JOURNAL OF SOFTWARE ENGINEERING & INTELLIGENT SYSTEMS
ISSN 2518-8739
30 April 2021, Volume 6, Issue 1, JSEIS, CAOMEI Copyright © 2016-2021
www.jseis.org

# EXTENDED DYNAMIC SOFTWARE PRODUCT LINES ARCHITECTURES FOR CONTEXT INTEGRATION AND MANAGEMENT

**AMOUGOU NGOUMOU [1], MARCEL FOUDA NDJODO [2]**

[1] *Department of Computer Science, College of Technology, University of Douala, Cameroon*

[2] *Department of Computer Science, Higher Teacher Training College, University of Yaoundé I, Cameroon*

Email: ngoumoua@yahoo.fr [1], marcel.fouda@dite-ens.cm [2]

OPEN ACCESS

**Academic Editor:** Abubakar Elsafi, Saudi Arabia.

**Published:** 30 April, 2021

**Funding:** Authors have no funds.

**Article Type:** Research Article

**ABSTRACT**

Nowadays, many embedded system families and application domains such as ecosystems, service-based applications, and self-adaptive systems in pervasive systems and cloud computing require runtime capabilities for flexible adaptation, reconfiguration, and post-deployment activities. However, we still have semi Dynamic Software Product Lines (DSPLs) architectures that need improvement for providing mechanisms for runtime adaptation and behaviour of products. There is an advancement toward designing more dynamic software architectures and building more adaptable software able to handle autonomous decision-making, according to varying conditions in the context. Recent development in DSPLs attempt to address the challenges of the dynamic conditions of such systems but the state of these solution architectures is still modest. In order to provide a more comprehensive architecture there is a need to take into account the context of the DSPLs models, their solution architectures and to cope with uncertainty at runtime. In this research work, we provide a formal representation of the context in the solution architecture of DSPLs models and the management of their behaviour according to that context.

**Keywords:** dynamic software product lines; self-adaptive software; runtime adaptation; solution architectures; context modelling; variability management;

## 1. INTRODUCTION

Abounding application software systems in areas such as mobile computing, autonomic computing, robotics computing, ubiquitous computing and cloud computing [1] require changes at runtime. These adaptive software systems are able to adapt themself dynamically at runtime particularly in response to users' dynamically varying needs as well as environmental constraints [2]. Such changes could be on the features of the system or on how the system caters those features [3].

Adjustment at runtime is a complex process and runtime adaptive systems are at times prone to be unstable, inefficient and unreliable [4]. Furthermore, not taking a systematic approach toward adaptation design results to two difficulties: in a system which is complex and an adaptation mechanism which is hard to modify [2]. Therefore, there have been efforts for the systematic development of runtime adaptive systems by separating the application logic from adaptation logic [5]. Since adaptation logic depend on user requirements, environmental context, and so on, many of these approaches suffer from limitations such as being domain specific, having low performance or limited adaptability [4]. Therefore, the research community still tries to introduce more efficient frameworks and processes for the development of runtime adaptive systems [1].

The Software Product Line Engineering (SPLE) paradigm [6] suggests an effective way to deal with the variability of similar products especially when satisfying requirements of different operating environments and users. This group of similar products is called a product family which is a set of products sharing common features. In SPLE, the development of a system that fits the user-needs and operating environment relies heavily on the reuse of assets through two development phases: domain engineering and application engineering [6]. In the domain engineering phase, those reusable assets needed for developing products are specified and built. These assets consist of common parts and variation points. In the application engineering phase, the target product for the specific operating environment and user requirements are derived using the reusable assets developed in the domain engineering phase. The binding of these assets into a product is mostly performed at design, compile or link time, after which the system stays the same during its lifetime.

***Corresponding Author:*** Amougou Ngoumou
***Email Address:*** ngoumoua@yahoo.fr

SPLE and runtime adaptive systems have much in common [7]. The software product line paradigm (SPL) is a solution for managing the variability of products in a product family. In SPLE, the variability of the products is captured at the domain engineering phase, and the best variant for a specific operating environment and user requirements is selected during the application engineering phase. As such, designing a runtime adaptive system may be considered to be a variability management problem, where first and foremost the variability of the system is captured at design time, and then the best product variant is selected at runtime according to context requirements. The fact that both of these two paradigms are dealing with variability management, one as the problem and one as a solution, has motivated researchers to consider the synergy of these two paradigms in dynamic software product line (DSPL) engineering [8].

In the synergy mentioned above, many challenges front in how to take into account the context? That is how to integrate it and how to manage it such that systems adapt themself dynamically at runtime according to changes in the context. In [9], we have proposed a metamodel of the context for dynamic software product lines. Therefore, the context, that is adaptation logic, is captured as a separate entity near business components, which contents application logic. In this paper, we propose an integration of depending on the context system behaviour, in the application logic that is in system architecture, and give a process on how to manage system behaviour according to changes of the context.

The rest of the article is organized as follows: after this introduction, we give a summary of the context metamodel (section 2), then we show how depending on the context system behaviour can be integrated in system architectures (section 3) and finally before related works, the conclusion and further researches, we proposed context integration functions called producers which formally specify how to manage depending on the context system behaviour.

## 2. CONTEXT METAMODEL

### 2.1 The context

For a better understanding of what we consider as the context, Mahdi et al. gived a goog clarification in [3] that we recall here. The trigger of an adaptation influences how monitoring and planning activities operate in a runtime adaptive system. The types of changes which can trigger adaptation may be generally classified as context changes, system, and user changes.

