{}])$_{ACTION}$(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)$_{TARGET}$
/ sub process of the process C_{11}*/*
C_{111} = (recruit)$_{ACTION}$(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)$_{TARGET}$(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)$_{DETAIL}$
C_{112} = (advance)$_{ACTION}$ (applications, civil servants governed by the general status or the labor code, decisions) $_{TARGET}$
C_{113} = (liquidate) $_{ACTION}$ (applications, civil servants governed by the general status or the labor code, decisions) $_{TARGET}$
…

Realization :
Solution :
tasks: decomposition(action(C_{11}))
data: target(C_{11})
dataaccess: {(t, c) ∈ decomposition(action(C_{11})) × target(C_{11}) / decomposition(t) ∩ operations(c) ≠∅}
messages: {(t1, t2) ∈ decomposition (action(C_{11})) × decomposition (action(C_{11})) / decomposition (t1) ∩ decomposition (t2) ≠∅}
Adaptation points: {}

Fig. 21. A prrocess view of Cameroon civil servant management IS

Process :
. . .
C_{111} = (recruit = [{}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}])$_{ACTION}$(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)$_{TARGET}$(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)$_{DETAIL}$
. . .

Realization :
Solution :
. . .
Adaptation points:
{(recruit, {{absorb by qualification}, {absorb by competitive examination}, {contractualize}, {engage}, { absorb by qualification, absorb by competitive examination }, { absorb by qualification, contractualize}, { absorb by qualification, engage}, {absorb by competitive examination, contractualize}, { absorb by competitive examination, engage}, {contractualize, engage},{ absorb by qualification, absorb by competitive examination, contractualize}, { absorb by qualification, absorb by competitive examination, engage}, { absorb by competitive examination, contractualize, engage}, { absorb by qualification, contractualize, engage}, { absorb by qualification, absorb by competitive examination, contractualize, engage}}) }

Fig. 22. A refined process view of Cameroon civil servant management IS

3.2.4 Logical view refinement

The purpose of the logical view refinement activity is to derive a logical view of an application domain of a domain from the logical view of that domain. This activity is carried by a total function *LMR,* the logical model refiner, defines in Figure 24, which refines module business components of a domain to specific business components of an application domain by using decompositions of non atomic business activities of input module business components.

A logical view refinement is triggered by a decomposition of a business activity of the logical view. Any decomposition defines an application domain since it specifies a specific manner to implement the business activity. Decompositions define how (common business activities, optional business activities, variability) abstract business activities of domains are implemented in application domains. Figure 23 shows example of a decomposition of an abstract business activity (absorption by qualification belonging to *optional(action* (C_{111})) of the Cameroon civil servant management information system.

absorb by qualification = [{}, {prepare = [{initiate, validate, append visa, remove validation, modify, delete}, {}, {}], sign}, {}]

Fig. 23. Decomposition of the non atomic business activity "absorption by qualification"

In this decomposition, the abstract business activity "absorption by qualification" is implemented by two optional business activities.

The refinement of a logical view (see Figure 24 for the formal definition) replaces the decomposed business activity by its decomposition and integrates the new variability constraints in the new model.

Input:
- A logical perspective *lp* of an organization,
- A decomposition *D* of a non atomic business activity *a* of *lp*.

Output: A specific logical view perspective *LMR(lp, D)* of an application domain

Construction schema:

LMR (lp,D)= (LMR .name(lp,D), LMR.descriptor(lp,D), LMR .realization(lp,D))

Semantics rules:

1. *LMR .name(lp,D) = name(lp)*
2. *LMR.descriptor(lp,D) = descriptor(lp)*
3. *LMR .realization(lp,D) = (LMR .solution(lp,D), LMR . adaptation _points (lp,D))*
   - 3.1. *LMR .solution(lp,D) = solution(realization(lp))*
   - 3.2. *LMR. Adaptation_ points (lp,D) = adaptationpoints (realization(lp))*
     if $(optional(D) = \varnothing \wedge variabilities(D) = \varnothing)$

and

LMR . adaptation _points (lp,D) = adaptationpoints (realization(lp)) ∪ {(m, variants(m))}
if $optional(D) \neq \varnothing \vee variabilities(D) \neq \varnothing$

Fig. 24. The logical view refinement rule

Figure 26 shows the result of the refinement of the logical view of the civil servant management information system (Figure 25) based on the decomposition of Figure 23.

