A Command Oriented Derivation Approach with Product-Specific Architecture Optimization

Amougou Ngoumou\(^1\) and Marcel Fouda Ndjodo\(^2\)

\(^1\)Department of Computer Science, College of Technology, University of Douala, Cameroon
ngoumoua@yahoo.fr

\(^2\)Department of Computer Science, Higher Teachers Training College, University of Yaounde I, Cameroon
marcel.fouda@dite-ens.cm

Abstract

FROM/BCS is a promising product line engineering method which has the advantage of integrating variability in software development diagrams or models and to explicitly give variants of variation points. This approach has been proposed to allow transposition of industrial chain production to software world. However, to come closer to the real reduction of cost and risk and taking into consideration the evolution of software requirements, the method have to be improved since it does not give a way to set characteristics so as to derive a specific product easily. To address this issue in this paper, through the concept of command, we give a modeling approach that allows requirements fixation. In the new approach named CODA (Command Oriented Derivation Approach), given a requirement variation, the specific product architecture is derived fluently from the generic architecture and the command specification.

Keywords: Product line engineering, command specification, information system, business components, domain analysis, variability

1. Introduction

In software products lines approach, which is a transposition of industrial chain production to software world [1], the idea is to minimize software construction cost in a particular application domain, by developing each software not separately, but by designing them from reusable elements.

The first difficulty in software product line is in the design of architecture allowing the building of many systems [1]. Therefore, the management of variation—the so-called variability management in product line engineering—has become more important [2].

Until the last decade, variability could be defined either as an integral part of development artifacts (amalgamated approaches) or in a separate variability model (separated approaches). 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 [3], feature models [4, 5], message sequence diagrams [6], class diagrams [7, 6], and activity diagrams [8] to represent variability. Many other approaches have been proposed that suggest the definition of variability information in a separate model, among them, “orthogonal variability model” (OVM) which, according to Pohl et al. [9], is a model that explicitly defines the variability of a software product line, has been proposed as well as “decisions” which are variation points that constrain other variation points and that can explicitly be related to a domain concept. Hence, decisions are meant to structure and document variation points, especially their inter-relationships and dependencies. A decision defines a question that captures the essence of the related variability in the sense that answering this question corresponds to resolving the decision,
which is a variation point. A simple decision is a variation point identifying only one location in a generic artifact. It is a decision that is directly related to a generic artifact and that does not constrain other variation points [10, 11].

In [12-15], we presented a promising method called FORM/BCS, a hybrid approach that tries to reconcile the two precedent orientations. Up till now, the approach has been characterized by a lack of supports for resolving the variability and obtaining product models. It is only useful for product line architecture description. In this paper, we want to show how a fluent derivation of a specific architecture from an adaptable architecture of FORM/BCS, renamed CODA (Command Oriented Derivation Approach), and a command specification can be done. For that, it is important to specify commands which represent entities in which users express their needs. The enrichment done represents an important step for software product line architectures management.

The rest of the paper is organized as follow. In section 2 we present FROM/BCS software product line core assets, in section 3 we propose a derivation approach of specific products and discuss related work and in section 4 we draw conclusions and future research issues.

2. The Assets of FORM/BCS

This section describes the five main assets of FORM/BCS: feature business components (section 2.2), subsystem architecture business components (section 2.3), process architecture business components (section 2.4), module business components (section 2.5) and adaptable system architectures (section 2.6). Since the four basic assets of FORM/BCS are reusable business components, section 2.1 presents the conceptual modeling framework of reusable business components which is used to define the FORM/BCS basic assets.

2.1 The Formal Model of FORM/BCS basic Assets

We use the model for business components specification defined by Ramadour and Cauvet [16] to specify the basic assets of FORM/BCS. In this model, each asset has three constituents: a name, a descriptor and a realization. The descriptor presents the conceptual modeling problem to be solved in a particular context. This problem can be the decomposition of a system, an activity organization or an object description. Goals, activities and objects concerned are carried on an application field and/or an engineering method. The realization section of a reusable component provides a solution to the modeling problem expressed in the descriptor section of the component. This solution may have adaptation points that are parameters whose values are fixed at the reuse moment. The following Z schema formalizes reusable business components:

<table>
<thead>
<tr>
<th>Table 1. Specification of Reusable Business Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReusableBusinessComponent = = [name: Text; descriptor: Descriptor; realization: Realization ]</td>
</tr>
</tbody>
</table>