Context: Context changes are those changes that take place in the external environment of a system. When the system needs to responds to context changes, it requires the right sensors to measure the properties of interest in real-time. The gathered information is then used in planning for adaptation. Therefore, a mechanism should be provided by the engineering approach for capturing the context information. For example, CAPucine uses the COSMOS framework for monitoring the context. In this framework, a component is developed for each sensor named context node, which represents that sensor as a component.

System: System changes are those changes that take place in a system internally. Examples of these changes are failure of a component, performance of a component, and exceptions. For capturing this type of changes, sensors should be integrated within the system implementation. For example, in [10] sensors are used as part of the architecture, monitoring the performance of components such as component response time and memory usage. This information is then used by the adaptation manager for planning about possible system adaptations.

User: Changes are alterations in user requirements or priorities at runtime, and therefore the trigger for adaptation is an external entity such as a user. User preferences are usually captured using an interface. The change in requirements is expressed in high level terms such as features, and is realized by an adaptation manager. In the Helleboogh et al. approach [11], the user of the system can change the operation mode of an automotive transportation system in a warehouse in response to changes in the operating condition. For example, when a load of goods arrives at the warehouse, the operation mode of the system is changed by the user to efficiently unload the packages. Another example for adaptation as a result of user preference changes is that proposed by Wolfinger et al. [12] where the users' preferences are captured by wizard-like dialogues following a decision model. In each dialogue, the user selects from available alternatives in a decision point while the system offers the consequence of selecting each alternative. The adaptation target configuration for the system is built based on user decisions captured by the wizard.

### 2.2 The metamodel

To specify context metamodel freely and formally, we use Z notation. Indispensable types for specification that is abstract domains (Variable_Id, String, Type, and Context_Id) are represented by basic types. We have renamed these domains in capital letters to respect Z conventions.

[VARIABLE_ID, TEXT, TYPE, CONTEXT_ID]

TYPE::= Int | Boolean | Real |Enumeration | Text

Several works on context modelling have proposed a metamodel to describe the use of context models with an abstract syntax [13, 14]. We thus gather the literature existing concepts and reuse them by specifications below. The root of the context metamodel is the main context. Each Context may have sub contexts or variables. Attributes are of type int, boolean, real, string or enumeration, which is a fixed set of values.

### 2.2.1 Variables

Variables are described by a name which is a TEXT not empty, a type, a value and an identifier. Two different variables cannot have a same name. Given a variable, his value belongs to his type. Given a variable identifier, a function (num_v) determines his values. To define Variables, we separate identifiers to values. A variable is represented by two schemas: a schema VariableT and his extension VariableExt.

**VariableT** == [featurename: **TEXT**

featuretype**: TYPE**

actualvalue: **TEXT** |

∀ va: **Variable**T • featurename (va) ≠ ""

∀ va: **Variable**T • actualvalue (va) ϵ featuretype (va)

∀ va1, va2: **Variable**T • featurename (va1) ≠ featurename (va2)]

VariableExt = = [ num_v: VARIABLE_ID ⇸ **Variable**T ]

### 2.2.2 Contexts

A context is characterized by a name which is a TEXT not empty and an identifier. Two different contexts have different names. We specify contexts by the following schemas.

ContextT = = [ name_c: TEXT|
∀ co: ContextT • name_c (co) ≠ "";
∀ co1, c02: ContextT • name_c (co1) ≠ name_c (co2)]

ContextExt = = [ num_c: CONTEXT_ID ⇸ ContextT]

### 2.2.3 Variables of a context

Contexts are specified by sets of variables. Given a context identifier, a function (Specified by) determines the set of its variable identifiers. The domain of this function is included in the domain of the function which given a context identifier determine values of this context. Variables of a context are specified by the following schema:

Specified_byExt = = [ContextExt
VariableExt
Specified_by: CONTEXT_ID ⇸ ℙ VARIABLE_ID|
dom Specify_by ⊆ dom num_c ⋀ ran Specify_by ϵ 𝔽 VARIABLE_ID]

### 2.2.4 Sub contexts of a context

A context is decomposed in a set of sub contexts. Given a context identifier, a function (decomposed as) gives the set of his sub context identifiers. The domain of this function is included in the domain of the function which given a context identifier determine values of this context. Sub contexts of a context are represented by the following schema:

Decomposed_asExt = = [ContextExt
Decomposed_as: CONTEXT_ID ⇸ ℙ CONTEXT_ID|
dom Decomposed_as ⊆ dom num_c ⋀ ran Decomposed_as ϵ 𝔽 CONTEXT_ID]

### 2.2.5 Context model

We synthesized the previous declarations and a context model is specified by a set of functions:

– *root* which given a context identifier determine values of that root context;
– *Specified_by* which given a context identifier determine the set of variables identifiers of that context;
– *Decomposed_as* which given a context identifier determine the set of sub contexts identifiers of that context;
Two different contexts cannot have a same root. Context models are captured by the schema below:

[VARIABLE_ID, TEXT, TYPE, CONTEXT_ID]

ContextModel = = [ root: CONTEXT_ID ⇸ ContextT;
specified_by: CONTEXT_ID ⇸ ℙ VARIABLE_ID;
decomposed_as: CONTEXT_ID ⇸ ℙ CONTEXT_ID |
dom Specified_by = dom root
dom Decomposed_as = dom root
∀ cm1, cm2: ContextModel • root (cm1) ≠ root (cm2)]

The initial state of the context is:

InitContextModel = = [ContextModel′ |
root′ = {}
Specified_by′ = {}
Decomposed_as′ = {}]

### 2.2.6 Operations

We are satisfied here by some consultation operations and some modification operations. The principe is the same for others operations.

