Realising Variability in Dynamic Software Product Line Solutions

Jane D. A. Sandim Eleutério[a,b], Breno B. N. de França[b], Cecilia M. F. Rubira[b], Rogério de Lemos[c,d]

[a] *Faculty of Computing, UFMS, Campo Grande, MS, Brazil*
[b] *Institute of Computing, UNICAMP, Campinas, SP, Brazil*
[c] *School of Computing, University of Kent, UK*
[d] *CISUC, University of Coimbra, Portugal*

Abstract

Modern systems need to be able to self-adapt to changes in user needs, and changes affecting the system itself or its environment. Dynamic software product line (DSPL) is an engineering approach for developing self-adaptive systems based on commonalities and variabilities for a family of similar systems. Currently, many DSPL approaches fail to meet all adaptability requirements, and in many cases, they are developed in a such unstructured manner that the controller is not explicitly represented, for example. We specify a two-dimension taxonomy to address basic technical issues for realising variability in DSPLs. The self-adaptation dimension classifies the different design choices for the adaptability requirements. The DSPL variability dimension classifies different design choices for implementing variability schemes and for creating different kinds of feature models. Our study was substantiated by surveying several DSPL approaches, and evaluating and comparing their different design strategies. We also summarise practical issues and difficulties, identify major trends in actual DSPL proposals, and suggest directions for future.

Keywords: Dynamic Software Product Line, Self-Adaptive Systems, Adaptation Taxonomy, Variability Taxonomy

1. Introduction

Modern software systems need to adapt both at design time and runtime to heterogeneous environments and devices. Software Product Lines (SPLs) are related to emerging techniques where several artefacts are reused. Software Product Line (SPL) deals with the modelling of commonalities and variabilities among a family of similar systems [1]. Commonality corresponds to similar parts among family products, while variability is defined as the ability of a product to be extended, modified, customised, or

Email addresses: jane@facom.ufms.br (Jane D. A. Sandim Eleutério), breno@ic.unicamp.br (Breno B. N. de França), cmrubira@ic.unicamp.br (Cecilia M. F. Rubira), r.delemos@kent.ac.uk (Rogério de Lemos)

configured for a specific context [2]. In general, feature models [3] are used to represent variability and commonality by means of features that can be classified as mandatory, optional or alternative [4]. Mandatory features are present in all products derived from SPL. Optional features may or may not appear in derived products, while alternative features may be selected according to mutual exclusion constraints. While SPLs deal with static variability, which is defined at design time, and its decision is performed using binding, Dynamic Software Product Line (DSPL) deal with both static and dynamic variability. Dynamic variability, also called late or runtime variability, is also defined at design time, however, its decision is performed using binding at deployment or runtime.

Systems able to adapt their behaviour and/or structure at runtime are called self-adaptive software systems (SASS) [5]. These systems should are able to adapt in response to changes that occur to the system itself, its environment, or even its goals, with no human interference. Feedback control loops provide a generic mechanism of self-adaptation [6], and it is often modelled as the MAPE-K loop [7].

Software engineers could systematically reuse SASS good practices to develop DSPL approaches.

From the perspective of variability, software product line (SPL) deals with commonalities and variabilities in product families by performing binding during the design phase. Dynamic software product lines (DSPLs) are considered a sub-type of SPLs, which handles variability by performing binding at runtime. From the perspective of dynamicity, self-adaptive software systems (SASS) are able to dynamically adapt their structure during runtime when responding to changes.

According to Bencomo *et al.* [8], many DSPL approaches are not as dynamic as they should be, because they partially (or do not) implement self-adaptation activities. Bencomo *et al.* [8] also concludes the research on DSPL variability is still heavily based on the specification of variability decisions during design time. In other words, variability and its decision options are defined at design time, and variability decision making occurs during execution. Ideally, a DSPL solution should allow new decision options to be incorporated at runtime, improving the dynamicity of such solution. Dynamicity can be understood as the system's ability to undergo changes and adapt at runtime, reacting to foreseen, foreseeable, and unanticipated changes in the most autonomous way as possible. Ideally, DSPL systems should be easy to understand, maintain and reuse. With such systems growing in size and complexity, employing self-adaptation and variability techniques while satisfying the software quality attributes, such as modifiability, reusability, and testability, are deep concerns to the DSPL systems engineers.

The goal of this chapter is to define a two-dimensional taxonomy that aims to identify how design issues related to variability and self-adaptation can affect the key quality attributes of DSPLs. In the context of self-adaptive system, variability design issues are related to the managed subsystem (or target system), the self-adaptation design issues are related to managing subsystem (or controller, or feedback control loop), and the quality attributes are related to the self-adaptive system that encapsulates both the managed and managing subsystems. Based on the above, the major contributions of this chapter are: (i) identification of design issues related to variability and self-adaptation that are relevant to DSPLs; and (ii) evaluation of these design issues in the

context of quality attributes that are pertinent to DSPLs. Based on our findings, we have also identified several research challenges when building DSPLs that make use of the SASS principles.

The remainder of this chapter is organised as follows. Section 2 gives a brief description of dynamic software product line (DSPL) and self-adaptive software systems (SASS). Section 3 describes a proposed taxonomy for classifying different design issues to DSPL, analysing of with a set investigated DSPL approaches. Section 4 presents an appropriate design criteria based on quality attributes for an ideal DSPL solution. Section 5 discusses some research challenges. Finally, Section 6 presents some concluding remarks and lessons learned.

2. Perspectives on Variability and Self-adaptation

Over the years, several contributions have emerged that apply variability modelling concepts, using features models, in the development of self-adaptive software systems, such as MoRE [9], MADAM [10], and DiVA [11]. From the viewpoint of runtime adaptation, variability modelling can include both system (functional) and context (environmental) information, and if well established, variability modelling promises to be a valuable basis for the definition of appropriate models in time of execution. Context information includes data related to the environment where the system is running, such as network, memory resources, battery level, and battery consumption, and other computing resources, while system information is obtained by specific sensors that monitor the system components at runtime.

On the other hand, research on DSPLs has increasingly been using SASS techniques to improve the management of runtime variability, such as ASPL [12], ArCMAPE [13], and BSN-DSPL [14]. The MAPE-K feedback control loop (see Section 2.2) can be applied to automate variability and product derivation management processes at runtime. Thus, we can infer that there is a synergy between these two areas of research, which encompasses both DSPLs that support dynamic variability using SASS techniques, and SASS that apply (D)SPL techniques to manage variability at runtime.

2.1. Dynamic Software Product Line

The Dynamic Software Product Line (DSPL) extends the concept of conventional SPL since the latter emphasises the variability analysis, decision-making and product configuration at the design time. DSPL emphasises variability analysis at design time, but postpones the decision of the variability and the application reconfiguration to be made at runtime.

In SPLs, the binding can occur at design time to generate a product using static binding. In the case of DSPLs, the binding should occur at runtime in order to support dynamic variability. Dynamic variability (also called late or runtime variability) can be represented using dynamic compositions, which is a set of features with dynamic binding [15]. In contrast, static variability can be represented using static compositions, which is a set of features with static binding [16].

2.2. Self-adaptive Software Systems

Self-adaptive software systems are systems that are able to modify their behaviour and/or structure in response to changes that occur to the system itself, its environment, or even its goals [17].

Self-adaptation enables a system to adjust itself in response to changes, and there is a wide range of approaches to engineering self-adaptive software systems through tried and tested reference models. For example, Weyns et al. formalise the self-adaptive system's environment as a collection of *processes* and *attributes* using their FOrmal Reference Model for self-adaptation (FORMS) [18]. Although there are several reference models for self-adaptive software systems [7, 19, 20], most of them share the common use of a feedback loop [6, 21, 22]. In this chapter, we adopt as a feedback control loop, the Monitor, Analyse, Plan, Execute - Knowledge (MAPE-K) reference model [7], as shown in Figure 1.

The main feedback control loop, in the *Managing Subsystem*, which embodies the stages of the MAPE-K reference model, observes (via probes) and adapts (via effectors) a *Managed Subsystem*. The *Monitor* stage enables to obtain the state of the target system and its environment. The *Analyse* stage analyses the state of the target system and its environment in order, first, to decide whether adaptation should be triggered, and second, to identify the appropriate courses of action in case adaptation is required. The *Plan* stage, first, selects amongst alternative course of action those that are the most appropriate, and second, generates the plans that will realise the selected course of action. The *Execute* stage executes the plans that deploy the course of action for adapting the system. Finally, *Knowledge* represents any information related to the perceived state of the target system and environment that enables the provision of self-adaptation.

2.3. DSPL versus SASS

Weyns et al. [23] use the general terms manager subsystem and managed subsystem to indicate the constituent parts of a self-adaptive software system. The first part of Figure 1 (left side) represents this separation, where the managing subsystem manages the managed subsystem and is organised according to an autonomic control loop, and the managed subsystem consists of the application logic that provides the system's domain functionality.

The second part of Figure 1 (right side) illustrates a DSPL application represented as a self-adaptive system, in order to encompass managed and managing subsystems. The managed subsystem is a DSPL, which contains the application logic, according to its domain. This subsystem is unconscious of the rest of the system, and its components (parts) are isolated and do not access the rest of the system. The managing subsystem is a controller, which includes the adaptation logic and is independent of DSPL. So the logic can be reused in different domains. The managing subsystem is transparent to the DSPL and is responsible for monitoring, adaptation reasoning and acting on DSPL. Finally, the entire system interacts with an environment, which refers to the context in which the system is running, including hardware, operating system, other systems and networks access.