Name: *Logical model of the recruitment of personals governed by the general status or the labor code in the Cameroonian civil servant management information system.*

Descriptor :

Intention : *(specify)* $_{ACTION}$ *((recruit = [{}, { absorb by qualification, absorb by competitive examination, contractualize, engage}, {}]])*$_{ACTION}$*(candidates, applications, competitive examination, civil servants governed by the general status or the labor code, decisions)*$_{TARGET}$*(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)*$_{DETAIL}$ $)_{TARGET}$

Context :

Domain : C = (manage)$_{ACTION}$*(career, salaries, training, network, mail, system)*$_{TARGET}$

Process: C_{111} *= (recruit = [{}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}]])*$_{ACTION}$*(candidates, applications, competitive examinations, civil servants governed by the general status or the labor code, decisions)*$_{TARGET}$*(If the candidate has succeeded a competitive examination or has obtained a diploma giving the right to absorption or the Presidency of the Republic has given the authorization)*$_{DETAIL}$

/ sub process of the process* C_{111} **/*

(absorb by qualification)$_{ACTION}$ *({decision, civil servants governed by the general status})* $_{TARGET}$

(absorb by competitive examination)$_{ACTION}$ *({decision, civil servant governed by the general status})* $_{TARGET}$

(contractualize)$_{ACTION}$ *({decision, civil servant governed by the labor code})* $_{TARGET}$

(engage)$_{ACTION}$ *({decision, civil servant governed by the labor code })* $_{TARGET}$

Realization :

Solution :

pseudonym : recruit;

parameters: {candidates, applications, competitive examination, civil servants, decisions};

task: < {}, {absorb by qualification, absorb by competitive examination, contractualize, engage}, {}>;

*include: 𝔽**Module**;*

*external: 𝔽**Module**]*

specification: ***PseudoCode***

Adaptation points :

{}

Fig. 25. A logical view of Cameroon civil servant management IS

```
                                    . . .
Realization :
   Solution :
        pseudonym : recruit;
        parameters: {candidates, applications, competitive examinations, civil servants, decisions};
  task: < {}, {absorb by qualification = [{}, {prepare = [{initiate, validate, append visa, remove
          validation, modify, delete}, {}, {}], sign}, {}], absorb by competitive examination,
          contractualize, engage}, {}>;
        include: 𝔽Module;
        external: 𝔽Module]
        specification: PseudoCode
   Adaptation points :
                    {(absorb by qualification,
                              {{prepare},
                              {sign},
                              {prepare, sign}})
                                                  }
```

Fig. 26. A refined logical view of Cameroon civil servant management IS

4. Conclusion

Until the last decade, variability could be defined either as an integral part of development artefacts or in a separate variability model. Concerning the first trend, many research contributions have suggested the integration of variability in traditional software development diagrams or models such as use case models (Oliviera et al., 2005), feature models (Kang et al., 2002; Bashroush et al., 2008), message sequence diagrams (Ziadi, 2004), class diagrams (Clauss, 2001; Ziadi, 2004), and activity diagrams (Razavian et al., 2008) to represent variability. Many others approaches have been proposed that suggest to define the variability information in a separate "orthogonal variability model" (OVM) which, according to Pohl et al. (2005), is a model that explicitly defines the variability of a software product line. In this chapter, we have presented an approach that tries to reconcile the two precedent orientations. The main idea is to envelop assets of a domain-specific design method managing variability with a domain knowledge layer which provides for each asset the context in which it can be reused. The domain knowledge layer is in fact an OVM that highlights the variability of the assets.