- **Consultation operations**

Let's take an example: find sub contexts of a context. Let's decompose:

1. Sub contexts of a context context? are the restriction of the relation Decomposed_as to that context: *Decomposed_as ⊳{ context? }*
2. To have sub contexts identifiers, we project the range: *ran (Decomposed_as ⊳{ context? } )*
3. To have sub contexts we use the relation *contexts*: *contexts (⦇ran (Decomposed_as ⊳{ context? } )⦈)*
Thus the following schema:

SubContextOf = = [ Ξ Context
context?: CONTEXT_ID
subcontexts!: ℙ ContextT |
subcontexts! = contexts (⦇ran (Decomposed_as ⊳ { context? } )⦈)]

Let's take another example: find variables of a context

4. Variables of a context context? are the restriction of the relation Specified_by to that context: *Specified_by ⊳{ context? }*
5. To have variables identifiers, we project the range: *ran (Specified_by ⊳{ context? } )*
6. To have variables we use the relation *variables*: *variables (⦇ran (Specified_by ⊳{ context? } )⦈)*
Thus the following schema:

VariablesOf = = [ Ξ Context
context?: CONTEXT_ID
variables!: ℙ VariableT |
variables! = variables (⦇ran (Specified_by ⊳ { context? } )⦈)]

- **Modification operations**

We have first to verify preconditions to assure invariants. For example, add a variable means add the context if it's s not exist. A good modification method consist to separate different cases in different schemas and then to make union of schemas.

Let's take an example: add a variable of a context. many cases are possible:

1. The variable exists, we don't register them;
2. The variable don't exists
   a) The context don't exists, we refuse
   b) The context exists, we register the variable

ANSWERS::= ExistingVariable | NonexistingContext | RegisteredVariable

PossibleAddVariable = = [ ΔContextModel
var ?: VariableT;
con?: CONTEXT_ID
variable_id ? : VARIABLE_ID
ans!: ANSWERS|
variable_id? ∉ dom num_v
con? ∈ dom num_c
variable_id′ = variable_id ⋃ {variable_id ⇸ var?}
ans! = RegisteredVariable]

AddExistingVariable = = [ ΞContextModel
var ?: VariableT;
con?: CONTEXT_ID
variable_id ? : VARIABLE_ID
ans!: ANSWERS|
variable_id? ϵ dom num_v
ans! = ExistingVariable]

AddVariable NonexistingContext = = [ΞContextModel
var ?: VariableT;
con?: CONTEXT_ID
variable_id ? : VARIABLE_ID
ans!: ANSWERS|
variable_id? ∉ dom num_v
con? ∉ dom num_c
ans! = NonexistingContext]

AddVariable = = PossibleAddVariable ⋁ AddExistingVariable ⋁ AddVariable NonexistingContext

## 3. DEPENDING ON THE CONTEXT SYSTEM BEHAVIOUR INTEGRATION IN SYSTEM ARCHITECTURES

Many embedded systems use context information to adjust their behaviour at runtime. For instance, hardware sensors detect the changes in the environment and the system reacts to the new conditions. In other cases, software detects changes in the quality conditions and selects a new alternative or service at the latest binding time in order to perform the same operation better. According to [15], systems that exploit context awareness must gather data from three different sources: (i) hardware sensors, (ii) software sensors able measure the qualities of the running system (e.g., Internet, services, and scripts), and (iii) user input (e.g., mobile software users that activate a system feature). Consequently, the context-awareness conditions for DSPLs should enable this multi-data entry capability to activate or switch system features dynamically. For DSPLs, runtime variability mechanisms should use features specifically related to manage context information on-demand and offer runtime reconfiguration alternatives for highly configurable products.

In our extended software product line architecture, to manage context data, we distinguish two alternatives: For context data coming from hardware sensors and software sensors, the activity for depending on the context features is a dynamic business activity for which behaviour change according to the value of a context variable. Secondly, the activity of features for which the source of change is user input is managed statically with a static business activity. Context data coming from hardware sensors and software sensors are used by a MAPE (Monitor-Analyze-Plan-Execute) manager, a mechanism from the self-adaptation field used to reconfigure products of self-adaptive software dynamically, for our DSPL autonomic context-aware products. To integrate dynamism in FORM/BCS, the software product line method proposed in [16, 17, 18], following specifications is proposed for a feature business components which for 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.

***FeatureBusinessComponent*** *= =[name:* ***Text;***
*descriptor:****Descriptor***
*realization:* ***Realization***
*]*

The descriptor of a feature business component gives an answer to the following question: "when 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 feature business component is dynamic, which means that it has variables for which values change regularly. Formally, descriptors are defined by the following schemas:

***Descriptor*** *= = [intention :* ***Intention*** *;*
*context :* ***Context***
*|]*

Context = = [ root: CONTEXT_ID ⇸ ContextT;
Specified_by: CONTEXT_ID ⇸ ℙ VARIABLE_ID;
Decomposed_as: CONTEXT_ID ⇸ ℙ CONTEXT_ID |
dom Specified_by = dom root
dom Decomposed_as = dom root
∀ c1, c2: Context • root (c1) ≠ root (c2)]

***Intention*** *= = [action:* ***EnginneeringActivity****;*
*target:* ***Interest*** *|]*

***EngineeringActivity*** *= =* ***AnalysisActivity*** *|* ***DesignActivity***

***AnalysisActivity*** *= {analyze, …}*

***DesignActivity*** *= {design, decompose, describe, specify…}*

***Interest*** *=* ***Domain*** *|* ***BusinessObjects***

***Domain*** *= = [action:* ***BusinessActivity****;*
*target :* ***BusinessObjects*** *;*
*precision :* ***Precision*** *|]*

***BusinessObjects*** *= = 𝔽* ***Class***

***Class*** *= = [name:* ***Name****;*
*attributs : 𝔽* ***Attribut****;*
*operations : 𝔽* ***Operation*** *|]*

***Precision***

***Name***

***Attribute***

***Operation***

The realization section of a feature business 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. This solution may have adaptation points with values comes from three different sources: (i) hardware sensors, (ii) software sensors able measure the qualities of the running system (e.g., Internet, services, and scripts), and (iii) user input (e.g., mobile software users that activate a system feature). 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.