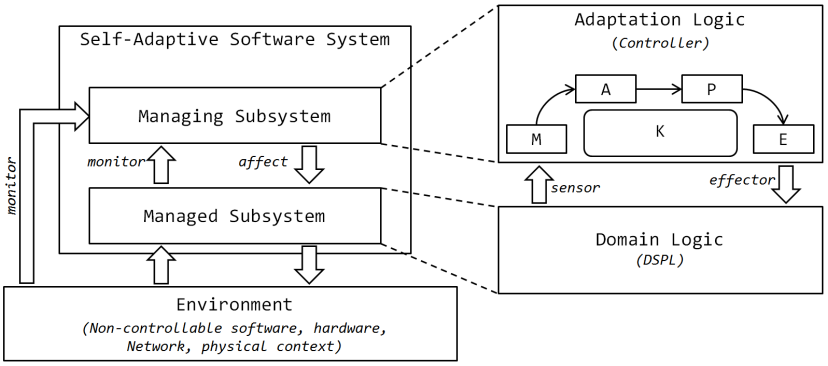

Figure 1: DSPL in the context of SASS.

3. A Taxonomy for DSPL

There are several design issues for implementing DSPL approaches, and for each approach there are different associated alternative strategies, called design solutions. The proposed taxonomy identifies the several common design issues for implementing DSPLs, and classifies their different design solutions. The taxonomy was developed based on previous studies that define taxonomies in the area of self-adaptation [20, 24, 25, 26] and DSPL [27, 28].

3.1. Dimensions of Taxonomy

Our proposed taxonomy classifies the design issues of a DSPL scheme into two dimensions: the first dimension is related to variability design issues, while the second dimension is related to self-adaptation design issues.

Variability dimension. This dimension was based on previous taxonomies [28, 27]. Bashari *et al.* [28] present a detailed adaptation taxonomy for DSPL, establishing an individual classification for each MAPE-K activity, that is: monitor, analyse, plan, and execute. Galster *et al.* [27] have provided a variability characterisation for software systems.

Using the taxonomies of Bashari *et al.* [28] and Galster *et al.* [27], we combine similar design issues and remove design solutions that are applicable only to SPL (static variability). Besides, design issues related to the self-adaptation process were reallocated to the self-adaptation dimension (Section 3.4), such as design issues and its design solutions for the MAPE-K loop.

After, the variability dimension was analysed using the DSPLs selected for validation (Section 3.2). This analysis also resulted in the refining of the set of design solutions and the addition of the design issue for feature modelling strategy and the platform to implement variability. Section 3.3 describes the variability dimension and applies it to the selected DSPLs.

Self-adaptation dimension. This dimension was based on previous taxonomies [20, 24, 25, 26]. McKinley *et al.* [24] have proposed a compositional adaptation taxonomy, comparing *how*, *when* and *where* the recomposition occurs. Later, Salehie and Tahvildari [25] proposed a taxonomy for self-adaptation, that relies on the object to adapt (*what*), realisation issues (*how*), temporal characteristics (*when*), and interaction concerns of adaptation (*where*), extending the taxonomies of Oreizy *et al.* [20] and McKinley *et al.* [24]. Andersson *et al.* [26] have presented an adaptation taxonomy for self-adaptive software systems, focusing on modelling dimension.

Using these previous taxonomies, we combine similar design issues and remove design solutions that are not applicable to DSPLs, generating a first version of the self-adaptive dimension of our taxonomy. Besides, we have added design issues related to the feedback control loop MAPE-K.

After, our initial self-adaptation dimension was analysed using the DSPLs selected for validation (Section 3.2). This analysis also resulted in the refining of the set of design solutions and the addition of the design issue for the architectural pattern used. Section 3.4 describes the self-adaptation dimension and applies it to the selected DSPLs.

3.2. Selection of DSPL Approaches

We have selected relevant approaches in the intersection of DSPLs and SASS. In particular, we have selected two types of systems: (i) SPLs that support dynamic variability, and (ii) self-adaptive systems that apply SPL techniques to manage variability at runtime. The selected DSPL approaches that support some form of dynamic variability are: Abbas *et al.* (ASPL) [12, 29], Abotsi *et al.* [30], Baresi *et al.* [31], Bencomo *et al.* (Genie) [32, 33], Casquina *et al.* (Cosmapek) [34], Cetina *et al.* (MoRE) [35, 36], Cubo *et al.* (Dynamic DAMASCo) [37, 38], Fuentes and Gamez [39, 40], Gomaa and Hashimoto [41], Hallsteinsen *et al.* (MADAM) [10, 42, 43], Lee *et al.* [44, 45], Morin *et al.* (DiVA) [46, 47, 11], Nascimento *et al.* (ArCMAPE) [13, 48], Parra *et al.* (CAPucine) [49, 50], Pessoa *et al.* [14, 51], Rosenmüller *et al.* [15, 1]. This list was defined based on two sources of information: (i) the systematic mapping study developed by the authors to identify approaches that include dependability attributes in DSPLs [52], and (ii) two surveys [8, 28] that identify DSPLs and analyse the use of the MAPE-K feedback control loop.

This previous systematic mapping study [52] reviewed papers about Dependable DSPLs selecting nine primary studies. In addition, we made a comparison of the primary studies regarding the MAPE-K loop activities and the DSPL dimension. The survey developed by Bencomo *et al.* [8] questioned the dynamism level of DSPL approaches compared to MAPE-K loop. They selected nine DSPL approaches and analysed whether each DSPL meet phases of feedback control loop at runtime (dynamic) or design-time (static). The survey presented by Bashari *et al.* [28] proposed a conceptual framework for comparing adaptation realisation in DSPL based on MAPE-K loop. They also compared seven DSPL approaches using the proposed framework.

Tables 1 and 2 provide a summary of the main aspects of the DSPL approaches presented in this section, showing the different strategies for each design issue of our taxonomy (Section 3). In order to compare the DSPL approaches, we assign weights to each design strategy, which reflect the contribution (negative, neutral, positive and

+positive) of that strategy in facilitating the construction of DSPL applications. A negative weight (-1) is attributed when a solution clarifies that it does not explicitly support a design issue or when the chosen design issue is a poor choice. However, in some situations, it is not possible to claim there is support or not a specific design issue, and in this case, we choose not to penalise the solution, indicating as 'Not specified' (–). A positive weight (+1) is attributed to the solution when a good design issue is chosen. And a +positive weight (+2) is attributed to the solution when it chooses the design issue considered the best choice. Adding up the weights of DSPL approach produces a final score that is an indication of the suitability of that implementation for the development of well structured DSPLs.

The reader should note that some design issues have mutually exclusive design strategies, while for others, the strategies have an additive character (these are marked with '+' in Tables 2 and 1). Thus, the highest score attainable by a approach is 38, obtained by adding the maximum possible scores for each design issue. Section 3.3 evaluates the selected approaches (Section 3.2) for each design issue included in the Variability Dimension (D1). Section 3.4 evaluates the same approaches considering the design issues of the Self-Adaptation Dimension (D2). Section 3.5 summarises our findings.

3.3. Variability Dimension

Table 1 presents the design issues and their respective design solutions of the variability dimension applied to the selected DSPLs, which are discussed below.

D1-A1. Variability Type. Static variability is performed by the compiler while dynamic variability is performed by the system itself at runtime. A DSPL approach must support at least dynamic variability. We classify the design approaches for supporting DSPL variability in two types: (i) *only dynamic variability*, and (ii) *both static and dynamic variabilities*. For the purpose of enhancing the structuring of DSPL systems, it is desirable to allow the support of both variability types. As a result, the support of both static and dynamic variability is assigned weight +2 while the support of only dynamic variability received +1.

D1-A2. Variability Conceptual Model. Although Kang and Lee [3] present only two variability models for SPLs, feature model and decision model, we also consider other strategies for the variability modelling raised by Galster *et al.* [27]. Thus, the variability can be represented as: (i) *feature model*, (ii) *decision model*, (iii) *change scenarios*, (iv) *profiles*, (v) *rules / conditions* and (vi) *variant labels / annotations*. The representation of variability as feature models is a classical approach used by most of the (D)SPL approaches. In the second scheme, decision model represents the variability as a set of decisions, commonly in tabular notation or textual notation. In the third scheme, change scenarios are modelled to describe events or options that trigger changes in the system. In the fourth approach, profiles are created to represent descriptive summaries of artefacts in the environment (as a table, model or a set of expressions). In the fifth approach, a set of rules or conditions is defined referring to elements or artefacts that realise the system in order to support variability. In the sixth approach, variant labels or annotations are added to artefacts that represent the DSPL.

Table 1: Summary of variability dimension.