2.1.1 Descriptors: The descriptor of a reusable business component gives an answer to the following question: “when to and why use this component?” 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 reusable business component is the knowledge which explains the choice of one alternative and not the other. Formally, descriptors are defined by the following schemas:
Table 2. Specification of Business Component Descriptors

| Descriptor = [intention : Intention; context : Context] |
| Intention = [action: EngineeringActivity; target: Interest] |
| Context = [domain : Domain; process : $\mathcal{F}$ Context] |
| EngineeringActivity = AnalysisActivity | DesignActivity |
| AnalysisActivity = \{analyze, \ldots\} |
| DesignActivity = \{design, decompose, describe, specify, \ldots\} |

The detailed specification is given in [12]. For the intelligibility of this paper, we give an important type used in the above specification below: Interest.

The engineering activity defined in the intention (hereafter referred to as the action of the intention) of a reusable business component acts on a “target” which can be a business domain or a set of business objects. Here are two examples of intentions formalized in FORM/BCS:

- (analyze)$\rightarrow$Target(civil servant management system)
- (describe)$\rightarrow$Target(civil servant recruitment application)

Interests of engineering activities are specified by the following schemas in which $\mathcal{F}A$ denotes the set of finite subsets of $A$.

Table 3. Specification of Intention’s Interests

| Interest = Domain | BusinessObjects |
| Domain = [action: BusinessActivity; target: BusinessObjects; precision : Precision] |
| BusinessObjects = $\mathcal{F}$ Class |
| Class = [name: Name; attributes : $\mathcal{F}$ Attribute; operations : F BusinessActivity] |
| Precision |
| Name |
| Attribute |

A business activity is a set of (sub) business activities divided into three disjoint categories: the set of common (sub) business activities of the activity which indicate reuse opportunity (the commonality of the business activity), the set of optional (sub) business activities of the activity (the options of the business activity) and, the set of groups of alternate (sub) business activities of the activity (the switch ability of the business activity). The capacity to have options and the switch ability for a business activity constitute its variability. A business activity is primitive (i.e., it cannot be decomposed) or not.

Table 4. Specification of Business Activities

| BusinessActivity = [name: Name; common: $\mathcal{F}$ BusinessActivity; optional: $\mathcal{F}$ BusinessActivity; switchable: $\mathcal{F}$ $\mathcal{F}$ BusinessActivity; primitive: Logic] |

When the context is clear, given business $a$, we write:

decomposition$(a)$ for $\text{common}(a) \cup \text{optional}(a) \cup (\cup(S \in \text{switchable}(a)))$
2.1.2 Realizations: The realization section of a reusable component provides a solution to the modeling problem expressed in the descriptor section of the component. It is a conceptual diagram or a fragment of an engineering method expressed in the form of a system decomposition, an activity organization or an object description. The goals, the activities and the objects figuring in the realization section concern the application field (product fragment) or the engineering process (process fragment).

The solution, which is the reusable part of the component, provides a product or a process fragment. The types of solutions depend on the type of reusable business component i.e. a solution of a feature business component (respectively a reference business component) is a feature (respectively reference business architecture). This solution may have adaptation points with values fixed at the reuse moment. Adaptation points enable the introduction of parameters in the solutions provided by reusable components. Those parameters are values or domains of values of elements of the solution.

Table 5. Specification of Business Component Realizations

| Realization = = [solution: Solution; adaptationpoints : AdaptationPoints ] |

Solutions and adaptation points of the different types of reusable components (feature business components, subsystem architecture business components, process architecture business components, and module business components) are defined in the corresponding subsections below.

2.2 Feature Business Components

In FORM, a feature model of a domain gives the “intention” of that domain in terms of generic features which literally mark 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. Feature business components are used to support both the engineering of reusable domain artifacts and the development of applications using domain artifacts.

Table 6. Specification of Feature Business Components

| FeatureBusinessComponent = = [name :Name ; descriptor:Descriptor; realization: Realization ] |

∀ fbc:FeatureBusinessComponent, 
(solution(realization(fbc)) ∈ Feature ∧ Adaptationpoints(realization(fbc)) ∈ ℱ (Feature × ℱ Feature))
Feature = ={activity: BusinessActivity ; objects: BusinessObjects ; decomposition:{common: ℱ Feature; optional: ℱ Feature; switchable: ℱ ℱ Feature} 
generalization: ℱ Feature | ]