***Realization* = =** *[solution:* ***Feature;***
*adaptationpoints : 𝔽(**Feature** ×𝔽 **Feature**) | ]*

***Feature*** *==* ***DynanicFeature*** */* ***NonDynamicFeature***

***DynamicFeature*** *= =[name:TEXT*
*activity:* ***DynamicBusinessActivity*** *;*
*objects:* ***BusinessObjects;***
*behaviour:* ***Behaviour*** *;*
*decomposition:[common: 𝔽**Feature**; optional: 𝔽**Feature**; variabilities: 𝔽𝔽 **Feature**]*
*generalization: 𝔽**Feature***
*| ]*

***NonDynamicFeature*** *= =[name:TEXT*
*activity:* ***NonDynamicBusinessActivity*** *;*
*objects:* ***BusinessObjects;***
*behaviour:* ***Behaviour*** *;*
*decomposition:[common: 𝔽**NonDynamicFeature**]*
*| ]*

***AddDynamicBusinessActivity*** = = [
Δ Feature ΔContextModel
var ?: VariableT;
con?: CONTEXT_ID
variable_id ? : VARIABLE_ID
ans!: ANSWERS|
variable_id? ∈ dom num_v
con? ∈ dom num_c
dynamicbusinessactivity′ = *𝔽 (featuretype(var?) x* ***BehaviourInstance****)*
ans! = RegisteredDynamicBusinessActivity
]

***NonDynamicBusinessActivity= = BehaviourInstance***

***BehaviourInstance == TEXT***

## 4. DEPENDING ON THE CONTEXT SYSTEM BEHAVIOUR MANAGEMENT

From the early efforts made in the area of systematic implementation of runtime adaptive systems, the separation between parts of the system focusing on adaptation logic and application logic has been promoted [19]. Adaptation logic is concerned with selecting the best possible variant of the running system based on the current context while the application logic provides the required functionalities of the system. This practical distinction makes the development of a runtime adaptive system easier and results in a more reliable system [4]. The separation is usually implemented as a two-layered structure. In this structure, an adaptation management layer is implemented over the application layer. The adaptation management layer accommodates adaptation logic. The application layer enables runtime reconfiguration and also provides the interfaces for monitoring and reconfiguring the system for the adaptation management layer as well as accommodating the application logic. Figure 1 shows the adaptation manager layer and the application layer.

The focus of DSPL engineering approaches is mostly on building adaptation managers for runtime adaptive systems. In order to break down the concerns for building an adaptation manager, we conceptually represent an adaptation manager as a MAPE-K loop model [20] similar to what Bencomo et al. [21] have proposed. In their research, they have proposed a conceptual model for DSPL adaptation management process in which the system adaptation manager can be viewed as a MAPE-K loop. Mahdi et al. [3] adopted their conceptual model for adaptation management and provided descriptions of how this process can be mapped to the MAPE-K loop. However, certain explanations and models are still semi-formal and that don't facilitate the writing of maps mechanisms of DSPL adaptation management process to the MAPE-K loop. Here, by a rigorous specification, we improve the conceptual model for DSPL adaptation management process and how

this process can be mapped to the MAPE-K loop. The MAPE-K loop (Fig. 1) model is used in autonomic computing to represent the main concerns for building an autonomic manager.

An autonomic manager is responsible for handling autonomic properties of a system. MAPE consists of the initial characters of the four main steps to be taken by an autonomic manager: Monitor, Analyze, Plan, and Execute. The added K character following the hyphen stands for knowledge which is usually represented by models which are used in the first four steps. The monitoring step is about capturing those properties which are required for planning the adaptation of the system. The analysis step addresses concerns regarding the examination of the monitored data to get new information by considering a combination of monitored values and taking into account the monitoring history which will be useful for the planning step. The planning step decides if adaptation is essential and selects the best system variant if adaptation is necessary. After planning for the adaptation, the execution step safely executes the adaptation.

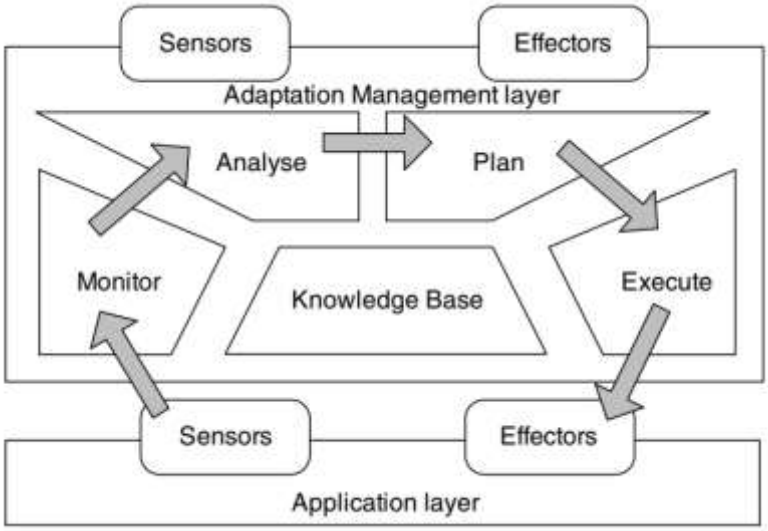


**Figure. 1** MAPE-K loop [3]

We have established the correspondences that can be made between the Feature Oriented Reuse Method with Business Component Semantics and concepts in MAPE-K loop model. Thus, the Knowledge Base corresponds to the Feature Business Component and the Application layer corresponds to the Suitable Variant Instance layer. By doing that, the following model is derived from the MAPE-K loop model.