DSPL Variability Dimension	Design Issue \ DSPL Approach	Abbas *et al.*	Abotsi *et al.*	Baresi *et al.*	Bencomo *et al.*	Casquina *et al.*	Cetina *et al.*	Cubo *et al.*	Fuentes & Gamez	Gomaa & Hashimoto	Hallsteinsen *et al.*	Lee *et al.*	Morin *et al.*	Nascimento *et al.*	Parra *et al.*	Pessoa *et al.*	Rosenmüller *et al.*
Variability Type	Dynamic	+1	+1	+1	+1		+1		+1	+1	+1		+1	+1	+1	+1	
	Static & Dynamic					+2		+2				+2					+2
Variability Conceptual Model	Feature model	+2	+2	+2	+2	+2	+2	+2	+2	+2		+2	+2	+2	+2	+2	+2
	Decision model																
	Change scenarios																
	Profiles																
	Rules/Conditions																
	Labels/Annotations										0						
Feature Modelling Strategy	FM only with dynamic features	0	0	0	0		0		0	0			0	0	0	0	
	Multiple models					+1		+1									+1
	Single model											+2					
Architectural Model	Unsupported		–							–					–		-1
	Customized lang.	+1		+1		+1	+1	+1			+1	+1		+1		+1	
	ADL				+1				+1				+1				
Architectural Style	Unsupported																-1
	Component-based	+1			+1				+1		+1					+1	
	Service-oriented			+1			+1					+1					
	Hybrid		+2			+2		+2		+2			+2	+2	+2		
Variability Managed Element	Code																+1
	Component	+1	+1		+1	+1		+1	+1	+1	+1		+1	+1	+1	+1	
	Service		+1	+1			+1	+1		+1		+1	+1	+1	+1		
	Aspect			+1					+1				+1		+1		
	Architecture																
Variability Traceability	Direct link	+2	–	–	+2	+2			–		–	+2	+2	+2	–	–	+2
	Traceability matrix						+1	+1		+1							
	Transform. rules																
	Score	8	7	7	8	11	7	11	7	8	4	11	11	10	8	6	6

– Not specified.

Kang *et al.* [3] consider the feature model as the most adequate variability representation due to its clearer semantics and graphical notation. Our choice of weights in Table 1 expresses this preference. The feature model technique is assigned weight +2 since it is a well-known technique for SPL development. Decision model and change scenarios techniques represent variability explicitly using textual notations, while profile and rules/ condition definitions represent variability in a more implicitly manner. The variant label/annotation technique promotes variability to be scattered and tangled in the source code, which decreases the system's modularity. We consider the use of techniques, such as (i) decision model, (ii) change scenario, (iii) profile definition, and (iv) rule/condition definition more robust than using the variant label technique. Correspondingly, these techniques have received weight +1 while the label/annotation technique received weight 0.

D1-A3. Feature Modelling Strategy. There are at least two design strategies for the DSPL feature representation: (i) *only dynamic feature modelling* and (ii) *static and dynamic features modelling*. In the first case, the feature model represents only dynamic features with dynamic binding, encompassing FODA[1]-like feature models, orthogonal variability models (OVM) and common variability language (CVL). In the second case, the mechanism supports both static and dynamic features at the same time. Most of the traditional notations of feature models emphasise only *variability in space*, having no special representation to define whether a variation point should be bound at design time or runtime. Thus, there are at least two design strategies for combining static with dynamic features and representing *variability in time*: (i) use of *multiple models* with a feature model to represent all (static and dynamic) features and a separate model to differentiate static and dynamic features or to represent its binding times; and (ii) use of a *single feature model* based on an extended notation to represent both static and dynamic features. We have assigned weight 0 to approaches which support only dynamic features, whereas weight +1 was assigned to those which support both dynamic and static features using multiple models, and weight +2 to those which support dynamic and static features using a single model.

D1-A4. Architectural Model. There are at least three design techniques for representing PLA architectures: (i) using *customised languages*, and (ii) using *ADL*. In the first case, some DSPL approaches define customised languages or ad-hoc models to represent the architectural model. In the second case, Architecture Description Languages (ADLs) are used to define the software architecture. ADL is any language used in an architecture description and it can be used by one or more viewpoints to represent identified system concerns within an architecture description [53]. This design issue has mutually exclusive strategies. Adopting design strategies for representing architectural models produce well-structured software systems. This justifies the positive weight assigned to this design issue and the negative weight assigned to the approach that did not offer any support.

[1] Feature-Oriented Domain Analysis [4].

D1-A5. Architectural Style. It refers to the highly granular entities of the system and how they are connected to each other [28]. There are at least three different architectural styles used by DSPL approaches: (i) *component-based style*, (ii) *service-oriented architecture style*, and (iii) *hybrid architectural style*. In the first approach, component-based architectures provide functionalities structured as architectural configurations composed of components and connectors. In the second approach, service-oriented architectures are based on services to provide systems' functionalities to service clients. In the third approach, hybrid architectural styles are based on components and services, using specifications such as Service Component-Architecture (SCA). We have assigned weight -1 to approaches which do not support any architectural style, whereas weight +1 was assigned to those which support some kind of architectural style; the hybrid approach was given weight +2.

D1-A6. Variability Managed Element. The variability managed element is the part of the system that changes when the variability is carried out. The variability can be attached to different levels of abstraction, such as: (i) *code*, (ii) *component*, (iii) *service*, (iv) *aspect*, and (v) *software architecture*. In the first case, the variability promotes changes in the code, which is generated, compiled and deployed at runtime. In the second case, the variability is attached to components, allowing the connection and disconnection of components. In the third case, the variability promotes changes at service level by allowing the disconnection and connection of services. In the fourth case, the dynamic variability is performed by a dynamic aspect weaving. In the last case, two or more architecture are compared and one is chosen to meet variability.

For the purpose of enhancing the structuring and dynamicity of DSPL systems, it is desirable to allow changes to be applied to different kinds of managed elements as many as possible. This design issue has an additive character.

D1-A7. Variability Traceability. It refers to the mapping between the variability and architectural models. There are at least three strategies for supporting traceability: (i) *direct link*, (ii) *traceability matrix*, or (iii) *transformation rules*. In the first approach, a direct link is defined between the variability elements and architectural elements, without the creation of a mapping element between both. For instance, using OVM to relate features to architectural elements. In the second approach, the traceability is performed by using a traceability matrix, such as a table or a mapping model, relating variability elements of variability to architectural elements. In the third approach, transformation rules define the mapping between variability elements and architectural elements. The direct link approach promotes well-structured DSPL easier to be changed and maintained since less complicated solutions can be developed. In order to reflect these qualities, we have assigned a weight +2 to approaches which support direct link traceability, whereas weight +1 was assigned to those which support some traceability technique using two separated models; approaches that did not specify or declare the traceability used did not receive a weight.

3.4. Self-adaptation Dimension

Table 2 presents the design issues and their respective design solutions of the self-adaptation dimension applied to the selected DSPLs, which are discussed below.

Table 2: Summary of self-adaptation dimension

Self-Adaptation Dimension		Design Issue \ DSPL approach	Abbas *et al.*	Abotsi *et al.*	Baresi *et al.*	Bencomo *et al.*	Casquina *et al.*	Cetina *et al.*	Cubo *et al.*	Fuentes & Gamez	Gomaa & Hashimoto	Hallsteinsen *et al.*	Lee *et al.*	Morin *et al.*	Nascimento *et al.*	Parra *et al.*	Pessoa *et al.*	Rosenmüller *et al.*
Adaptation Cause		Context	+			+		+	+	+	+	+	+	+	+		+	+
		System	+	+	+		+					+	+	+	+	+		
		User			+												+	
Adaptation Automation		Assisted			0													
		Autonomous	+1	+1		+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
Adaptation Binding Time		Design					+		+				+					+
		Load	+				+	+	+	+		+						+
		Runtime	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Adaptable Arquitetural Pattern		Not supported		-1	-1				-1				-1			-1	-1	
		Partly supported									0							
		Microkernel								+1								
		Reflection	+2			+2	+2	+2				+2		+2	+2			+2
Adaptation Realisation Type		Close		0	0		0	0		0	0				0	0	0	0
		Open	+1			+1			+1			+1	+1	+1					
Adaptation Realisation Technique		Replacement	+	+	+			+			+	+					+	+
		Reorganisation				+	+		+	+			+	+	+	+		
		Code generation																
Use of MAPE-K Pattern	Monitor	Realisation	+1	+1	0	+1	+1	+1	+1	0	+1	+1	+1	+1	+1	+1	+1	+1
		How & When	+1	0		+1	0	+1	0		+1	+1	+1	+1	0	+1	+1	+1
	Analyse	Realisation	+1	0	+1	+1	+1	0	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
		How & When	+1		0	0	0		+1	0	0	+1	+1	+1	0	0	0	0
	Plan	Realisation	+1	0	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
		How & When	+1		0	0	0	0	+1	0	0	+1	0	+1	0	0	-1	0
	Execute	Realisation	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
		How & When	+1	0	+1	0	0	0	+1	0	+1	+1	0	+1	+1	+1	+1	+1
	Knowledge Model Support	Model-free																
		Model-based (static)		+1	+1		+1	+1		+1	+1				+1	+1		+1
		Model-based (dynamic)	+2			+2			+2			+2	+2	+2			+1	
	MAPE-K Pattern	Not supported		-1	-1	-1			-1	-1			-1					
		Partly supported						0			0	0		0		0		0
		Fully supported	+1				+1								+1		+1	
		Score	20	5	7	13	14	12	14	9	11	19	13	18	14	10	12	15

D2-A1. Adaptation Cause. It refers to the source that triggers/initiates the adaptation process. Such sources can be represented as: (i) *context*, (ii) *system*, and (iii) *user*. Changes in *context* occur in the environment and are external to the system. The ability to react to context changes requires the capture of context information. Context information can be divided into the system context and user context [10, 42]. The system context information includes data related to the environment where the system is running, such as network, memory resources, battery level and consumption, and other computing resources. The user context information includes data such as position (based on GPS information in a smartphone, for instance), and environmental information in which the user is entered, such as light and noise. Changes in *system* occur internally, for instance, the failure of a component, the performance of a service, and exceptions. The ability to react to system changes requires specific sensors within the implemented system according to the change to be detected. The change can also be triggered by some *user* action while using the system and they refer to changes in user requirements or needs at runtime. This design issue has an additive character, and a approach could add up to 3 points.

D2-A2. Adaptation Automation. Automation refers to the degree of outside intervention during adaptation. Andersson *et al.* [26] define the automation as a degree, which can vary from (i) *assisted* to (ii) *autonomous*. In the first case, the adaptation is externally assisted, either by another system or by human intervention. In the second case, the adaptation process is fully automated with no external influence guiding how the system should adapt. Since autonomic computing promises that, increasingly, the adaptation processes should be completely autonomous, this justifies the neutral weight assigned to the assisted approach and the positive weight assigned to the autonomous adaptation technique.

D2-A3. Adaptation Binding Time. It defines the moment when the adaptation binding occurs. There are three different design solutions for the binding occurrence: (i) *at design time*, (ii) *at load time*, and (iii) *at runtime*. In general, SPLs implement binding at design time, while DSPLs can support the three bindings types, although DSPLs should support at least the binding at runtime. In addition, each DSPL approach can optionally support design and/or load binding time mechanisms. This design issue also has an additive character, and an approach could add up 3 points.

D2-A4. Adaptable Architectural Pattern. Here, we consider the architectural patterns for adaptive systems presented by Buschmann *et al.* [54], which are Microkernel and Reflection patterns. Hence, the use of adaptable architectural pattern can be: (i) *not supported* when the approach does not follow an adaptable architectural pattern; (ii) *partly supported* when the approach follows another architectural pattern that is not specifically designed for self-adaptive applications, such as the layer architectural pattern, that brings several benefits such as separation of concerns, maintainability, and reusability [54]; (iii) *microkernel*, when the approaches follows the Microkernel Architectural Pattern; or (iv) *reflection*, when the approaches follow the Reflection Architectural Pattern.

Transparency refers to whether an application is aware of the "infrastructure" needed for adaptation. Different degrees of transparency (concerning application source, virtual machine, and so on) can be implemented. The reflection pattern supports a high degree of transparency, while the microkernel pattern supports a lower degree. In order to reflect these qualities, we have assigned weight -1 to approaches which have not used any architectural pattern; and weight 0 to approaches which support another architectural pattern not specific to adaptive systems, whereas weight +2 was assigned to those which support the reflection pattern, and weight +1 was given to those approaches which applied the microkernel pattern.

D2-A5. Adaptation Realisation Type. According to Salehie and Tahvildari [25], there are two different types of adaptation: (i) *close adaptation*, and (ii) *open adaptation*. In the first approach, the system has only a fixed number of adaptive options, and no new behaviours and alternatives can be introduced at runtime. In the second, the system can be extended, and new alternatives can be added at runtime. As a consequence, new behaviour and even new adaptable elements can be introduced into the system for use by adaptation mechanism. We have assigned a neutral weight when the approach implements close adaptation; and a positive weight when the approach supports open adaptation.

D2-A6. Adaptation Realisation Technique. It is related to the implementation approach to perform the adaptation. There are at least three different design solutions for implementing the adaptation: (i) *replacement*, (ii) *reorganisation of the architecture*, and (iii) *code generation*. In the first approach, adaptation is achieved by replacing one element with another with a same interface, without affecting the rest of the system architecture. In the second approach, a reconfiguration of architecture is performed when the adaptation occurs, reorganising the architectural structure. In the third approach, the adaptation is performed by generating, compiling and deploying a new portion of source code in order to change or fix the system behaviour. This design issue also has an additive character, and a approach could add up 3 points.

D2-A7. Use of the MAPE-K Pattern. This requirement should consider at least three design issues: (i) whether or not the four activities of the MAPE-K are present in the adaptation mechanism implementation as individual functionalities (*realisation of individual MAPE-K activities*), (ii) to what extent the knowledge base supports the representation of models (*knowledge model support*, and (iii) whether or not the approach adheres to *MAPE-K pattern*, as shown in Table 2.

The first aspect identifies whether or not the implementation of the monitor, analyse, plan, and execute functionalities are supported. Each functionality can be classified as: (i) *not implemented*, or (ii) implemented. Moreover, when the activity is implemented, one should consider how the set of adaptation options is defined and when the option binding is performed. These options could be: (i) *statically defined + design time binding*: statically defined at design time and it could not be changed during runtime execution, that is, the static binding is performed, (ii) *statically defined + runtime binding*: statically defined at design time but the option binding is performed at runtime, and (iii) *dynamically defined + runtime binding*: the set of options is dynamically defined, in the sense that new options could be included/removed during runtime,

and the option binding is also performed at runtime. For each activity (Monitor, Analyse, Plan and Execute), we have assigned a neutral weight when the functionality was not implemented and a positive weight when it was implemented. When the activity is implemented, we have assigned a negative weight (-1) when the set of decisions is statically defined at design time, and their binding cannot be changed at runtime; a neutral weight (0) when the set of decisions is statically defined at design time but their binding is realised at runtime; and a positive weight (+1) when the set of decisions is dynamically defined and their binding is also realised at runtime.

For the second design issue, the knowledge base can be (i) *model-free* or (ii) *model-based*. In the first approach, the knowledge base has no predefined model for the system or the environment. In the second approach, the knowledge base uses a model of the system and its context. Two kinds of models exist according to the adaptation realisation type: (i) *static models* or (ii) *dynamic models*. We have assigned a neutral weight when the approach is model-free. When the approach supports the representation of knowledge models (model-based), we have assigned a +1 weight when the approach implements static models; and a +2 weight when the approach supports dynamic models.

On the one hand, static models are used by systems with close adaptation and model-based knowledge. Static models cannot be extended at runtime, that is, the set of adaptation options are defined at design time, and they cannot be changed at runtime. One of these options is chosen at runtime by the analyser and/or the planner components. On the other hand, dynamic models are used by systems with open adaptation and model-based knowledge. Dynamic models incorporate a set of adaptation options that can be changed at runtime by including new options or removing existing ones. The option to be executed is also chosen during runtime by the analyser and planner components.

The third aspect refers to whether or not a control loop pattern is supported. The control loop pattern can be: (i) *not supported* when the approach does not implement the feedback loop; (ii) *partly supported* when the control loop is structured in an ad-hoc manner; or (iii) *fully supported* when the approach explicitly applies the control loop pattern. We have assigned weight +1 for full support of control loops, weight 0 for partial support, and weight -1 for no support.

3.5. Summary

As much as possible, DSPL approaches should be highly adaptable, dynamic, reliable and simple. In spite of these requirements, the previous discussions revealed that several decisions taken in the design of the studied DSPL approaches resulted in solutions which are too inflexible, static and complex. Ranking the studied approaches according to their final score (Table 3), out of the maximum of 38, we have: Morin*et al.* (29); Abbas *et al.* (28); Casquina *et al.*, Cubo *et al.* and Nascimento *et al.* (25); Lee*et al.* (24); Hallsteinsen *et al.* (23); Bencomo *et al.* and Rosenmüller *et al.* (21); Cetina *et al.* and Gomaa and Hashimoto (19); Parra *et al.* (18); Fuentes and Gamez (16); Baresi *et al.* (14); and Abotsi *et al.* (12). Tables 1and 2 present, in a summarised fashion, both the positive and negative design issues of each approach, allowing the software engineer to compare different approaches and evaluate potential difficulties and the impact of a given choice in the construction of DSPLs. Therefore, Tables 1and 2 are meant

Table 3: Summary of dimensions.

Dimension \ DSPL Approach	Abbas *et al.*	Abotsi *et al.*	Baresi *et al.*	Bencomo *et al.*	Casquina *et al.*	Cetina *et al.*	Cubo *et al.*	Fuentes & Gamez	Gomaa & Hashimoto	Hallsteinsen *et al.*	Lee *et al.*	Morin *et al.*	Nascimento *et al.*	Parra *et al.*	Pessoa *et al.*	Rosenmüller *et al.*
Self-Adaptation	20	5	7	13	14	12	14	9	11	19	13	18	14	10	12	15
DSPL Variability	8	7	7	8	11	7	11	7	8	4	11	11	10	8	6	6
Final Score	28	12	14	21	25	19	25	16	19	23	24	29	24	18	18	21

primarily as a guide for decision-making rather than an absolute measurement of the suitability of a given approach.

4. General Design Criteria

The taxonomy developed in Section 3 identifies several design issues, which should be taken into account while designing a DSPL. The design decisions should be taken according to the demanding quality attributes.This section outlines the main criteria that can be followed by software engineers to build effective DSPLs. Based on these criteria, we identify the choices for designing an ideal DSPL approach.

4.1. Quality Attributes

Q1. Dynamicity. Dynamicity is a system's ability to undergo changes and adapt at runtime. Dynamicity can be measured as the degree of a system to self-adapt to runtime, reacting to foreseen, foreseeable, and unanticipated changes in the most autonomous way possible.

Q2. Autonomy. Autonomy is the degree of external intervention during adaptation [26]. Autonomous systems operate without the direct intervention of humans or others systems, and have some kind of control over their actions and internal state [55].

Q3. Flexibility. Flexibility is the degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in contexts beyond those initially specified in the requirements [56]. Flexibility can be achieved by adapting a system for additional user groups, tasks and cultures [56]. Flexibility enables systems to take account of circumstances, opportunities and individual preferences that might not have been anticipated in advance [56]. If a system is not designed for flexibility, it might not be safe to use the product in unintended contexts [56].

Q4. Performance. Performance is the degree of a system or component to accomplish its designated functions within given constraints, such as speed, accuracy, or memory usage [57]. Performance can be a measure of a system's ability to perform its functions, including response time, throughput, and number of transactions per second [58]. Performance is an indication of the system responsiveness to execute any action within a given time interval. It can be measured in terms of latency or throughput. Latency is the required time to respond to any event. Throughput is the number of events that occur in a period of time.

Q5. Complexity. Complexity establishes the degree to which a system's design or code is difficult to understand because of numerous components or relationships among components [57]. Complexity is a measurement of how many more computing resources the approach of a problem requires as the problem grows in number of variables [58]. Feedback control loop and adaptation pattern are examples of techniques to cope with and reign in the management complexity of dynamic systems [59].

Q6. Transparency. Transparency refers to whether an application or system is aware of the infrastructure needed for recomposition [24]. Different degrees of transparency determine both the proposed approach's portability across platforms and how easily it can add new adaptive behaviour to existing programs [60].

Q7. Separation of Concerns. This principle is used to deal with the complexities that exist in the definition and use of software systems [61]. Separation of concerns can be understood as the principle of software design that the source code be separated into layers and components that each have distinct functionality with as little overlap as possible [62].

Q8. Modularity. The modularity is related to the degree to which a system or computer program is composed of discrete components so that a change to one component has minimal impact on others [57]. Modularity is the ability of a system to be composed of separate, interchangeable components, each of which accomplishes one function and contains everything necessary to accomplish this. Modularity increases cohesion and reduces coupling and makes it easier to extend the functionality (modifiability) and maintain the system (maintainability). Cohesion is the manner and degree to which the tasks performed by a single software module are related to one another [57]. Coupling is the manner and degree of interdependence between software modules [57].

Q9. Modifiability. Software modifiability refers to a measure of how easy it may be to change an application to cater for new functional and nonfunctional requirements [63]. The modifiability of a system is improved as system modularity improves. Modifiability can be understood as a design quality attribute that composes maintainability.

Q10. Reusability. The the reusability is the degree to which an software asset can be used in more than one software system, or in building other assets [57]. Software reusability is the use of existing software assets in some form within the software product development process. Reusing the code of adaptation logic should be possible. Ideally, the adaptation logic should be defined independently of the application logic, so that adaptation logic components can be reused in distinct domains.

Q11. Testability. The effort required to test software is called testability [57]. Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing [64]. In general, system testability relates to several structural issues [64]: separation of concerns, the level of documentation, and the degree to which the system uses information hiding. In the ideal world, it should not be difficult to test every adaptation scenario in a systematic manner. However, testability is affected by the dynamism of the self-adaptive system, as it increases the degree of unpredictability. Non-dynamic systems can be tested at design time to reduce the occurrence of unforeseen situations. Whereas dynamic systems are prepared to anticipate the changes that may occur at runtime, changes at runtime in self-adaptive systems may have different degrees of anticipation [5]: foreseen (taken care of), foreseeable (planned for), and unforeseen (not planned for).

Q12. Reliability. Reliability is the capability of the software product to maintain a specified level of performance when used under specified conditions [57]. Reliability is the ability of a system to remain operational over time, and how the system behaves in varying circumstances. Performing tests and correcting failures result in improved reliability.

4.2. An Ideal DSPL Approach

After describing the quality attributes in the previous section, we proceed by discussing each design issue and present an ideal model for developing DSPLs, considering both the self-adaptive dimension and the variability dimension. We discuss how each design choice can affect the quality attributes while designing the DSPL. Table 4 shows a summary of our findings. A '+' in the Table 4 indicates a positive relationship between a design alternative and a quality attribute, that is, the use of the design alternative helps in the achievement of the quality goal. A '−' in the table indicates the opposite situation. Finally, a blank cell indicates that, depending on the context of use, it could have a negative or positive effect. Both design issues and quality attributes are abstract, and the scores of the tables are meant to guide developers in their choices among design alternatives, and identify possible design conflicts. The interactions among design alternatives and quality attributes can be complex, and these design choices should be understood in the context of other design decisions. Trade offs are also discussed in this section since the quality attributes can be conflicting to each other.

4.2.1. Variability Dimension (D1)

D1-A1. Variability Type. A DSPL approach should support at least the dynamic variability. From the viewpoint of developing well-structured DSPLs, it seems to be reasonable the approach to provide support for both static and dynamic variability since part of the software variability could be decided during design time. Thus, an ideal DSPL approach should deal with static variability and dynamic variability.

D1-A2. Variability Conceptual Model. An ideal DSPL approach should represent the variability as feature model. Pohl *et al.* state that the explicit representation of the software variability has significant advantages as the improvement of modifiability,

Table 4: Taxonomy design issues *versus* quality attributes.

Dimension		Design Issue \ Quality Attributes	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Variability Dimension	Variability Type	Dynamic												
		Static & Dynamic				+	+						+	+
	Variability Conceptual Model	Feature model								+	+	+		
		Decision model								+	+			
		Change scenarios								+	+			
		Profiles								+	+			
		Rules / Conditions								+	+			
		Variant labels / Annotations								–		–		
	Feature Modelling Strategy	FM only with dynamic features												
		Multiple models					–				–			
		Single model					+				+	+		
	Architectural Model	Unsupported					–			–	–	–	–	–
		Custom languages									+	+	+	
		ADL									+	+	+	
	Architectural Style	Unsupported												
		Component-based												
		Service-oriented												
		Hybrid			+									
	Variability Managed Element	Code			+						+			
		Component			+						+			
		Service			+						+			
		Aspect			+						+			
		Architecture			+						+			
	Variability Traceability	Direct link					+				+	+		
		Traceability matrix					–				–	–		
		Transformation rules					–				–	–		
Self-Adaptation Dimension	Adaptation Cause	Context	+		+	–	–						–	–
		System	+		+	–	–						–	–
		User	+		+	–	–						–	–
	Adaptation Automation	Assisted		–										–
		Autonomous		+	+		–				–		–	–
	Adaptation Binding Time	Design			+		–							
		Load			+		–							
		Runtime			+		–							
	Adaptable Arquitetural Pattern	Not supported												
		Partly supported												
		Microkernel pattern			+	–		–	+	+	+			
		Reflection pattern			+	–		+	+	+	+	+		
	Adaptation Realisation Type	Close	–		–		+				+		+	+
		Open	+		+	–	–						–	
	Adaptation Realisation Technique	Replacement			–		+				+	+	+	
		Reorganisation			–	–				+	+	+	+	
		Code generation			+	–				–		–	–	–
	Use of MAPE-K Pattern: Activity Realisation	Not implemented												
		Statically + Design time binding	–		–									
		Statically + Runtime binding	–		–									
		Dynamically + Runtime binding	+		+	–	–						–	–
	Use of MAPE-K Pattern: Knowledge Model Support	Model-free												
		Model-based (static)			–						+		+	+
		Model-based (dynamic)	+		+	–	–				+		+	+
	Use of MAPE-K Pattern: MAPE-K Pattern	Not supported												
		Partly supported												
		Fully Supported	+							+	+	+	+	+

modularity and reusability [65]. The decision model, change scenario, profile definition, and rule/condition techniques represent variability in a more implicitly manner when compared to the feature model scheme, as a consequence, their use can affect negatively the traceability quality attribute. However, these techniques can support some variability modularity and modifiability since variability changes can be located in few places. We consider that performing changes to a single component is better than making a widespread change to the system. The variant label/annotation technique promotes variability to be scattered and tangled in the source code, which can decrease the system's modularity and variability traceability and reusability.

D1-A3. Feature Modelling Strategy. Considering that variability should be represented explicitly as feature models (*D1-A2* paragraph), and ideally DSPLs should support both static and dynamic variability (*D1-A1* paragraph), then the representation of both static and dynamic features is the design choice to be naturally taken. The single model scheme as a design alternative has some advantages by promoting the creation of simpler systems that are easier to be modified, traced and reused. On the other hand, the primary and secondary model strategy can promote the creation of more complex systems, it becomes more difficult to be maintained and more difficult to achieve variability traceability.

D1-A4. Architectural Model. An ideal approach for DSPL should adopt a design strategy for representing architectural models in order to benefit from all the advantages of a well-known architecture-centric approach [66], that is, yield systems easier to modify, reuse, test, and trace. The non-use of architectural model is an design alternative that can have a negative effect on the system's complexity, modifiability, traceability, modularity, reusability, testability, and reliability.

D1-A5. Architectural Style. The three main styles considered in our study are: the component-based style, the SOA style, and the hybrid approach which combines both styles. An ideal DSPL approach should follow a specific architectural style, either component-based or SOA style, however, the use of the hybrid approach provides support to more flexible DSPLs.

D1-A6. Variability Managed Element. The variability managed element is the part of the system that changes when the variability is carried out. An ideal approach should provide a multi-level attachment of variability since it is important for DSPL developers/architects to be able to attach variability at different levels of system structure. This design alternative improves the system's flexibility and modifiability.

D1-A7. Variability Traceability. It refers to the mapping between the variability and architectural models. Ideally, a DSPL approach should use the direct link approach between variability elements and architectural elements, without the creation of a mapping artefact between both. This design choice improves modifiability, changes traceability, understanding, and readability since the system is less complex. Besides, direct links between the artefacts can facilitate systematic and consistent reuse [65]. The use of traceability matrix or transformation rule implies in the creation of an extra artefact to define the mapping between variability and architectural elements, which can cause

a negative effect on the system's complexity. Moreover, the traceability matrix technique represents more explicitly the mapping when compared to the transformation rule technique.

4.2.2. Self-adaptation Dimension (D2)

D2-A1. Adaptation Cause. Some of the required properties for DSPL, prescribed by Hallsteinsen *et al.* [67], are: context and situation awareness, unexpected changes, and changes by users, such as functional or quality requirements. Ideally, a DSPL approach should react to the three kinds of adaptation causes (context, system, and user). As a consequence, the approach would be more dynamic, complete, and flexible. However, the implementation of the three strategies in the same system can cause a negative impact on the system's complexity, testability, performance, and reliability. In order to support context and system changes, the system should support different kinds of sensors according to the information to be collected. Also, the volume of information could be very high, meaning that the performance of the system could be compromised. Moreover, since context, system and user changes occur at runtime when a system implements all strategies, this approach could also affect negatively the system's testability and reliability.

D2-A2. Adaptation Automation. According to Hallsteinsen *et al.* [67], DSPLs should include autonomic capabilities. This design alternative improves system's flexibility, however, it can increase the system's complexity, and, as a consequence, the system can be more difficult to be tested, harder to be maintained and less reliable. Each of the predicted situations should be supported by the system, and unpredicted situations should be handled properly during runtime. When the adaptation process is assisted by humans or other systems, the approach can become less autonomous and less reliable since software adaptation, when performed by humans or other systems, becomes an onerous (regarding time, effort, and money) and error-prone activity, mainly due to involuntary injection of uncertainties by developers [68, 25].

D2-A3. Adaptation Binding Time. Runtime binding is required for DSPLs. Hallsteinsen *et al.* [67] also indicate the necessity of supporting other kinds of bindings in order to ease maintenance throughout their life cycle. So, an ideal approach should realise the three types of binding time (design time, load time and runtime). This design alternative can improve system's flexibility, however, it can increase system's complexity since a high number of variation points are going to be specified in the system.

D2-A4. Adaptable Architectural Pattern. Reflection and microkernel are well-known patterns recommended to build adaptive systems [54]. We consider that an ideal approach should apply the reflection architectural pattern since it supports transparency because the base level (managed subsystem) is unaware of the meta-level (managing subsystem). Changing a software system is easy since the metaobject protocol provides a safe and uniform mechanism for changing software [54].

The use of the microkernel pattern also improves modifiability, and flexibility since additional adaptations only requires the addition or extension of internal elements [54]. Also, this pattern provides support for separation of concerns and modularity since

external elements can implement their own policies. However, transparency is not fully satisfied since the use of a metaobject protocol is not required.

A disadvantage of using both patterns is a possible negative impact on the system's performance. Reflective software systems are usually slower than non-reflective systems, as a consequence of the complex relationship between the base level and the meta level [54]. Microkernel systems can be slower when compared to monolithic systems due to their support for flexibility and modifiability.

D2-A5. Adaptation Realisation Type. The design decision of using open adaptation has a number of advantages since the system can be extended, and new alternatives and behaviours can be added at runtime. It leads to better dynamicity and flexibility; however, the runtime discovery mechanisms can be complex, and they can impact negatively on the system's performance and testability. On the other hand, the close adaptation strategy leads to more reliable and testable systems since the system has only a fixed number of adaptive options, and no new behaviours and/or alternatives can be introduced at runtime. Moreover, it makes the system simpler and easier to maintain, although, it impacts dynamicity and flexibility negatively.

D2-A6. Adaptation Realisation Technique. An ideal model for DSPL should allow reconfigurations of the architecture structure when adaptations occur. This design alternative promotes an architecture-centric adaptation approach at a higher level of abstraction than code, based on coarse-grained architectural elements, realised by components and/or services. This design decision also promotes the construction of modular software systems, which in turn improves modifiability, reusability, and testability. However, it can lead to a worse performance and flexibility since on-the-fly adaptations can be complex and fine-grained modifications are not supported.

The replacement technique is simpler and less flexible than architecture reorganisation since the components can be changed individually at runtime, but no modifications to the architectural configuration are carried out. The use of this technique can make the system easier to reuse, test, trace and modify since it reduces the number of possible applications scenarios and limits customisability of DSPLs [1]. This strategy is also based on changes of coarse-grained elements when compared to the code generation technique, which promotes fine-grained modifications.

In this context, one can claim that the code generation technique can lead to more flexible systems, however, it becomes less reliable, less modular and less reusable. Moreover, this technique can also impact negatively on performance and testability since code generation, compilation, and loading are performed during runtime. However, a hybrid approach also could be adopted by the architect, combining coarse-grained and fine-grained adaptation realisation techniques.

D2-A7. Use of MAPE-K Pattern. Ideally, a DSPL should implement all four activities of the MAPE-K control loop. Moreover, it should be possible for the set of decisions to be dynamically defined and their binding be realised at runtime. This strategy can lead to more dynamic and flexible systems since options can be included/modified/removed during runtime. However, it can impact negatively the complexity, testability, reliability and performance. The design choice using the dynamically defined adaptation

options with runtime binding is more dynamic and flexible when compared to the most commonly used design choice in DSPLs, which is the statically defined with runtime binding [8].

An ideal model should support a model-driven approach for representing knowledge models at runtime. Cheng *et al.* [69] argue that models at runtime support the development of runtime assurance strategies. All studied approaches have realised model-based approaches for designing the knowledge base. It leads to approaches which are easier to test, more reliable, and easier to modify. Moreover, ideally, dynamic models should be supported in order to improve the system's dynamicity and flexibility. However, this design alternative can impact negatively the complexity and performance since the execution of runtime adaptations is not a simple task and the correctness of the dynamically modified models should be validated. The use of static models solves part of these limitations during design time; however, the approach becomes less flexible, but more reliable.

Lemos *et al.* [70] consider that the use of feedback control loop patterns is one of the cornerstones for developing self-adaptive software systems since they can be associated to different kinds of assurances at the conceptual level and at the code level. It leads to approaches which are more modular and reliable, easier to modify, test, and reuse.

5. Research Challenges

Despite many improvements in mechanisms to support runtime adaptation, the full potential of building dynamic software product line systems depends on advances in other research fields.

Testing and Assurance. self-adaptive software systems (SASS) can take different configurations at runtime in response to changes in context, in the system itself or its goals. Considering scenario variations, decision making and binding at runtime, performing the test and validation tasks becomes very complex. This complexity is further extended in systems that perform open adaptation because they can introduce new behaviours and new options at runtime, such as systems that perform service discovery at runtime.

In the context of DSPLs, Metzger and Pohl [71] claim that to address (context) situations unknown during design time, quality can be only partially assured and under certain assumptions. The authors raise the following issues as challenges: how to model such assumptions? How to check if the assumptions hold in the actual situation? How to ensure the quality of an application during runtime if not all potential adaptations of the application are known and predefined?

There are several researches in model-based testing for SPL as listed in Razak *et al.* [72]. However, these surveys only consider static variability, and may not be able to test situations that might occur at runtime. In the context of DSPLs, Santos *et al.* [73] address the formal verification of DSPL, proposing a formal structure, which specifies the DSPL adaptive behaviour to reason about the adaptations that could be triggered at runtime; however, their approach is focused on DSPL verification at design time. Other two works, A-FTS [74] and DFPN [75], were proposed for models supporting

and model checking of DSPL, focusing on execution states ("ready", "wait"). Ongoing work is the Devasses project [76] which has as one of its objectives to deal with model-based online testing, aiming to research, define and implement a solution to employ model-based testing techniques to SOA orchestrations at runtime.

Fault Tolerance and Reliability.. In a previous work [52], we also compared DSPL approaches regarding dependability and fault tolerance. Most of the DSPLs analysed in this study did not have as one of their objectives the improvement of dependability or the ability to tolerate failures. However, Nascimento *et al.* [13] deal with the highest amount of dependability attributes, exploring the different software fault tolerance techniques based on design diversity.

Besides, there are some approaches dealing with variability-aware reliability analysis techniques that can be applied in DSPLs, as [77, 78]. For instance, Rodrigues *et al.* [78] proposed a model for feature-aware discrete-time Markov chains, called FDTMC, for verifying probabilistic properties (e.g., reliability and availability) of product lines. Among the DSPL approaches analysed in our study (listed in Section3.2), only Pessoa *et al.* [14] deal with reliability. Pessoa *et al.* [14] proposed a DSPL approach that is explored and evaluated in the medical area, in particular in the Body Sensor Network domain, in which reliability and maintainability are key requirements. The proposed approach by Pessoa *et al.* [14] is based on FDTMC [78].

A Uniform Solution for Modelling Static and Dynamic Variability. Bosch *et al.* [79] list dynamic variability and runtime concerns as one of the trends of software variability area. According to Bosch *et al.* [79], designing variation points such that the variability mechanism, which determines the binding time, can be easily replaced during system implementation is particularly important. In this context, we pose another challenge: How to represent dynamic variability efficiently?

Despite the numerous notations for modelling variability using feature model, most of them deal only with static variability. The modelling of dynamic variability has been neglected, with few attempts to represent it, as showed in Table 1. Among sixteen analysed approaches (Table 1), only four of them deal with both static and dynamic variabilities [1, 34, 38, 44], using their own notations, as Lee *et al.* [44], or multiple models in order to complement the feature model. Thus, one of the challenges is the definition of a standard notation to represent both static and dynamic variabilities.

Reference Models, Reference Architectures, Implementation Frameworks and Automated Tools. In the context of DSPLs, there is no reference model to be followed on how to build a DSPL. Salehie and Tahvildari [25] raised the following issues that are applicable to the DSPL project: which architecture styles and design patterns are appropriate for this purpose? Which component model provides the best support for the sensing and effecting in vivo mechanisms? Which interfaces and contracts need to be considered?

On architectural styles, our taxonomy concluded that DSPLs use both hybrid, component-based, and service-oriented architectures. On architectural patterns, DSPLs are mostly based on the reflection architectural pattern. However, there is a gap between the definition of architectural styles and patterns and the concrete realisation of

DSPLs. There is still a lack of models and processes that guide the development of DSPLs, applying these architectural styles and patterns.

Regarding the component model, at the level of the managing subsystem, it is necessary to build monitoring and adaptation components that are reusable in other DSPLs, that is, they are domain independent. At the managed subsystem level, a DSPL needs to be constructed in order to be "understood" by the controller, to expose important information through sensors, and expose its variation points to facilitate variability management by effectors.

Regarding interfaces and contracts, although numerous research efforts have investigated approaches to develop DSPLs, there is still a lack of reference models and reference architectures that could help realising in a systematic manner adaptation processes, variability management, and the instrumentation of probes/effectors.

Based on the sixteen DSPL approaches (Table 1) from our study, we identified the following challenges: how to facilitate the technologies selection that fits in a given purpose? Bashari *et al.* [28] proposed a conceptual reference framework for DSPL, but they do not address practical issues for developing and realising DSPLs. There is still a gap in reference models and reference architectures that can serve as a basis or guide the design and construction of DSPLs. Similarly, implementation frameworks and automated tools are needed as means of supporting and instrumenting the process of building DSPL, and to make them widely usable by the industry.

Feedback Control Loops. Metzger and Pohl [71] state that the use of the autonomic computing concepts is a challenge in the DSPL development. In particular, the application of feedback control loops when building well-structured DSPLs. Also, Capilla *et al.* [80] discuss the need to explore mechanisms for compositional adaptation [24] in order to implement runtime variability in DSPLs, in particular, the adoption of MAPE-K loop. Moreover, Lemos *et al.* [70] discuss the importance of using the feedback control loops in order to obtain assurances [81].

According to Table 2, few approaches have applied the MAPE-K loop, for example. More specifically, only four approaches [12, 34, 13, 14] have used the MAPE-K feedback control loop for structuring the managing subsystems. The adoption of feedback control loops contributes to raising the maturity of DSPLs.

6. Conclusions

The trend in the development of new Dynamic software product line (DSPL) approaches indicates the combined use of self-adaptation and variability management. However, for that to be achieved there is the need to define a clear taxonomy that allows to compare existing approaches from dynamic software product line and self-adaptive software systems.

This chapter has provided a comprehensive taxonomy for comparing DSPL approaches regarding two key dimensions, namely, self-adaption and variability. Using our taxonomy, sixteen prominent DSPL approaches were compared and weighted to evaluate their design decisions, aiming to obtain a better understanding of the research area. Then, these design decisions were analysed in the face of quality attributes. Thus, as key messages of this chapter we realise that: (i) there is no single way to design and

develop DSPLs, and this chapter demonstrated, based on the taxonomy, that there are a variety of design decisions taken by the DSPL approaches, (ii) in the same way, each design decision has trade-offs when analysed concerning the quality criteria, and this analysis guides the architectural decisions of the engineers in the development of the DSPL, (iii) the taxonomy presented in this chapter supports the reuse of knowledge from the SASS research, which is a promising way to develop more dynamic DSPLs, and (iv) both the DSPL and SASS research areas can take advantage of the intersection of these two areas, providing more dynamicity and autonomy for the DSPLs and providing more means to represent the variability for adaptation in SASS.

References

[1] M. Rosenmüller, N. Siegmund, S. Apel, G. Saake, Flexible feature binding in software product lines, Automated Software Engineering 18 (2) (2011) 163–197.

[2] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software: Practice and Experience 35 (8) (2005) 705–754.

[3] K. C. Kang, H. Lee, Variability Modeling, in: R. Capilla, J. Bosch, K.-C. Kang (Eds.), Systems and Software Variability Management: Concepts, Tools and Experiences, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, Ch. 2, pp. 25–42.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. rep., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990).

[5] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle, Software Engineering for Self-Adaptive Systems: A Research Roadmap, in: Software Engineering for Self-Adaptive Systems, Vol. 5525 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 1–26.

[6] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw, Engineering self-adaptive systems through feedback loops, in: B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, Vol. 5525 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 48–70.

[7] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[8] N. Bencomo, J. Lee, S. Hallsteinsen, How dynamic is your Dynamic Software Product Line?, in: 4th International Workshop on Dynamic Software Product Lines (DSPL), 14th International Software Product Line Conference (SPLC 2011), 2010, pp. 61–68.

[9] C. Cetina, V. Pelechano, Variability in Autonomic Computing, in: R. Capilla, J. Bosch, K.-C. Kang (Eds.), Systems and Software Variability Management: Concepts, Tools and Experiences, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, Ch. 17, pp. 261–267.

[10] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, Using Product Line Techniques to Build Adaptive Systems, in: 10th International Software Product Line Conference (SPLC'06), IEEE, 2006, pp. 141–150.

[11] B. Morin, O. Barais, G. Nain, J.-M. Jezequel, Taming Dynamically Adaptive Systems using models and aspects, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, Vancouver, BC, 2009, pp. 122–132.

[12] N. Abbas, J. Andersson, Architectural reasoning for dynamic software product lines, in: Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops, ACM Press, New York, New York, USA, 2013, p. 117.

[13] A. S. Nascimento, C. M. Rubira, F. Castor, ArCMAPE: A Software Product Line Infrastructure to Support Fault-Tolerant Composite Services, in: 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, IEEE, 2014, pp. 41–48.

[14] L. Pessoa, P. Fernandes, T. Castro, V. Alves, G. N. Rodrigues, H. Carvalho, Building reliable and maintainable Dynamic Software Product Lines: An investigation in the Body Sensor Network domain, Information and Software Technology 86 (2017) 54–70.

[15] M. Rosenmüller, N. Siegmund, G. Saake, S. Apel, Code generation to support static and dynamic composition of software product lines, in: Proceedings of the 7th international conference on Generative programming and component engineering - GPCE '08, ACM Press, New York, New York, USA, 2008, p. 3.

[16] J. van Gurp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: Proceedings Working IEEE/IFIP Conference on Software Architecture, IEEE Comput. Soc, 2001, pp. 45–54.

[17] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. GÃ¶schka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezza, C. Prehofer, W. Schafer, R. Schlichting, D. Smith, J. Sousa, L. Tahvildari, K. Wong, J. Wuttke, Software engineering for self-adaptive systems: A second research roadmap, in: R. de Lemos, H. Giese, H. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II, Vol. 7475 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 1–32.

[18] D. Weyns, S. Malek, J. Andersson, FORMS: a FOrmal Reference Model for Self-adaptation, in: Proceeding of the 7th international conference on Autonomic computing - ICAC '10, ACM Press, 2010, p. 205.

[19] J. Kramer, J. Magee, Self-managed systems: An architectural challenge, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268.

[20] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (3) (1999) 54–62.

[21] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli, A survey of autonomic communications, ACM Trans. Auton. Adapt. Syst. 1 (2) (2006) 223–259.

[22] J. L. Hellerstein, Y. Diao, S. Parekh, D. M. Tilbury, Feedback Control of Computing Systems, John Wiley & Sons, 2004.

[23] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K. M. Göschka, On Patterns for Decentralized Control in Self-Adaptive Systems, Lecture Notes in Computer Science 7475 (2013) 76–107.

[24] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, Composing adaptive software, Computer 37 (7) (2004) 56–64.

[25] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and Research Challenges, ACM Transactions on Autonomous and Adaptive Systems 4 (2) (2009) 1–42.

[26] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Modeling Dimensions of Self-Adaptive Software Systems, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 5525 LNCS, Springer Berlin Heidelberg, 2009, pp. 27–47.

[27] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in Software Systems – A Systematic Literature Review, IEEE Transactions on Software Engineering 40 (3) (2014) 282–306.

[28] M. Bashari, E. Bagheri, W. Du, Dynamic Software Product Line Engineering: A Reference Framework, International Journal of Software Engineering and Knowledge Engineering 27 (02) (2017) 191–234.

[29] N. Abbas, J. Andersson, W. Löwe, Autonomic Software Product Lines (ASPL), in: Proceedings of the Fourth European Conference on Software Architecture Companion Volume - ECSA '10, Vol. 46, ACM Press, New York, New York, USA, 2010, pp. 324–331.

[30] K. S. Abotsi, S. T. Kurniadi, H. I. Alsawalqah, D. Lee, A software product line-based self-healing strategy for web-based applications, in: Proceedings of the 15th International Software Product Line Conference on - SPLC '11, ACM Press, New York, New York, USA, 2011, pp. 31:1–31:8.

[31] L. Baresi, S. Guinea, L. Pasquale, Service-Oriented Dynamic Software Product Lines, Computer 45 (10) (2012) 42–48.

[32] N. Bencomo, G. Blair, C. Flores, P. Sawyer, Reflective Component-based Technologies to Support Dynamic Variability, in: Second International Workshop on Variability Modelling of Software-Intensive Systems, 2008, pp. 141–150.

[33] N. Bencomo, P. Grace, C. Flores, D. Hughes, G. Blair, Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems, in: Proceedings of the 13th international conference on Software engineering - ICSE '08, ACM Press, New York, New York, USA, 2008, p. 811.

[34] J. C. Casquina, J. D. A. S. Eleuterio, C. M. F. Rubira, Adaptive Deployment Infrastructure for Android Applications, in: 12th European Dependable Computing Conference Adaptive, 2016, pp. 218–228.

[35] C. Cetina, J. Fons, V. Pelechano, Applying Software Product Lines to Build Autonomic Pervasive Systems, in: 2008 12th International Software Product Line Conference, Universidad Politécnica de Valencia, IEEE, 2008, pp. 117–126.

[36] C. Cetina, P. Giner, J. Fons, V. Pelechano, Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes, Computer 42 (10) (2009) 37–43.

[37] J. Cubo, N. Gamez, L. Fuentes, E. Pimentel, Composition and Self-Adaptation of Service-Based Systems with Feature Models, in: Safe and Secure Software Reuse, 13th International Conference on Software Reuse, ICSR 2013, Pisa, June 18-20. Proceedings, Vol. 7925 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 326–342.

[38] J. Cubo, N. Gamez, E. Pimentel, L. Fuentes, Reconfiguration of Service Failures in DAMASCo Using Dynamic Software Product Lines, in: 2015 IEEE International Conference on Services Computing, IEEE, 2015, pp. 114–121.

[39] L. Fuentes, N. Gamez, A feature model of an aspect-oriented middleware family for pervasive systems, in: Proceedings of the 2008 workshop on Next generation aspect oriented middleware - NAOMI '08, ACM, 2008, pp. 11–16.

[40] L. Fuentes, N. Gámez, Modeling the Context-Awareness Service in an Aspect-Oriented Middleware for AmI, in: 3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008, Vol. 51, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 159–167.

[41] H. Gomaa, K. Hashimoto, Dynamic software adaptation for service-oriented product lines, in: Proceedings of the 15th International Software Product Line Conference on - SPLC '11, 2, ACM Press, New York, New York, USA, 2011, p. 1.

[42] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven, Using architecture models for runtime adaptability, IEEE Software 23 (2) (2006) 62–70.

[43] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, E. Stav, A comprehensive solution for application-level adaptation, Software: Practice and Experience 39 (4) (2009) 385–422.

[44] J. Lee, G. Kotonya, D. Robinson, Engineering Service-Based Dynamic Software Product Lines, Computer 45 (10) (2012) 49–55.

[45] Jaejoon Lee, G. Kotonya, Combining Service-Orientation with Product Line Engineering, IEEE Software 27 (3) (2010) 35–41.

[46] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, A. Solberg, Models@ Run.time to Support Dynamic Adaptation, Computer 42 (10) (2009) 44–51.

[47] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, G. Blair, An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability, in: Model Driven Engineering Languages and Systems, Vol. 5301 LNCS of Lecture Notes in Computer Science (LNCS), Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 782–796.

[48] A. S. Nascimento, C. M. F. Rubira, R. Burrows, F. Castor, A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL, in: 2013 VII Brazilian Symposium on Software Components, Architectures and Reuse, IEEE, 2013, pp. 119–128.

[49] C. Parra, X. Blanc, L. Duchien, Context Awareness for Dynamic Service-Oriented Product Lines, in: SPLC '09 Proceedings of the 13th International Software Product Line Conference, ACM, San Franscisco, CA, 2009, pp. 131–140.

[50] C. Parra, X. Blanc, A. Cleve, L. Duchien, Unifying design and runtime software adaptation using aspect models, Science of Computer Programming 76 (12) (2011) 1247–1260.

[51] L. M. Pessoa, Flexibilidade em Linhas de Produtos Dinâmicas Cientes de Qualidade: uma Abordagem Baseada em Linguagens Específicas de Domínio, Master's thesis, Universidade de Brasília, [in Portuguese] (2014).

[52] J. D. A. S. Eleuterio, F. N. Gaia, A. Bondavalli, P. Lollini, G. N. Rodrigues, C. M. F. Rubira, On the Dependability for Dynamic Software Product Lines A Comparative Systematic Mapping Study, in: 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 323–330.

[53] ISO/IEC/IEEE, Systems and software engineering – Architecture description, Tech. Rep. ISO/IEC/IEEE 42010:2011, IEEE (2011).

[54] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture - A System of Patterns, Vol. 1 of Wiley Series in Software Design Patterns, John Wiley & Sons, 1996.

[55] M. C. Huebscher, J. a. McCann, A survey of autonomic computing—degrees, models, and applications, ACM Computing Surveys 40 (3) (2008) 1–28.

[56] ISO/IEC, System and Software Engineering - System and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models., Tech. Rep. ISO/IEC 25000:2014, ISO/IEC (2014).

[57] ISO/IEC/IEEE, Systems and software engineering – Vocabulary, Tech. Rep. ISO/IEC/IEEE 24765:2010, IEEE (2010).

[58] IBM. IBM Terminology [online, cited 28/Mar/2017].

[59] H. A. Müller, H. M. Kienle, U. Stege, Autonomic Computing Now You See It, Now You Don't, in: Software Engineering, Vol. 5413 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 32–54.

[60] P. K. Mckinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, A Taxonomy of Compositional Adaptation, Tech. rep., Michigan State University, East Lansing, Michigan (2004).

[61] J. van Zyl, Product Line Architecture and the Separation of Concerns, in: G. J. Chastek (Ed.), Software Product Lines: Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 90–109.

[62] W. L. Hursch, C. V. Lopes, Separation of Concerns, Tech. Rep. NU-CCS-5-03, Northeastern University, Boston (1995).

[63] I. Gorton, Software Quality Attributes, in: Intergovernmental Panel on Climate Change (Ed.), Essential Software Architecture, Vol. 1, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 23–38.

[64] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison Wesley, 2003.

[65] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering – Foundations, Principles, and Techniques, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[66] J. Bosch, Architecture-Centric Software Engineering, in: C. Gacek (Ed.), Software Reuse: Methods, Techniques, and Tools: 7th International Conference, ICSR-7 Austin, TX, USA, April 15–19, 2002 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 347–348.

[67] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic software product lines, Computer 41 (4) (2008) 93–95.

[68] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Reflecting on self-adaptive software systems, in: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 38–47.

[69] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, N. M. Villegas, Using Models at Runtime to Address Assurance for Self-Adaptive Systems, in: Models@run.time (Foundations, Applications, and Roadmaps), Vol. 8378 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 101–136.

[70] R. D. Lemos, D. Garlan, C. Ghezzi, H. Giese, Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511), Dagstuhl Reports 3 (12) (2014) 67–96.

[71] A. Metzger, K. Pohl, Software product line engineering and variability management: achievements and challenges, in: Proceedings of the on Future of Software Engineering - FOSE 2014, ACM Press, New York, New York, USA, 2014, pp. 70–84.

[72] S. A. Razak, M. A. Isa, D. N. A. Jawawi, O. L. Fuh, Model-Based Testing for Software Product Line: A Systematic Literature Review, International Journal of Software Engineering and Technology 2 (2) (2017) 27–34.

[73] I. S. Santos, L. S. Rocha, P. A. S. Neto, R. M. C. Andrade, Model Verification of Dynamic Software Product Lines, in: Proceedings of the 30th Brazilian Symposium on Software Engineering - SBES '16, ACM Press, New York, New York, USA, 2016, pp. 113–122.

[74] M. Cordy, A. Classen, P. Heymans, A. Legay, P.-Y. Schobbens, Model Checking Adaptive Software with Featured Transition Systems, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7740 LNCS, Springer, Berlin, Heidelberg, 2013, pp. 1–29.

[75] R. Muschevici, D. Clarke, J. Proenca, Feature Petri nets, in: Proceedings of the 14th International Software Product Line Conference (SPLC), 2010, pp. 99–106.

[76] DEVASSES, DEsign, Verification and VAlidation of large scale, dynamic Service SystEmS [cited 23/Mai/2017].
URL `http://www.devasses.eu/`

[77] V. Nunes, D. Mendonça, G. Rodrigues, V. Alves, Towards Compositional Approach for Parametric Model Checking in Software Product Lines, in: Workshop on Dependability in Adaptive and Self-Managing Systems – WDAS, 2013, pp. 19–22.

[78] G. N. Rodrigues, V. Alves, V. Nunes, A. Lanna, M. Cordy, P.-Y. Schobbens, A. M. Sharifloo, A. Legay, Modeling and Verification for Probabilistic Properties in Software Product Lines, in: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering, IEEE, 2015, pp. 173–180.

[79] J. Bosch, R. Capilla, R. Hilliard, Trends in Systems and Software Variability, IEEE Software 32 (3) (2015) 44–51.

[80] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, M. Hinchey, An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry, Journal of Systems and Software 91 (2014) 3–23.

[81] B. Schmerl, J. Andersson, T. Vogel, M. B. Cohen, C. M. Rubira, Y. Brun, A. Gorla, F. Zambonelli, L. Baresi, Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems, in: R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (Eds.), Software Engineering for Self-Adaptive Systems (SEfSAS), no. 9640 in Lecture Notes in Computer Science, Springer, 2017, pp. 1–29.