In the above schemas, 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 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.
A reusable feature business component $fbc$ is well formed if it satisfies the following four characteristic properties which require that the realization section of a feature business component correspond to the intention of that business component:

(fbc1) The solution given in the realization section of $fbc$ is a solution of the intended contextual business activity of $fbc$:

$$\text{action(domain(context(descriptor(fbc))))} = \text{activity(solution(realization(fbc))))}$$

(fbc2) The target of the intended contextual business activity of $fbc$ is exactly the set of objects collaborating in the business activity of the solution given in the realization section of $fbc$:

$$\text{target(domain(context(descriptor(fbc))))} = \text{objects(solution(realization(fbc))))}$$

(fbc3) Any requirement expressed in the form of a business process in the intended contextual business activity of $fbc$ has a unique solution in the realization section of $fbc$:

$$\forall p \in \text{process(context(descriptor(fbc))), } \exists ! g \in \text{decomposition(solution(realization(fbc)))}, \bullet \text{ activity}(g) = \text{action(domain(p))} \land \text{objects}(g) = \text{target(domain(p))}$$

(fbc4) Any solution in the decomposition of the solution of the realization of $fbc$ expressed in the form of a feature solves a unique requirement expressed in the form of a business process in the intended contextual business activity of $fbc$:

$$\forall g \in \text{decomposition(solution(realization(fbc))), } \exists ! p \in \text{process(context(descriptor(fbc)))}, \bullet \text{ activity}(g) = \text{action(domain(p))} \land \text{objects}(g) = \text{target(domain(p))}$$

2.3 Subsystem Architecture Business Components

A subsystem architecture business component is a reusable business component which describes a system in terms of abstract high level subsystems and the relationships between them.

Table 7. Specification of Subsystem Architecture Business Components

<table>
<thead>
<tr>
<th>SubSystemBusinessComponent</th>
<th>=</th>
<th>[name: Name; descriptor: Descriptor; realization: Realization]</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall ssbc: SubSystemBusinessComponent , ( solution(realization(ssbc)) \in SubsystemArchitecture \land adaptationpoints(realization(ssbc)) \in \mathcal{F} (SubSystem \times \mathcal{F} SubSystem) )</td>
<td>]</td>
<td></td>
</tr>
<tr>
<td>SubsystemArchitecture</td>
<td>=</td>
<td>[subsystems: SubSystem ; links: \mathcal{F} (SubSystem \times SubSystem) ]</td>
</tr>
<tr>
<td>SubSystem</td>
<td>==</td>
<td>[name: Name; features: \mathcal{F} Feature]</td>
</tr>
</tbody>
</table>

A reusable subsystem business component $sbc$ is well formed if it satisfies two characteristic properties which require that the realization section of a subsystem business component corresponds to the intention of that business component. These are:

(sbc1) Any requirement expressed in the form of a business process in the intended contextual business activity of $sbc$ has a unique solution in the realization section of $sbc$:

$$\forall p \in \text{process(context(descriptor(sbc))}, \exists ! ss \in \text{subsystems(solution(realization(sbc))}, \exists ! f \in ss, \bullet$$
activity(f) = action(domain(p)) \land 
\text{objects}(f) = target(domain(p))

(sbc2) Any link in the solution of the realization section of \textit{sbc} resolves requirements expressed as collaborations between business processes in the intended contextual business activity of \textit{sbc}:
\[
\forall (ss1, ss2) \in \text{links(solution(realization(sbc)))), \exists (f1, f2) \in ss1 \times ss2 \bullet 
\text{decomposition}(f1) \cap \text{decomposition}(f2) \neq \emptyset
\]

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.

2.4 Process Architecture Business Components

A process architecture business component is a reusable business component which 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.

Table 8. Specification of Process Architecture Business Components

| ProcessBusinessComponent | = = [name: Name; descriptor: Descriptor; 
| realization: Realization |
| \forall pbc:ProcessBusinessComponent, 
| (solution(realization(pbc)) \in \text{ProcessArchitecture} \land 
| \text{adaptationpoints}(realization(pbc)) \in 
| \mathbb{F} (\text{BusinessActivity} \times \mathbb{F} \text{BusinessActivity})] |
| \text{ProcessArchitecture} | = = \text{[tasks: } \mathbb{F} \text{BusinessActivity ; 
| \text{datas} : \mathbb{F} \text{Class; 
| \text{messages} : } \mathbb{F} \text{[name: Name; 
| \text{call: } (\text{BusinessActivity} \cup \{\text{null}\}) \times 
| (\text{BusinessActivity} \cup \{\text{null}\})] | | ] |

On the above schemas, the type \text{ProcessArchitecture} specifies process architectures. A process architecture is a set of business activities and objects (data). The business activities operate on data and exchange messages between them (in the form of actions call) or with the environment (null).