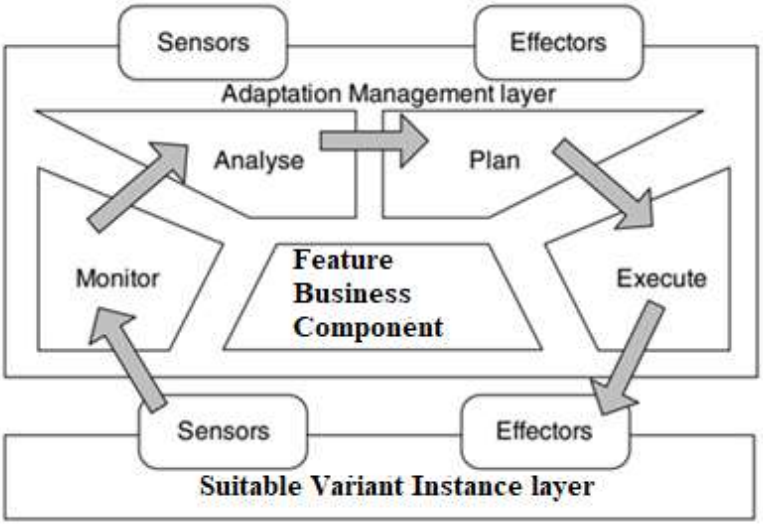


**Figure. 2** Feature Business Component MAPE-K loop

Since the MAPE-K model derives its name from the main tasks in the feedback control loop typical of self-adaptive systems: monitoring, for detecting events that might require adaptation; analysis, for analysing a change's impact on the product's requirements or constraints; planning, for deriving a suitable adaptation to cope with the new situation; and executing, for carrying out the adaptation. K denotes the knowledge necessary to perform these tasks in a useful way. To improve the adaptation management layer, we define three functions called producers:

the context information producer which supports the context reasoning activity consisting of accept one or more existing context information and produce new more abstract context information, the architecture model producer which supports planning activity consisting of using a reasoner which considers the variability model that is adaptation points, the context, and the solution that is the current variant of the system as inputs and then develops the most suitable variant for the current context according to its planning knowledge that is the architecture model; the target variant producer which supports the activity of reconfiguring a running instance such that current variant gives its place to the target variant while ensuring system consistency. The context information of a system is the starting point of the process; this information is obtained by context sensing mechanisms which are usually implemented by enforcing a standard for designing sensors (e.g. defining standard interfaces for sensor components) which provide a unified way for capturing contextual information.

### 4.1 Context reasoning

The purpose of the context reasoning activity is to provide abstract context information from existing context information. This activity is carried by the total function CIP, the context information producer, whose purpose is to produce abstract information from concrete information of contexts. Figure 3, which intensively uses the context model defined in section 2, specifies the context information producer. In that figure, any text inside /* */ is a comment which explains the formal notation. The context information production rules are mentioned below in Figure 3.

**Input:** Context identifier *c_id***,** existing context information provider function *ci* for changes taking place in the external environment of a system and the abstract context information provider function *abs*

**Output:** Abstract context information *CIP(c_id, ci, abs)* produced from context identifier *c_id*, existing context information provider function *ci* for changes taking place in the external environment of a system and the abstract context information provider function *abs*

**Construction schema:**
*CIP(c_id, ci, abs) = (CIP.root(c_id, ci, abs), CIP.specify_by(c_id, ci, abs), CIP.decomposed_as (c_id, ci, abs))*

**Semantics rules:**

1. $CIP.root(c\_id, ci, abs) = root(c\_id)$

/* the root for the new context information is the root for the previous context information */

2. $\forall v\_id \in specify\_by(c\_id) \bullet CIP.num\_v(v\_id).actualvalue = abs(ci(v\_id))$

   2.1. *ci* is a function defined as follows *:*
   *ci*: VARIABLE_ID $\nrightarrow$ TEXT

   2.2. *abs* is a function defined as follows *:*
   *abs*: TEXT $\nrightarrow$ TEXT

/* for each variable in the context, the value is replace by the abstract actual value obtained from the existing context information provider function *ci* from changes taking place in the external environment of the system */

3. $\forall context\_id \in decomposed\_as(c\_id), variable\_id \in specify\_by(context\_id) \bullet CIP.num\_v(variable\_id).actualvalue = abs(ci(variable\_id))$

/* for each variable in a subcontext of the target context, the value is replace by the abstract actual value obtained from the existing context information provider function *ci* from changes taking place in the external environment of the system */

**Figure. 3** The context information production rule

### 4.2 Planning

The purpose of the planning activity is to provide the most suitable variant for the current context according to its planning knowledge that is the architecture model. This activity is carried by the total function AMP, the architecture model producer, whose purpose is to develop the most suitable variant for the current context from the adaptable perspective of the system which is a feature business component with adaptation points and the context identifier. Figure 4 defines the architecture model producer. The architecture model production rules are mentioned below in Figure 4.