The resulting SPL engineering methodology has domain engineering activities, referred to as the horizontal engineering process, whose aim is to develop a product lines's reusable core assets to provide a production capability for products, and application engineering activities, referred to as the vertical engineering process, whose aim is to generate new systems utilizing the assets developed by horizontal engineering; The ultimate goal of the

vertical engineering process is therefore to configure a suitable business application from domain engineering.

5. References

Atsa, E.R., Fouda, N.M., Priso, E.N. & Abessolo, A.G. (2010). Improving the quality of service of a public service workflow based on ant theory: A case study in Cameroon. *The Electronic Journal of Information Systems in Developing Countries*, Vol.41(1),pp. 1-15.

Bashroush, R., Spence, I., Kilpatrick, P., Brown, T.J., Gillan, C. (2008). Multiple Views Models for Variability Management in Software Product Lines. *Second International Workshop on Variability Modeling of Software Intensive System (VaMoS'08)*, Essen, Germany.

Bernus, P. (2003). Enterprise Models For Enterprise Architecture and ISO9000:2000. *Annual Reviews in Control,* 27, pp. 211-220.

Clauss, M. (2001). Generic Modelling Using UML Extensions for Variability. *Workshop on Domain Specific Visual Languages*, pp 11-18.

Eriksson, M., Börstler, J. & Borg, K. (2010). A Systems Product Line Approach, In: *Applied Software Product Line Engineering,* Kyo C. Kang, Vijayan Sugumaran, Sooyong Park, pp. 109-139, Crc Press, Taylor & Francis Group, ISBN 978-1-4200-6841-2, Boca Raton.

Fouda, N.M. & Amougou, N. (2009). The Feature Oriented Reuse Method with Business Component Semantics. *International Journal of Computer Science and Applications*, Vol. 6, No. 4, pp 63-83.

Fouda, N.M. & Amougou, N. (2010). Product Lines' Feature-Oriented Engineering for Reuse: A Formal Approach. *International Journal of Computer Science Issues*, Vol. 7, Issue 5, pp 382-393.

Fox, M.S. & Gruninger, M. (1998). Enterprise Modeling, *AI Magazine Fall 1998*, pp. 109-121.

Kang, K.C., Lee, K., Lee, J. & Kim, S. (2003). Feature-Oriented Product Line Software Engineering: Principles and Guidelines. *Domain Oriented Systems Development: Perspectives and Practices*, K. Itoh et al., eds., pp. 29-46.

Kang, K.C., Sugumaran, V. & Park, S. (2010). Software Product Line Engineering: Overview and Future Direction, In: *Applied Software Product Line Engineering,* Kyo C. Kang, Vijayan Sugumaran, Sooyong Park, pp. 3-14, Crc Press, Taylor & Francis Group, ISBN 978-1-4200-6841-2, Boca Raton.

Kang, K.C., Kim, S., Shin, E. & Huh, M. (1998). "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, Vol. 5, pp. 143-168.

Kang, K.C., Lee, J. & Donohoe, P. (2002). Feature-Oriented Product Line Engineering. *IEEE Software*, Vol. 19, no. 4, pp. 58-65.

Lankhorst, M. (2004). Enterprise Architecture Modeling – The Issue of Integration. *Advanced Engineering Informatics,* 18, pp 205-216.

Lee, K., Kang, K.C. & Choi, W. (2000). Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse. *Software-Practice and Experience*, 30, pp.1025-1046.

Northrop, L. (2002). SEI's software product line tenets. *IEEE Software* 19 (4): 32–40.

Oliviera, E.A., Gimenes, I., Huzita, E., Maldonado, J.C. (2005). Variability Management Process for Software Product Lines. *In Proc. of CASCON 2005*, Toronto, Canada.

Partsch, H.A. (1990). Specification and transformation of programs: A Formal Approach To Software Development. *Springer-Verlag New York, Inc.* New York, USA.

Pohl, K., Bockle, G., Linden, F.V.D., (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. *Springer-Verlag Berlin,* Heidelberg.