A reusable process architecture business component \textit{pbc} is well formed if it satisfies the following three characteristic properties which require that the realization section of a process business component correspond to the intention of that business component:

\textit{(pbc1)} Any requirement expressed as a business process in the intended contextual business activity of \textit{pbc} has a unique solution in the realization section of \textit{pbc}:
\[
\forall p \in \text{process(context(descriptor(pbc)))}, 
\forall a \in \text{decomposition(action(domain(p)))} 
(\exists ! t \in \text{tasks(solution(realization(pbc)))} \bullet 
\text{t} = a) \land (\text{target(domain(p))} \subseteq 
\text{datas(solution(realization(pbc)))}) \land (\text{dataaccess(solution(realisation(pbc)))}) \leftrightarrow ((a \in 
\text{decomposition(action(domain(p)))} \cap \text{operations(d)}) \neq 
\emptyset))
\]
where decomposition(a) is written for common(a) \cup \text{optional(a) } \cup (\cup \text{switchable(a)})

\textit{(pbc2)} Messages are sent only between tasks having common actions:
∀ p ∈ process (context(descriptor(pbc))), (((t1, t2) ∈ decomposition action(domain(p))) × decomposition (action(domain(p)))) \land (decomposition(t1) \cap decomposition(t2) ≠ \emptyset)) ⇔ ((t1, t2) ∈ messages (solution(realisation(pbc))))

2.5 Module Business Components

Module business architecture components are refinements of process business architecture components. A module 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. The following schemas formally define module business components:

**Table 9. Specification of Module Business Components**

| ModuleBusinessComponent = [name: Name; descriptor: Descriptor; realization: Realization] |
| ∀ mbc:ModuleBusinessComponent, (solution(realization(mbc)) ∈ Module \land adaptationpoints(realization(mbc)) ∈ F (Module × F Module)) |


| Parameter |
| PseudoCode |

In the above schemas, a 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 are external. A module business component mbc is well formed if it satisfies the following characteristic properties of module business components, which are:

(mbc1) The set of requirements expressed as a business process in the intended contextual business activity of mbc is a singleton and has a solution in the realization section of mbc.

]*(# process(context(descriptor(mbc))) = 1) \land ((p ∈ process(context(descriptor(mbc)))) ⇔ (action(domain(p)) = task(description(solution(realisation(mbc))))))*

(mbc2) A requirement expressed as a business process in the intended contextual business activity of mbc hasn’t no sub business process.

p ∈ process(context(descriptor(mbc))) ⇔ process(p) = ∅

2.6 Adaptable System Architectures

FORM-based adaptable business (or system) architectures 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 [17]: search, selection, adaptation, and composition operators.

<table>
<thead>
<tr>
<th>Table 10. Specification of Adaptable System Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptableArchitecture = = [serviceView: ( F ) FunctionalPerspective;</td>
</tr>
<tr>
<td>systemView : ( F ) StructuralPerspective;</td>
</tr>
<tr>
<td>processView: ( F ) ProceduralPerspective;</td>
</tr>
<tr>
<td>logicalView: ( F ) LogicalPerspective / ];</td>
</tr>
<tr>
<td>FunctionalPerspective = = FeatureBusinessComponent;</td>
</tr>
<tr>
<td>StructuralPerspective = = SubsystemBusinessComponent;</td>
</tr>
<tr>
<td>ProceduralPerspective = = ProcessBusinessComponent;</td>
</tr>
<tr>
<td>LogicalPerspective= = ModuleBusinessComponent</td>
</tr>
</tbody>
</table>

3. Specific Product Derivation and Related Work

The derivation process we give to derive specific product architectures is described in Figure 1. It takes as input an adaptable architecture (selected in a database of adaptable architectures) containing a set of perspectives. It is decomposed into three steps: perspective commands writing, perspectives adaptation, and specific product architecture optimization. They are:

Step 1: Perspective commands writing. In the first step, a perspective command is written for each generic perspective in the received adaptable architecture.

Step 2: Perspectives adaptation. At this level, adaptation points of generic perspectives are replaced by their actual values based on commands.

Step 3: Specific product architecture optimization. This last step consists of optimized views using refinement rules specified in [15]
3.1 Perspective Commands Writing

Regarding specifications above, respectively in subsections 2.2, 2.3, 2.4 and 2.5, it clearly appears that, to derive a specific perspective, it is essential to have a description in which the sleeping partner has to specify its choices. Being inspired by decision modeling approaches like synthesis [18], Schmid and John [19], KobrA [20], DOPLER [21] or VManage [22], in which decisions appear as an approach that is applied for modeling variability beside feature models [4, 5, 12, 13, 14, 15], we opt for models in which given a variation point, only its actual value that is, its specific value chosen by the sleeping partner is fixed. We call those models Perspective Commands. These models have to be put together with the associated perspective so as to derive a specific model. The schema below specifies a perspective command.
PerspectiveCommand == 
  {perspectivename: Name
   perspertype: FunctionalPerspective \ StructuralPerspective \ ProceduralPerspective \ LogicalPerspective;
   rules: $\mathcal{F}$ Rule
   \forall pc: PerspectiveCommand, #rules(pc) \geq 1
  }

Rule == 
  {name: VariationPoint;
   dependencies: $\mathcal{F}$ VariationPoint
   value: Feature \ SubSystem \ BusinessActivity \ Module
   \forall r: Rule, h \in \text{dependencies}(r) \land h \neq \text{value}(r)
  }

VariationPoint == 
  {type: "instance" | "class"; value: Name}

In the above schemas, the value of a rule is a feature which, respectively, is a sub system, a business activity or a module if and only if the type of the associated perspective is a functional perspective, a structural perspective, a procedural perspective or a logical perspective respectively. We write this property as follows:

$$\forall r: Rule, pc: PerspectiveCommand \cdot r \in \text{rules}(pc),
\text{value}(r) \in \text{Feature} \Leftrightarrow \text{perspertype}(pc) \in \text{FunctionalPerspective} \land
\text{value}(r) \in \text{SubSystem} \Leftrightarrow \text{perspertype}(pc) \in \text{StructuralPerspective} \land
\text{value}(r) \in \text{BusinessActivity} \Leftrightarrow \text{perspertype}(pc) \in \text{ProceduralPerspective} \land
\text{value}(r) \in \text{Module} \Leftrightarrow \text{perspertype}(pc) \in \text{LogicalPerspective}$$

Equally, names of rules in perspective commands correspond to names of adaptation points in the corresponding perspective. That is:

By setting $\text{Constituent} = \text{Feature} \cup \text{SubSystem} \cup \text{BusinessActivity} \cup \text{Module}$,
$$\forall p: Perspective, pc: PerspectiveCommand, r: Rule, c: \text{Constituent} \cdot r \in \text{rules}(pc),
\text{value}(\text{name}(r)) = \text{name}(c) \cdot (c, _) \in \text{Adaptationpoints(realization (p \cdot \text{name}(p) = \text{perspectivename}(pc)))}$$

3.2 Adaptation Mechanism

The first thing we have to know is that, each generic perspective has theory adaptation points and the derivation process of a specific perspective consists of replacing them by their actual values. From there, we define the two following functions which are essential in our derivation approach: $\text{is_resolved}$ and $\text{replace}$.

Given a perspective $p$ and the associated perspective command $pc$, the function $\text{is_resolved}$ returns true if for each adaptation point $ap = (c, _) \text{ of } p$ with $c$ a constituent of $p$, $pc$ contents a rule $r$ verifying $\text{value}(\text{name}(r)) = \text{name}(c)$ and we write $r = \text{rule}(ap)$ otherwise the function $\text{is_resolved}$ returns false.

Given a perspective $p$ and a rule $r$ the function $\text{replace}$ substitutes $\text{value}(\text{name}(r))$ in $p$ by $\text{value}(r)$. 

Copyright © 2015 SERSC
is_resolved: Perspective × PerspectiveCommand ⇔ Boolean

∀ p : Perspective, pc : PerspectiveCommand,

is_resolved (p, pc) = \text{true} ⇔ ∃ ap \in adaptationpoints(realization(p)),

∃ r \in rules(pc) • ap = (value(name(r)), _)

is_resolved(p, pc) = \text{false} ⇔ ¬ (∀ ap \in adaptationpoints(realization(p)),

∃ r \in rules(pc) • ap = (value(name(r)), _))

replace: Perspective × Rule ⇔ Perspective

∀ p : Perspective, r : Rule, replace(p, r) = p[value(name(r))\_\text{value}(r)]

Equally, we define the function constituents which, given a perspective p, provides the set of its constituents.

Constituents: Perspective ⇔ □ Constituent

∀ p : Perspective,

(1) p \in \text{FunctionalPerspective} ⇔ constituents(p) = decomposition(p)
(2) p \in \text{StructuralPerspective} ⇔ constituents(p) = subsystems(solution(realization(p)))
(3) p \in \text{ProceduralPerspective} ⇔ constituents(p) = tasks(solution(realization(p)))
(4) p \in \text{LogicalPerspective} ⇔ constituents(p) = includedmodules(solution(realization(p)))

Given a perspective p and the associated perspective command pc, we derive a specific perspective sp following the algorithm below:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Adapt ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input : p: Perspective, pc: PerspectiveCommand</td>
<td></td>
</tr>
<tr>
<td>Output : sp: Perspective</td>
<td></td>
</tr>
<tr>
<td>Local: wp: Perspective</td>
<td></td>
</tr>
<tr>
<td>Begin</td>
<td></td>
</tr>
<tr>
<td>If is_resolved (p, pc) then</td>
<td></td>
</tr>
<tr>
<td>For each r in rules(pc)</td>
<td></td>
</tr>
<tr>
<td>wp:= replace(p, r);</td>
<td></td>
</tr>
<tr>
<td>End for</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>Write (“ERROR”)</td>
<td></td>
</tr>
<tr>
<td>End if</td>
<td></td>
</tr>
<tr>
<td>sp:= wp</td>
<td></td>
</tr>
<tr>
<td>return sp</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Adaptation algorithm

Example: Let’s consider the functional perspective given below obtained by analysing urban management and the following perspective command:
Name: Functional perspective of real estate services and urban sanitizing

Descriptor:

Intention: \(\text{\textit{Analyse}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(real estate services and urban sanitizing)}}\) \(\text{\textit{TARGET}}\) \(\text{\textit{TARGET}}\)

Context:

Domain: \(C = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(real estate services and urban sanitizing)}}\) \(\text{\textit{TARGET}}\)

Processes:

\(C_1 = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(infrastructures)}}\) \(\text{\textit{TARGET}}\)
\(C_2 = \text{\textit{(anticipate)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(road traffic)}}\) \(\text{\textit{TARGET}}\)
\(C_3 = \text{\textit{(diffuse)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(statistics)}}\) \(\text{\textit{TARGET}}\)

/* \(C_1\) sub-processes */
\(C_{11} = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(occupations)}}\) \(\text{\textit{TARGET}}\)
\(C_{12} = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(liberations)}}\) \(\text{\textit{TARGET}}\)
\(C_{13} = \text{\textit{(establish)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(infrastructure states)}}\) \(\text{\textit{TARGET}}\)
\(C_{14} = \text{\textit{(follow)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(rubbish treatment)}}\) \(\text{\textit{TARGET}}\)

/* \(C_2\) sub-processes */
\(C_{21} = \text{\textit{(register)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(road traffic states)}}\) \(\text{\textit{TARGET}}\)
\(C_{22} = \text{\textit{(link)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(states to causes)}}\) \(\text{\textit{TARGET}}\)
\(C_{23} = \text{\textit{(propose)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(solutions to decongest)}}\) \(\text{\textit{TARGET}}\)

/* \(C_3\) sub-processes */
\(C_{31} = \text{\textit{(register)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(datas)}}\) \(\text{\textit{TARGET}}\)
\(C_{32} = \text{\textit{(consult)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(datas)}}\) \(\text{\textit{TARGET}}\)
\(C_{33} = \text{\textit{(produce)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(indicators)}}\) \(\text{\textit{TARGET}}\)
\(C_{34} = \text{\textit{(edit)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(informations)}}\) \(\text{\textit{TARGET}}\)

/* \(C_{11}\) sub-processes */
\(C_{111} = \text{\textit{(follow)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(lettings)}}\) \(\text{\textit{TARGET}}\)
\(C_{112} = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(building lot acquisitions)}}\) \(\text{\textit{TARGET}}\)
\(C_{113} = \text{\textit{(follow)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(rubbish containers filling)}}\) \(\text{\textit{TARGET}}\)

/* \(C_{12}\) sub-processes */
\(C_{121} = \text{\textit{(follow)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(moving house)}}\) \(\text{\textit{TARGET}}\)
\(C_{122} = \text{\textit{(manage)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(building lots selling)}}\) \(\text{\textit{TARGET}}\)
\(C_{123} = \text{\textit{(follow)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(rubbish evacuation)}}\) \(\text{\textit{TARGET}}\)

/* \(C_{31}\) sub-processes */
\(C_{311} = \text{\textit{(register)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(via cyberespace)}}\) \(\text{\textit{TARGET}}\)
\(C_{312} = \text{\textit{(register)}}\) \(\text{\textit{ACTION}}\) \(\text{\textit{(in the field)}}\) \(\text{\textit{TARGET}}\)

Realization:

Solution:

Adaptation points:

\{( \(c\), \{ \(c_3\), \(c_{11}\), \(c_{12}\), \(c_{13}\), \(c_{14}\) \} \},
\{( \(c_1\), \{ \(c_{11}\), \(c_{12}\), \(c_{13}\), \(c_{14}\) \} \},
\{( \(c_{12}\), \{ \(c_{11}\), \(c_{12}\), \(c_{13}\), \(c_{14}\) \} \},
\{( \(c_{11}\), \{ \(c_{11}\), \(c_{12}\), \(c_{13}\), \(c_{14}\) \} \})\}
From these two elements, by the algorithm describe above, we derive the specific solution below corresponding to the result of the analysis of urban management for a certain context:

### 3.3 Specific Product Architecture Optimization

All the four dimensions (or views) of specific product architectures are concerned by the optimization sub process: the service view, the system view, the process and the logical view.

**3.3.1 Service View Optimization:** The optimization algorithm we use to optimize service views is described in Figure 3. It takes as input the set of functional perspectives of the PL and it generates as output the set of functional perspectives of the specific product. It uses the functional model refiner (FMR) as specified in [15].
Algorithm: OptimizeServiceView()
Input: PL_functionalPerspectives: ServiceView
Output: P_functionalPerspectives: ServiceView
Local: refinedPerspectives: ServiceView
Begin
  refinedPerspectives:=∅;
  For each perspective p in PL_functionalPerspectives do
    For each abstract service as in decomposition (solution (realization (p))) do
      If non atomic(as) then
        p:=FMR(p, decomposition(as));
      End if
    End for
    refinedPerspectives:= refinedPerspectives ∪ {p};
  End for
  P_functionalPerspectives:= refinedPerspectives
End

Figure 3. Service View Optimization: the Optimization Algorithm

3.3.2 System View Optimization: The optimization algorithm we use to optimize system views is described in Figure 4. It takes as input the set of structural perspectives of the PL and it generates as output the set of structural perspectives of the specific product. It uses the structural model refiner (SMR) as specified in [15].

Algorithm: OptimizeSystemView()
Input: PL_structuralPerspectives: SystemView
Output: P_structuralPerspectives: SystemView
Local: refinedPerspectives: ServiceView
Begin
  refinedPerspectives:=∅;
  For each perspective p in PL_structuralPerspectives do
    For each abstract service as in a subsystem ss of p do
      If non atomic(as) then
        p:=SMR(p, decomposition(as));
      End if
    End for
    refinedPerspectives:= refinedPerspectives ∪ {p};
  End for
  P_structuralPerspectives:= refinedPerspectives
End

Figure 4. System view optimization: the optimization algorithm

3.3.3 Process View Optimization: The optimization algorithm we use to optimize process views is described in Figure 5. It takes as input the set of procedural perspectives of the PL and it generates as output the set of procedural perspectives of the specific product. It uses the procedural model refiner (PMR) as specified in [15].
**Algorithm:** OptimizeProcessView()
*Input:* PL_proceduralPerspectives: *ProcessView*  
*Output:* P_proceduralPerspectives: *ProcessView*  
*Local:* refinedPerspectives: *ServiceView*  

Begin  
refinedPerspectives:=∅;  
For each perspective *p* in PL_proceduralPerspectives do  
For each abstract task *t* of *p* do  
If non atomic(*t*) then  
    *p*:=PMR(*p*, decomposition(*t*));  
End if  
End for  
refinedPerspectives:= refinedPerspectives ∪ {*p*};  
End for  
P_proceduralPerspectives:= refinedPerspectives  
return P_proceduralPerspectives  
End

Figure 5. Process View Optimization: the Optimization Algorithm

**3.3.4 Logical View Optimization:** The optimization algorithm we use to optimize logical views is described in Figure 6. It takes as input the set of logical perspectives of the PL and it generates as output the set of logical perspectives of the specific product. It uses the logical model refiner (LMR) as specified in [15].

**Algorithm:** OptimizeLogicalView()
*Input:* PL_logicalPerspectives: *LogicalView*  
*Output:* P_logicalPerspectives: *LogicalView*  
*Local:* refinedPerspectives: *ServiceView*  

Begin  
refinedPerspectives:=∅;  
For each perspective *p* in PL_logicalPerspectives do  
For each abstract business activity *ba* of *p* do  
If non atomic(*ba*) then  
    *p*:=LMR(*p*, decomposition(*ba*));  
End if  
End for  
refinedPerspectives:= refinedPerspectives ∪ {*p*};  
End for  
P_logicalPerspectives:= refinedPerspectives  
return P_logicalPerspectives  
End

Figure 6. Logical View Optimization: the Optimization Algorithm

**3.3.5 Global view optimization:** The optimization algorithm we use to optimize product architectures is described in Figure 7. It takes as input the architecture of the PL and it generates as output the architecture of the specific product. It uses the four optimization algorithms specified above.
Algorithm: OptimizeSpecificProductArchitecture()

Input: PL_Architecture: AdaptableArchitecture

Output: P_Architecture: AdaptableArchitecture

Local: W_Architecture: AdaptableArchitecture

Begin

serviceView(W_Architecture):=
OptimizeServiceView(serviceView(PL_Architecture));
structuralView(W_Architecture):=
OptimizeStructuralView(structuralView(PL_Architecture));
proceduralView(W_Architecture):=
OptimizeProceduralView(proceduralView(PL_Architecture));
logicalView(W_Architecture):=
OptimizeLogicalView(logicalView(PL_Architecture));
P_Architecture:= W_Architecture;
return P_Architecture;
End

Figure 7. Global View Optimization: the Optimization Algorithm

3.4 Related Work

Product derivation methods slightly differ depending on whether the variability modeling follows an amalgamated approach for which the core characteristic is the mix of variability and product line asset concepts into a unique model; or a separated approach for which the key characteristic is the clear separation of concerns methods that this approach provides [23].

The CODA method proposed here is an enrichment of FORM/BCS, an approach constructed from FORM, an extension of FODA. The novelty is that, while some existing feature oriented derivation approaches in the literature are concentrated in feature diagram computation from the product line model [24], and others in configuration and composition of feature models [25], all the four views (service view, system view, procedural view and logical view) of FORM/BCS product line architectures (i.e., FORM/BCS domain adaptable architectures) are considered by the derivation process.

With the aim of helping software architects, a support for the automation of the approach is given. For this, the realization of a software platform accompanying the method can lean on proposed algorithms which lay down logical bases.

4. Conclusions and Future Research

In this paper we have specified product lines derivation techniques for the FORM/BCS approach. The goal was to show that FORM/BCS is not limited to PL architecture description only. For that and to provide a fluent derivation process of specific perspectives from generic ones, we first described perspective commands writing, secondly we proposed an algorithm for perspectives adaptation and finally we specified an algorithm for specific product architecture optimization.

The new research direction in which we are interested now is to enrich generic perspectives of the method by highlighting dependences between their constituents, so as to reinforce perspective coherence. Sure enough, in our previous works we have expressed constraints that must be respected independently of the concerned product family. These constraints highlight only a subset of constraints. Other specific constraints have to be expressed in the FORM/BCS approach, renamed CODA, to improve derivation of products.
Beside the above issue, based on the fact that in the early days of software product line research, very few software product line processes addressed the testing of end-product by taking advantage of the specific features of the product line (commonality and variability), we will attach the description of a testing method adapted to the product line context.

References


Authors

Amougou Ngoumou is holder of a Bachelor’s degree in Computer Science (1998), a Master’s degree in Computer Science (2001) and a PhD in Computer Science (2011) at the University of Yaounde I (Cameroon). He is a Senior Lecturer at the Institute of Technology, University of Douala (Cameroon). His main research interests include Domain Adaptable Architectures, Software Product Line Engineering and Information Systems.

Marcel Fouda Ndjodo is the Head of the Computer Science Department of the Higher Teachers Training College, University of Yaounde I (Cameroon). He is an Associate Professor and is a holder of a PhD in Computer Science at the University of Aix-Marseille II (France, 1992). He also heads the department of educational information systems and numerical technologies at the Higher Teachers Training College. He is the author of many scientific publications and has supervised many PhD theses in information systems and software engineering.