**Input:** The adaptable perspective of the system *aps* and context identifier *c_id*

**Output:** Most suitable variant *AMP(aps, c-id)* produced from The adaptable perspective of the system *aps* and the context identifier *c_id*

**Construction schema:**

*AMP(aps, c_id) = (AMP.name(aps, c_id), AMP.descriptor(aps, c_id), AMP.realization (aps, c_id))*

**Semantics rules:**

1. *AMP.name(aps, c_id) = name(aps)*

/* the name for the most suitable variant is the name for the received adaptable perspective of the system */

2. *AMP.descriptor(aps, c_id) = descriptor(aps) • root(context(descriptor(aps))) = c_id*

/* the descriptor for the most suitable variant is the descriptor of the received adaptable perspective of the system in which variables values are is the actual values of the context */

3. *AMP.realization(aps, c_id) = realization(aps) •*

3.1. $\forall f \in solution(realization(aps)), var \in specified\_by(c\_id) \bullet featurename(var) = name(f)$,
***if*** *activity(f) = DynamicBusinessActivity* ***then***
*behaviour(f ) = actualbehaviour(f, actualvalue(var))*
/* where *actualbehaviour* is a function for which given a feature and his associated variable provides the actual behaviour of the target feature */

3.2. $adaptationpoints(realization(aps)) = \varnothing$

**Figure. 4** The architecture model production rule

### 4.3 Reconfiguring

The purpose of the reconfiguring activity is to construct a most suitable running instance for the current context according to its actual architecture model without adaptation points provided by the planning activity. This activity is carried by the total function TVP, the target variant producer, whose purpose is to provide the most suitable running instance for the current context from the functional perspective of the system which is a feature business component without adaptation points. Figure 5 defines the target variant producer.

**Input:**
- The functional perspective *fp* of the system that is a feature business component which respects his metamodel *Mfp*;
- The metamodel *Mvi* of the most suitable variant instance *vi* of the system

**Output:**
The most suitable variant instance *TVP(fp, Mfp, Mvi)* produced from the functional perspective *fp*, his metamodel *Mfp* and the metamodel of the most suitable instance *vi* of the system *Mvi.*

**Construction schema:**

*TVP(fp, Mfp, Mvi) = vi*

/* The most suitable instance *vi* is the output model that is the code of the system for the target platform */

**-**

**Figure. 5** The target variant production rule

## 5. RELATED WORKS

As present-time systems such as pervasive systems or ubiquitous systems or Internet of Things demand more and more post-deployment activities and runtime capabilities, several DSPL approaches have emerged in the last decade to deal with these requirements [22]. For instance, some authors have proposed to leverage techniques from service-oriented architectures to build service-oriented DSPLs, i.e., DSPLs built by composing services [23, 24] or, similarly, aspect-oriented DSPLs [25]. Recently, Bashari et al.[3] proposed a classification of various DSPL implementations and compared their adaptation mechanisms:

The DSPL engineering approach utilizes SPL, Common Variability Language (CVL) [26] and aspect-oriented programming for enabling runtime reconfiguration of business processes represented with Business Process Execution Language (BPEL). BPEL is a language for the formal specification of business process behavior focusing exclusively on Web services. The approach by Baresi et al. has been built as an implementation of CVL, which is a suggested generic approach for managing variability in domain specific languages. In this CVL-based

approach, the variants of the system are chosen by selecting the desired features from the feature model of the system. The selected features in the feature model are mapped to series of substitutions in the base BPEL code. Base BPEL code is a process which is considered as a reference point for defining variability. A substitution is defined as placing a BPEL aspect from the CVL library into placement location in the base code. The target business process is derived using these placements after which it replaces the current business process. Runtime reconfiguration has been enabled in this approach by extending the ActiveBPEL engine [27] using aspect-oriented programming to modify BPEL at runtime by changing the included aspects [28].

Bencomo et al. (Genie): Genie is a model-based approach for developing selfadaptive systems for grid mobile computing and embedded systems. In this approach, two variability dimensions are recognized for a system and linked to each other in the development process: environment or context variability which represents variability in the environment and structural variability which represents variability in the architecture of the running system. The structure of the system can adapt according to changes in the environment by linking these two dimensions. In the Genie approach, the environment is represented by a state transition diagram, where states represent the states of the environment and the transitions represent possible environment state changes which are guarded by conditions over the context of the system. The structural variability of the system is represented with OpenCOM DSL which is a domain specific language that represents system architecture. This approach takes advantage of the OVM [29] for linking those models which represent two dimensions. OVM traditionally is used for tracing variability between models at different levels of abstraction in the software product line. Genie enables runtime reconfiguration using OpenCOM [30] middleware.

Cetina et al. (MoRE): MoRE is a reconfiguration engine which incorporates SPLE ideas for developing autonomic pervasive systems such as Smarthomes. This approach uses feature models as a variability space model at runtime. Runtime adaptation takes place in this approach by modifying the current feature model configuration of the system using a set of condition/resolution rules, where these rules define which features are activated/deactivated as context conditions change. The system managed by MoRE should be built using service-oriented architectures, where services and devices communicate using channels. The actual system adaptation is performed by mappings features of the feature model to the services and channels that realize these specific features. Therefore, a feature model configuration is mapped to a configuration of the system, where a set of services and channels are active while the others are inactive. This approach also proposes a technology-independent, domain-specific language named PervML [31] representing system architecture. PervML is used for representing services and devices and how they are connected though channels.

Floch et al. (MADAM): MADAM is a utility-based approach for building selfadaptive systems for mobile and distributed environments. This approach extends an SPL engineering concept named configurable product bases [32] to enable runtime adaptation. Configurable product bases is a type of SPLE approach which requires no product specific development during application engineering. The MADAM approach extends configurable product bases by adding a utility-based planner to it and enabling configurable product bases to reconfigure at runtime. In this utilitybased approach, a utility function is defined over the system properties and its context to represent desirability of the current configuration in the current context. Therefore, the goal of adaptation becomes the maximization of the value of this function. The planner functions by taking advantage of property predictor functions, which can predict different properties of the system according to the system configuration and the context. When the utility becomes unacceptable, the planner uses a brute-force technique to find a configuration with the highest predicted utility and adapts to it.

Gomaa et al. (REPFLC): The Reconfigurable Evolutionary Product Family Life Cycle (REPFLC) approach proposes a lifecycle for SPLE which covers product execution phase in addition to those related to domain engineering and application engineering. In this lifecycle, reconfiguration patterns are designed at the domain engineering phase, alongside other domain engineering assets, in order to define reusable patterns for safe runtime reconfiguration of components. These reconfiguration patterns provide state-based and scenario-based behavioral models for an adaptation. The state-based model represents how the system adapts in an adaptation while the scenario-based model represents the requirements of the situation when the adaptation is necessary. These reconfiguration patterns are available in the configured target system. The system uses these reconfiguration patterns to safely reconfigure at runtime when an adaptation is required.

Morin et al. (DiVA): This approach builds on ideas from SPLE and aspectoriented modeling to develop self-adaptive systems. In this approach, the architecture of the system is built in three layers. The bottom layer contains the application logic; the top layer plans the adaptation; and the middle layer creates the link between the top and the bottom layers. The middle layer creates the bottom to top link by analyzing data from sensors and converting the data into context information useful for reasoning. It also creates the top to bottom link by reflecting changes in the system architecture model into the running system. The top layer uses feature models at runtime for managing the variability of system at an abstract level. Feature models are used to decide

about the best configuration for the current context using a reasoner. Planning results in a feature model configuration which represents what system features should be available in this context. This feature model configuration is then mapped to the architectural model of the system using aspect model weaving. The approach uses a system variant configuration checker to ensure consistency of the target architecture model at runtime.

Parra et al. (CAPucine): In this approach, software product line and service oriented architecture have been used to create a process to build systems that monitor context changes in order to dynamically incorporate required assets at runtime. The variability of the system is kept modeled using a feature model. The approach gathers context information using a context sensing middleware which has been built based on the COSMOS framework. The changes in the context triggers changes in the feature model based on a set of rules. This approach suggests the use of aspect model weaving [33] for generating the architecture model of the system from feature model configuration at runtime. In aspect model weaving, features are mapped to aspect models representing different aspects of a given feature in the system architecture. For a feature model configuration, the corresponding aspect models of selected features are woven into the base model of the system to create an integrated model of the architecture of the system. Then the approach uses FraSCAti [34] which is a Fractal-based Service Component Architecture (SCA) [35] platform with dynamic properties, to reconfigure the system according to the target architecture model at runtime.

Bencomo et al. [36] discuss these different approaches for building a DSPL and show that these supposed DSPLs were not as 'dynamic' as expected. They also point out the need to cope with uncertainty at runtime by providing support for the DSPL evolution. In addition, although systems with the ability to adapt at runtime can be built using any of these approaches, the types of adaptation offered by them are different and there is any approaches which is universal for any problem domain. This encourages our own work to provide a more comprehensive taken into account of the context in DSPLs.

## 6. CONCLUSION AND FUTURE RESEARCH

New challenges that have emerged in digital domains such as pervasive computing, ubiquitous computing or Internet of Things and for which static or conventional SPL approaches cannot raises because of the fact that many systems nowadays require the adaptation to different context conditions or working under better quality conditions motivated this work. We intend to provide a more comprehensive taken into account of the context in DSPLs models, their solution architectures and to cope with uncertainty at runtime. The result is extended dynamic software product lines' architectures for context integration and management. In these architectures, the adaptation management was seen as a MAPE-K loop. However, given that certain explanations and models are still not very rigorous and don't facilitate the writing of maps mechanisms of DSPL adaptation management process to the MAPE-K loop, in the proposed architectures, by a formal approach, the conceptual model for DSPL adaptation management process and how this process can be mapped to the MAPE-K loop are improved. For future work, we are interested in evaluating our approach with software tools. However, over the course of time, SPLs are subject to evolution, which require engineers to change features, constraints, and their realizations to create new versions of the software. Thus, we are also interested in managing the evolution of SPLs.

## REFERENCES


1. B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun and B. Cukic, "*Software engineering for self-adaptive systems: A research roadmap*.", in Software Engineering for Self-Adaptive Systems, 2009, pp. 1–26.
2. P. Oreizy, N. Medvidovic and R. N. Taylor, "*Architecture-based runtime software evolution*.", in Proc. 20th Int. Conf. Software Engineering, 1998, pp. 177–186.
3. M. Bashari, E. Bagheri and W. Du, "*Dynamic Software Product Line: A Reference Framework*.", in International Journal of Software Engineering and Knowledge Engineering, Vol. 27, No. 2, 2017, pp 191–234.
4. D. Weyns, M. U. Iftikhar, S. Malek and J. Andersson, "*Claims and supporting evidence for self-adaptive systems: A literature study*.", in ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2012, pp. 89–98.
5. Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Mller, M. Pezz and M. Shaw, "*Engineering self-adaptive systems through feedback loops*.", in Software Engineering for Self-Adaptive Systems (Springer, 2009), pp. 48–70.
6. K. Pohl, G. Bockle and F. V. D. Linden, "*Software Product Line Engineering: Foundations, Principles and Techniques*.", Vol. 10 (Springer, 2005).
7. V. Alves, D. Schneider, M. Becker, N. Bencomo and P. Grace, "*Comparative study of variability management in software product lines and runtime adaptable systems*.", in Workshop on Variability Modelling of Software-intensive Systems, 2009, pp. 9–17.

8. S. Hallsteinsen, E. Stav, A. Solberg and J. Floch, "*Using product line techniques to build adaptive systems*.", in Software Product Line Conference, 2006, pp. 10–150.
9. N. Amougou, M. Fouda, "*Context metamodel in pervasive systems for dynamic software product lines*", Journal of Software Engineering & Intelligent Systems, Vol. 5, Is. 3, pp. xxx-xxx, 2020.
10. J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund and E. Gjorven, "*Using architecture models for runtime adaptability*.", IEEE Softw. 23(2) (2006) 62–70.
11. A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout and W. V. Betsbrugge, "*Adding variants on-the-°y: Modeling meta-variability in dynamic software product lines*.", in Proc. Third Int. Workshop on Dynamic Software Product Lines, 2009, pp. 18–27.
12. R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher and H. Prahofer, "*Supporting runtime system adaptation through product line engineering and plug-in techniques*.", in Int. Conf. Composition-Based Software Systems, 2008, pp. 21–30.
13. Matthias Baldauf, Schahram Dustdar and Florian Rosenberg, "*A survey on context-aware systems.*", Int. J. Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, 2007.
14. C. Bettini & al., "*A Survey of Context Modelling and Reasoning Techniques*.", Preprint submitted to Elsevier, 2008.
15. Capilla, R., et al., "*An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry*." J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038.
16. M. Fouda, N. Amougou, "*The Feature Oriented Reuse Method with Business Component Semantics*.", International Journal of Computer Science and Applications, Vol. 6, No. 4, pp 63-83, 2009.
17. M. Fouda, N. Amougou, "*Product Lines' Feature-Oriented Engineering for Reuse: A Formal Approach*.", International Journal of Computer Science Issues, Vol. 7, Issue 5, pp 382-393, 2010.
18. M. Fouda, N. Amougou, "*Transformational Variability Modelling Approach To Configurable Business System Application*.", in Software Product Line – Advanced Topic, Edited A. O. Elfaki, Intech Publisher, pp. 43-68, 2012.
19. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, "*An architecture-based approach to selfadaptive software*.", Intell. Syst. Their Appl. 14(3) (1999) 54–62.
20. J. O. Kephart and D. M. Chess, "*The vision of autonomic computing*.", Computer 36(1) (2003) 41–50.
21. N. Bencomo, S. Hallsteinsen and E. Almeida, "*A view of the landscape of dynamic software product lines*.", Computer 45(10) (2012) 36–41.
22. Capilla R, Bosch J, Trinidad P, Ruiz-Cortés A, Hinchey M. "*An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry*." Journal of Systems and Software 2014; 91: 3–23.
23. Gomaa H, Hashimoto K. "*Dynamic Software Adaptation for Service-oriented Product Lines*.", In: ACM; 2011; Munich, Germany: 35:1–35:8.
24. Baresi L, Guinea S, Pasquale L. "*Service-Oriented Dynamic Software Product Lines*.", Computer 2012; 45(10): 42–48.
25. Parra C. "*Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations.*" Thesis. Université des Sciences et Technologie de Lille - Lille I, 2011.
26. O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen and A. Svendsen, "*Adding standardized variability to domain specific languages*.", in Int. Software Product Line Conference, 2008, pp. 139–148.
27. A. Endpoints, Activebpel: The open source BPEL engine, http://www.activevos.com/learn/open-source.
28. A. Charfi and M. Mezini, "*Ao4bpel: An aspect-oriented extension to BPEL*.", World Wide Web 10(3) (2007) 309–344.
29. K. Pohl, G. Bockle and F. V. D. Linden, "*Software Product Line Engineering: Foundations, Principles and Techniques*.", Vol. 10 (Springer, 2005).
30. G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee and J. Ueyama, "*A component model for building systems software*.", in IASTED Software Engineering and Applications, 2004.
31. J. Munoz and V. Pelechano, "*Applying software factories to pervasive systems: A platform speci¯c framework*.", in ICEIS (3), 2006, pp. 337–342.
32. F. V. D. Linden, J. Bosch, E. Kamsties, K. Knsl and H. Obbink, "*Software product family evaluation*.", in Software Product Lines (Springer, 2004), pp. 110–129.
33. J.-M. Jzquel, "*Model driven design and aspect weaving*.", Softw. Syst. Model. 7(2) (2008) 209–218.
34. O. Consortium, Frascati project, http://frascati.ow2.org.
35. O. C. Oasis, "*Service component architecture (SCA)*.", 2011, http://oasis-opencsa.org/sca.
36. Bencomo N, Lee J, Hallsteinsen SO. "*How Dynamic is your Dynamic Software Product Line?*", In: ; 2010; Jeju Island, Korea: 61–68.

**AUTHORS PROFILE**

**Dr Amougou Ngoumou** has received Bachelor of Computer Science (1998), Master of Computer Science (2001) and PhD degree in Computer Science (2011) at the University of Yaounde I (Cameroon). He is a Senior Lecturer at the Institute of technology of the University of Douala (Cameroon). His main research interests include Dynamic Software Product Lines, Domain-Specific languages and Information Systems.

**Pr Marcel Fouda Ndjodo** is the Head of the Computer Science Department of the Higher Teacher Training College of the University of Yaounde I (Cameroon). He is a Full Professor and has received a PhD in Computer Science at the University of Aix-Marseille II (France, 1992). He coordinates besides the information systems and numerical technologies of education at the higher teacher training college. He is author of many scientific publications and has supervised many PhD thesis in information systems and software engineering.