Ramadour, P. & Cauvet, C. (2002). Approach and Model for Business Components Specification, *Proceeding of the 13th International Conference on Database and Expert Systems Applications, Lecture Notes In Computer Science*; Vol. 2453, pp 628-637.

Ramadour, P. (2001). Modèles et langage pour la conception et la manipulation de composants réutilisables de domaine. *PhD thesis*, Université d'Aix-Marseille III, Marseille, France.

Razavian, M., Khosravi, R. (2008). Modeling Variability in Business Process Models Using UML. *Fifth International Conference on Information Technology: New Generations*, Las Vegas, USA.

Recker, J., Mendling, J., van der Aalst, W. & Rosemann, M. (2006). Model-driven Enterprise Systems Configuration, *Proceeding of the 18th International Conference of Advanced Information Systems Engineering, CAiSE 2006*, Luxembourg, Lectures Notes in Computer Science, Vol. 4001, pp. 369-383, Springer.

Rosemanna, M. & van der Aalst, W.M.P. (2003). A Configurable Reference Modelling Language. *QUT Technical Report, FIT-TR-2003-05*, Queensland University of Technology, Brisbane, Australia.

Rotenstreich, S. (1992). Transformational Approach to Software Design. *Information Software Technology*, Volume 34, Issue 2, pp 106-116.

Subramanian, N. & Chung, L. (2001a). Software Architecture Adaptability: An NFR Approach, *Proceeding of the 4th International Workshop on Principles of Software Evolution*, New-York, USA, ACM Digital Library.

Subramanian, N. & Chung, L. (2001b). Metrics for Software Adaptability, *Software Quality Management Conference*.

van der Linden, F., Schmid, K. & Rommes, E. (2007). *Software product lines in action: The best industrial practice in product line engineering*, Berlin: Springer-Verlag.

Vernadat, F.P. (2002). Enterprise Modeling and Integration (EMI): Current Status and Research Perspectives. *Annual Reviews in Control*, 26, pp. 15-25.

Weiss, D.M. & Lai, C.T.R. (1999). *Software product line engineering: A family-based software development process*, Boston: Addison-Wesley Longman.

Whitten, J.L., Bentley, L.D. & Dittman, K.C. (2001). *Systems Analysis and Design Methods*, 5th ed., McGraw-Hill Companies, Inc.

Zachman, J. (1987). A Framework for Information System Architecture. *IBM Systems Journal*, Vol. 26(3).

Ziadi, T. (2004). Manipulation de Lignes de Produits en UML. *Phd Thesis*, Université de Rennes I, France.

Software Product Line - Advanced Topic

Edited by Dr Abdelrahman Elfaki

ISBN 978-953-51-0436-0
Hard cover, 122 pages
Publisher InTech
Published online 04, April, 2012
Published in print edition April, 2012

The Software Product Line (SPL) is an emerging methodology for developing software products. Currently, there are two hot issues in the SPL: modelling and the analysis of the SPL. Variability modelling techniques have been developed to assist engineers in dealing with the complications of variability management. The principal goal of modelling variability techniques is to configure a successful software product by managing variability in domain-engineering. In other words, a good method for modelling variability is a prerequisite for a successful SPL. On the other hand, analysis of the SPL aids the extraction of useful information from the SPL and provides a control and planning strategy mechanism for engineers or experts. In addition, the analysis of the SPL provides a clear view for users. Moreover, it ensures the accuracy of the SPL. This book presents new techniques for modelling and new methods for SPL analysis.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Marcel Fouda Ndjodo and Amougou Ngoumou (2012). Transformational Variability Modeling Approach to Configurable Business System Application, Software Product Line - Advanced Topic, Dr Abdelrahman Elfaki (Ed.), ISBN: 978-953-51-0436-0, InTech, Available from: http://www.intechopen.com/books/software-product-line-advanced-topic/transformational-variability-modeling-approach-to-configurable-business-system-application

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen