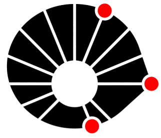

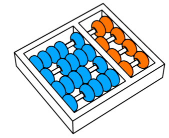

A Comparative Study of Dynamic Software Product Line Solutions for Building Self-Adaptive Systems

J. D. A. S. Eleutério *C. M. F. Rubira*

Technical Report - IC-17-05 - Relatório Técnico

April - 2017 - Abril

UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

A Comparative Study of Dynamic Software Product Line Solutions for Building Self-Adaptive Systems

Jane Dirce Alves Sandim Eleutério *   Cecília Mary Fisher Rubira†

Abstract

Modern systems need to able to self-adapt to changing user needs and system environments. Examples of systems that demand self-adaptive capabilities include mobile devices applications that should deal with environmental changes and service-oriented systems that should replace unreliable services on-the-fly. In this context, 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. However, researchers have reported that many DSPL solutions fail to meet all the system's adaptability requirements, and in many cases, they are developed in ad hoc manner. This paper surveys various DSPL solutions, evaluates and compares their different design strategies. A two-dimension taxonomy is specified to address basic technical issues for a given DSPL proposal. The DSPL dimension classifies the different design choices for implementing variability schemes, and for creating different kinds of feature models. The Self-adaptation dimension classifies the different design choices for the adaptation requirements and for the MAPE-K control loop implementation. Practical issues and difficulties are summarized, major trends in actual DSPL proposals are identified.

1 Introduction

Modern software systems need to adapt to heterogeneous environments and devices. Systems able to adapt their behavior or structure at runtime are called Self-Adaptive Software Systems (SASS) [25], and they can be self-adapted to adjust to the environment (hardware and other systems) or to correct failures that may occur at runtime, with no human interference. Autonomic control loops provide a generic mechanism of self-adaptation [18], and it is often modeled as the MAPE-K loop [104], which defines how systems adapt their behavior to keep their goals controlled, based on any regulatory control, disturbance rejection or optimization requirements [78]. The MAPE-K loop is divided into four activities: Monitor, Analyze, Plan, and Execute, including a Knowledge base that supports the required information throughout the loop [104]. In the context of SASS, several reference models are well established and widely used, such as FORMS [103], and they were used as the foundation to propose new solutions, such as Rainbow framework [48], and FUSION framework [38], and also served as a basis for other reference models and reference architectures, such as DYNAMICO and ACRA. DYNAMICO (Dynamic Adaptation, Monitoring, and Control model Objectives) [102] provides guidelines for design and implementation of SASS. ACRA (Autonomic Computing Reference Architecture) [78] is organized in three hierarchical layers (orchestration managers, resource managers, and managed resources).

Dynamic Software Product Line (DSPL) is an engineering approach to develop SASS based on the Software Product Line (SPL) paradigm [96], which deals with the modeling of commonalities

*Faculdade de Computação, UFMS, Campo Grande, MS. jane@facom.ufms.br
†Inst. de Computação, UNICAMP, Campinas, SP. rubira@ic.unicamp.br.

and variabilities among a family of similar systems [90, 10]. Commonality corresponds to similar parts among family products, while variability is defined as the ability of a product to be extended, modified, customized or configured for a specific context [98]. In general, feature models [64] are used to represent variability and commonality by means of features that can be classified as mandatory, optional or alternative [62]. 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. Variability is represented by variation points, which are differentiation points between products [68], and the variability decision, that is, the decision-making for the variation points, is known as binding time. In the SPL approach, the binding time occurs at design time to generate a product. For DSPLs, the binding time can occur (i) at design time to generate a product or (ii) at runtime to adapt the product allowing dynamic variability. Hence, DSPL allows Software Product Line to be reconfigurable at runtime [54]. Dynamic variability, also called late variability or runtime variability, can be represented using dynamic compositions. A dynamic composition is a set of features with runtime binding [92]. Whereas, the static variability can be represented using static compositions, which is a set of features with design-time binding [101].

However, engineers of DSPL systems could apply SASS reference and architectural models in order to reuse good practices in a systematic way. In these systems, the code devoted to autonomic and adaptation activities can be both numerous and complex. A number of DSPL solutions have been proposed in the literature. According to Bencomo *et al.* [15], many DSPL solutions are not dynamic as they should be because they partially (or do not) implement autonomic and adaptation activities in a complete way. Bencomo *et al.* [15] also conclude that the research on DSPL variability is still heavily based on the specification of decisions during design time, and that DSPLs are not as dynamic as researchers want to believe. Moreover, different strategies for implementing static and dynamic variabilities are proposed and implemented in ad hoc manner. Ideally, DSPL systems should be easy to understand, maintain and reuse. With such systems growing in size and complexity, employing adaptation and variability techniques while satisfying the requirements of software quality, such as maintainability, reusability and testability, are deep concerns to engineers of DSPL systems.

The purpose of this report is to investigate the applicability of the existing DSPL solutions for developing self-adaptive systems with effective quality attributes. The major contributions of this article are: (i) the definition of a taxonomy which is used to discuss technical design issues of a DSPL solution and to distinguish one solution from another, especially support for dynamic variability and adaptivity requirements are examined in detail, (ii) the presentation of a comprehensive survey of existing DSPL proposals, and (iii) the comparison and evaluation of the investigated proposals as well as the identification of the limitations in applying them in practice to develop self-adaptive systems. An extended version of this work is under development and will shortly be submitted to a well-known journal.

The remainder of this article is organized as follows. Section 2 gives a brief description of DSPL and self-adaptation terminology. This section also introduces some reference models for self-adaptive software systems. Section 3 describes our proposed taxonomy for classifying different design approaches to DSPL solutions. Section 4 discusses in more detail the investigated DSPL solutions. Finally, Section 5 presents some concluding remarks.

2 Self-Adaptation and DSPL Terminology

In this section, we present an overview of some important concepts of our work. First, Section 2.1 presents concepts of Self-Adaptive Software Systems (SASS, including reference model and reference architecture for SASS. Some backgrounds of Software Product Line (SPL) and Dynamic Software Product Line (DSPL) are presented in Section 2.2.

2.1 Self-Adaptation Terminology

Several techniques allow software to adapt to the execution environment, enabling changes in the software structure to fault recovery, improving performance, and increasing availability and safety in response to attacks [74]. Systems capable to adapt their behavior or runtime structure are called Self-Adaptive Systems (SASS) [25, 94, 83]. The system analysis is performed continuously and in parallel with the operation and maintenance of the system, to perform continued adaptation and evolution.

2.1.1 Self-Adaptation

According to Müller *et al.* [78], the self-adaptation can be presented in two types: *static*, where adaptation mechanisms are explicitly defined by system designers to choose from during execution; or *dynamic*, where the adaptation plans and monitoring requirements should be produced and selected by the system at runtime. According to Salehie and Tahvildari [94], the adaptation can be divided into two categories: internal and external. In *internal adaptation*, application logic and adaptation logic are intertwined, being based on programming language resources such as conditional expressions, parametrization, and exceptions. In *external adaptation*, the adaptation mechanism is in a separate external engine of the application logic.

2.1.2 Reflection Architectural Pattern

One important characteristic usually implemented in self-adaptive systems is the *reflective computation*, or computation about themselves [72]. Reflective computation allows the software system to have reflective capabilities, supporting introspection to observe and to reason about the system state to the decision-making about architectural reconfigurations [13]. *Reflection* is understood as the ability of a program to analyze and to modify its internal structure at runtime, if necessary [72]. Reflection allows operations such as start, stop and change the architecture that was outlined at design time, reasoning about the possible variation points and its variants. This characteristic stands out since it allows software systems to be modified at runtime. The reflection architectural pattern provides a mechanism to adapt the structure and the behavior of software systems at runtime. This architectural pattern separates the system into two parts: (i) *meta level* provides information about selected system properties and makes the system self-aware, and (ii) *base level* includes the application logic [20]. Also, a self-adaptive system has to support the dynamic loading of components to enable adaptation at runtime. Dynamic loading allows the software component to be started, stopped or updated independently without affecting other components or without restarting the entire system.

2.1.3 The Managed and Managing Subsystems

Following the external adaptation approach, Weyns *et al.* [104] use the general terms *managed subsystem* and *managing subsystem* for designating the constituent parts of a self-adaptive software

system. The managed subsystem is the application logic that provides the system domain functionality, also called *system layer* [48], *core function* [94], *base level* [20], and *base-level subsystem* [103]. The managing subsystem manages the managed subsystem and is organized according to the autonomic control loop (or feedback loop), and is also called *architecture layer* [48], *adaptation engine* [94], *meta level* [20], and *reflective subsystem* [103].

2.1.4 MAPE-K Control Loop Reference Model

Kephart and Chess [66] proposed the idea of an autonomic element as a building block for self-managing systems in the form of a *Monitoring-Analyze-Planning-Execute* loop and a shared knowledge (MAPE-K) [66, 58]. The MAPE-K is a reference model for creating autonomic control loop which encompasses four activities or modules: (i) *Monitoring*: in the first activity, monitors gather and process environmental context information that is relevant to the adaptation process by using sensors, and the information collected is sent to the second activity; (ii) *Analyze*: in this activity, analyzers use the collected data in the previous activity and correlate context information to infer data from the runtime environment and the system behavior; (iii)*Planning*: in the third activity, planners use the analyzed data to define adaptation plans; and (iv) *Execute*: in the last activity, executors implement and execute plans to adapt the running system and get the desired behavior through the use of effectors or actuators. The monitoring activity continuously feeds the adaptive loop, resetting it. The knowledge base supports the required information flow throughout the loop. It is common to find in the literature a variation of MAPE-K called MAPE, where the knowledge base (K) still exists, but it is abstracted for simplification purposes, as in [104, 36].

According to Weyns *et al.* [104], the managing subsystem may consist of one or more autonomic control loops, and MAPE functions may be mapped to different components, or may be integrated into one or more components. Thus, they present a *centralized* pattern and five different decentralized patterns, considering the interactions between the different stages of autonomic loops performed by the MAPE components [104]: *coordinated control*, *information sharing*, *master/slave*, *regional planning*, and *hierarchical control*.

2.1.5 FORMS Reference Model

FORMS: Weyns *et al.* [103] proposed FORMS (a FOrmal Reference Model for Self-adaptation) to allow the description and reasoning with precision about the architectural characteristics of self-adaptive systems. Existing frameworks and principles of self-adaptation served as the basis for establishing this reference model, including computational reflection [72], MAPE-K [66], and architecture-based adaptation [67, 84]. The FORMS reference model is extensible and consists of a set of modeling primitives and a set of relations between them that outlines the composition rules, as shown Figure 1. These modeling primitives correspond to the key variation points within self-adaptive software systems.

FORMS divides a self-adaptive system into one or more base-level and reflective subsystems. A base-level subsystem consists of domain models and base-level computations, including domain logic. A reflective subsystem manages another subsystem, which can be either a base-level or other reflective subsystem. According to FORMS, a reflective subsystem can manage another reflective subsystem, in which case there are several reflective levels.

2.2 Software Product Line Concepts

The Software Product Line (SPL) approach is becoming more important for dealing with identification of variability and commonalities among software systems [65]. An SPL is a set of software

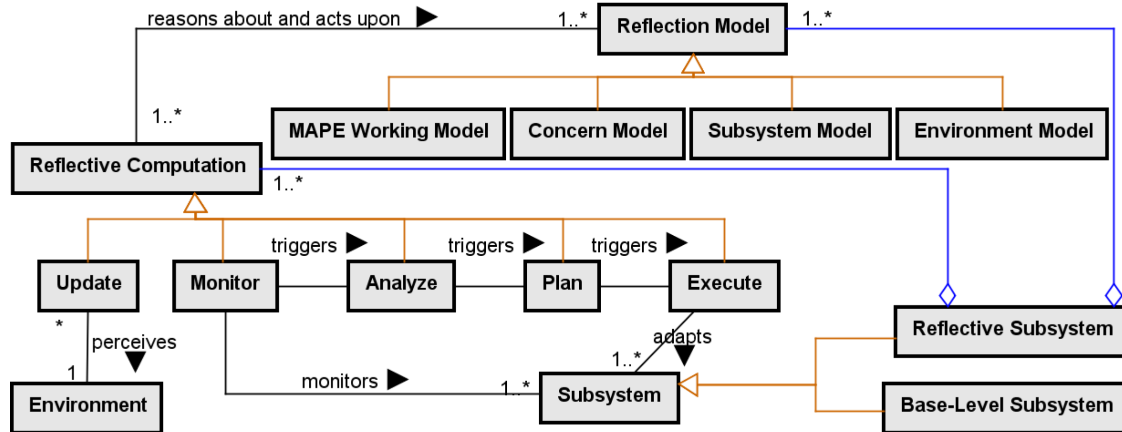

Figure 1: View of FORM's primitives and their relationships [103]

systems that shares a common managed set of features that meets the specific needs of a particular market segment or mission, and it is developed from a common set of core assets [26]. Commonality denotes features that are part of each product in the same form, while variability is defined as the ability of a software system or an artifact to be extended, modified, customized or configured for a specific context [65]. The moment when the variability resolution is performed is called the binding time [68]. Variability is represented by a variation point, which is a differentiation point between products [68] The variability modeling is commonly performed by means of feature model [65], representing variabilities and commonalities in the form of features. Features can be classified as mandatory, optional or alternative [62]. Mandatory features are present in all applications derived from SPL. On the other hand, the optional features may or may not appear in derivative applications. Finally, alternative features can be selected according to the mutual exclusion constraints. Each application derived from an SPL is also a configuration.

2.2.1 Product Line Architecture

Another important model of an SPL is the Product Line Architecture (PLA), which explicitly represents the commonalities and variabilities of architectural elements and their configurations [101]. A PLA model contains the architectural elements and is usually represented as a UML-based component diagram. An architectural element can be a component or another element of the architecture, as an interface or a connector, due to the programming language or component-based implementation model. In the software engineering process, the feature model is generated during the analysis phase, whereas the PLA model is generated during the design phase [101].

2.2.2 Mapping Feature Model to PLA Model

The relation between both models, which is the mapping between features and architectural elements, is usually carried out by a software architect who has a great understanding of the product line domain. However, there are some methods to assist this activity, as the FArM (Feature-Architecture Mapping) method [97]. FArM allows performing a sequence of transformations in the feature model to map features to a PLA model. The FArM method enables the construction of a transformed feature model containing exclusively functional features, whose business logic can be implemented into architectural elements [97].

Many iterations progressively transform the initial feature model. FArM has four transformation activities: (i) Removing non-architecture-related features and resolution of quality features; (ii) Transformation based on the architectural requirements; (iii) Transformation based on the interacts relations between features; and (iv) Transformation based on the hierarchy relations between features. The FArM transformation of features into a PLA model is not automated. However, it is a series of well-defined transformations on the feature model, which achieve a strong mapping between features and architectural elements [97]. Consequently, the FArM method ensures the feature model and PLA model are consistent with each other. It is widely accepted that FArM improves maintainability and flexibility of PLAs [97].

2.2.3 Dynamic Software Product Line

The Dynamic Software Product Line (DSPL) extends the concept of conventional SPL, allowing the generation of software variant at runtime. In particular, SPL emphasizes the variability analysis, decision-making and product configuration at the design phase. Whereas, DSPL emphasizes variability analysis at design time, postponing the decision of the variability and the application reconfiguration to be made at runtime. In other words, a DSPL is an SPL that requires the SASS area knowledge to manage the variability at runtime (adaptation). DSPL binds variation points at runtime, initially when the software is released to adapt to the current environment, as well as while operating to adapt to environmental changes [54]. DSPL dynamicity allows the SPL to be reconfigurable at runtime [54].

Using SPLs, the binding time can occur at design time to generate a product using static binding. In the case of DSPLs, the binding time should occur at runtime to allow product adaptation 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 [92], whereas, static variability can be represented using static compositions, which is a set of features with static binding [101].

2.2.4 Design and Runtime Binding

According to Alves *et al.* [5], the binding time in SPL is traditionally at pre-compile, compile, and link time, dealing with static variability. Whereas in systems that deal with dynamic variability can be at load time when the system is first deployed and loaded into memory, and more commonly at runtime after the system has begun executing. Consequently, since a DSPL can deal with static and dynamic variability, then it can perform all binding times. In order to simplify, we define as design time binding all binding times that occur during the design phase, which are: pre-compile, compile, and link time. So, we will consider only three binding times, which are: design time, load time and runtime. *Design time binding* is required by static composition, whereas *load time binding* and *runtime time binding* is required by dynamic composition.

2.2.5 Feature modeling

Feature model was proposed as part of the FODA (Feature-Oriented Domain Analysis) method [62] and is widely used nowadays. Since then, several extensions and variations of the FODA notation have been proposed [12, 21, 31, 32, 33, 40, 42, 52, 56, 57, 70, 63, 88, 89, 99, 101], as shown Figure 2. These feature modeling approaches is commonly called FODA-like notations.

The Orthogonal Variability Model (OVM) [68] is an FODA-like notation that relates the defined variability to other software development models, such as use case models and architectural models. Thus, OVM approach has an orthogonality integrated, providing a traceability between features

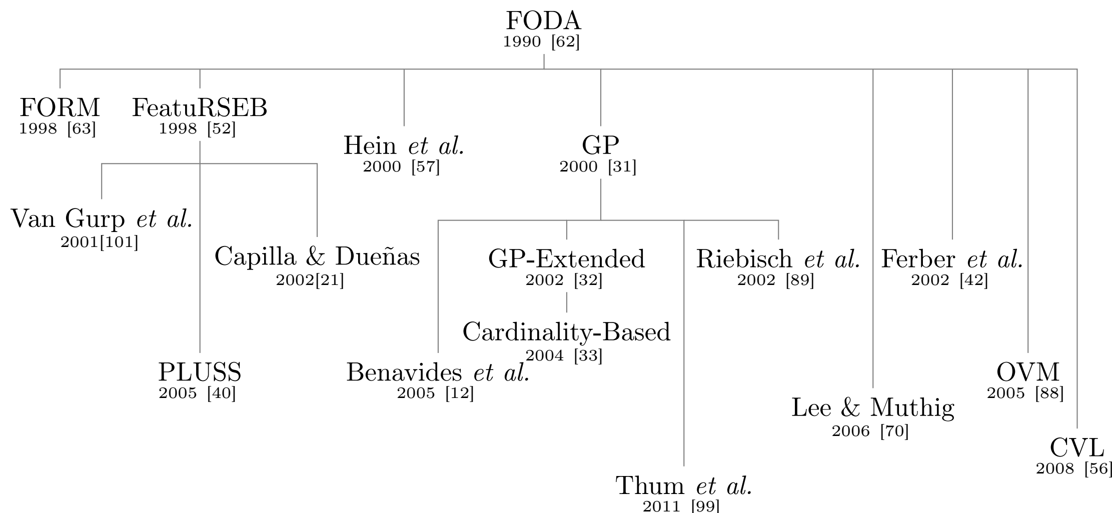

Figure 2: Feature model approaches (extended from [64, 17]).

and other elements that realize/implements the features. An OVM model provides a cross-sectional view of the variability among other software design artifacts, requiring no additional artifacts for traceability.

The Common Variability Language (CVL) [56] is a domain-independent language for specifying and resolving variability. CVL is based on a BVR approach (Base-Variation-Resolution), encompassing three models: base model describes the architecture is a language-independent, variability model defines variability on the base model, and resolution model defines how to resolve the variability model to create a new model in the base model. The CVL variability model is a feature model that can be considered an FODA-like feature model because it represents a tree of features and inherits other FODA concepts.

Most of these FODA-like notations approaches (Figure 2) focus on representing the *variability in space*, emphasizing improvements of notation, new types of features, cardinalities and feature attributes, and extended relationships to define more accurately the constraints and relationships between features [17]. Variability in space is the existence of an artifact in different shapes at the same time [68]. However, As presented in 2.2.4, SPLs support only design time binding, while DSPLs also support load and runtime binding. Nevertheless, there are few attempts to also represent the *variability in time* as [42, 70]. Variability in time is a property of variability models and products, that defines when features should or can be bound to their values to realize variability [17]. Variability in time refers to the binding time.

Ferber *et al.* [42] proposed a feature modeling approach, considering two different views on the feature model to represent the dynamic information. The first view is an FODA-like feature model notation, named *Feature Tree View*, which represents all variabilities without distinguishing between static or dynamic. The second is named *Feature Interaction and Dependency View* and represents feature interactions and dependencies to allow dynamic features. This second view uses a specific notation and is represented separated from the first view.

Lee and Muthig [70] create a new kind of model called *feature binding graph* to complement the feature model (FODA-like). In a feature binding graph, each node represents a feature binding unit (FBU), which is a set of features related to each other by the relationships in the feature model. The relationship between FBUs can be either static or dynamic, being differentiated by preconditions annotated in a dynamic binding relation to indicate when FBUs can be bounded at runtime. This

proposed model, *feature binding graph*, is used as a complementary model to feature model.

3 A Taxonomy for DSPL Solutions

There is a number of design issues for building DSPL proposals that will be used for constructing self-adaptive systems. However, the chosen strategy for designing each design issue varies from proposal to proposal. This section presents a taxonomy which identifies the several common design issues for building DSPLs, and classifies the different design solutions. The taxonomy was developed based on the set of analyzed DSPL solutions (Section 4), and on some reviewed previous studies in the area of self-adaptation and DSPL.

More specifically, in the area of self-adaptation, the work by Andersson *et al.* [6] have presented an adaptation taxonomy for self-adaptive software systems. Bashari *et al.* [9] present a detailed adaptation taxonomy for DSPL, establishing an individual classification for each MAPE-K activity, that is, monitor, analyse, plan, and execute. In the area of DSPL, Galster *et al.* [47] have provided a variability characterization for software systems. The work by Bencomo *et al.* [15] and the work by Raniel *et al.* [34] have presented systematic literature reviews to analyse DSPL solutions based on the MAPE-K autonomic control loop. Eleuterio *et al.* [37] have presented a systematic mapping study to identify DSPL solutions based on the MAPE-K autonomic control loop.

We have compiled and refined the different design issues proposed by these previous studies, combining similar aspects and removing design decisions not applicable to the DSPL context. We have also included new design issues related to DSPLs. Our proposed taxonomy classifies the design issues of a DSPL scheme into two dimensions: the first dimension is relation to self-adaptation design issues, while the second dimension is related to variability design issues. The self-adaptation dimension is classified into six aspects of interest: *(i)* adaptation architectural pattern, *(ii)* adaptation cause, *(iii)* adaptation realization technique, *(iv)* adaptation binding time, *(v)* adaptation automation, and *(vi)* use of MAPE-K pattern. The variability dimension is classified into nine aspects of interest: *(i)* variability type, *(ii)* variability conceptual model, *(iii)* feature modeling strategy, *(iv)* variability architectural model, *(v)* variability architectural style, *(vi)* variability managed element, *(vii)* variability orthogonality, *(viii)* variability traceability, and *(ix)* variability platform. In the following we discuss each aspect in turn.

3.1 The Self-Adaptation Dimension

Adaptation Cause. It refers to the source that trigger/initiates the adaptation process. Such sources can be represented as (i) *context change*, (ii) *system change*, and (iii) *user change. Context changes* occur in the environment and are external to a 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 [55, 44]. The system 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. The user context information includes data such as position (based on GPS information in a smartphone, for example), and environmental information in which the user is entered, as light and noise. *System changes* occur internally to the system, for instance, failure of a component, the performance of a service, and exceptions. The ability to react to system changes requires specific sensors within the system implemented according to change to be detected. *User changes* are triggered by some user action while using the system and they refer to changes in user requirements or user needs at runtime. User interface usually is the way to obtain information related to user changes.

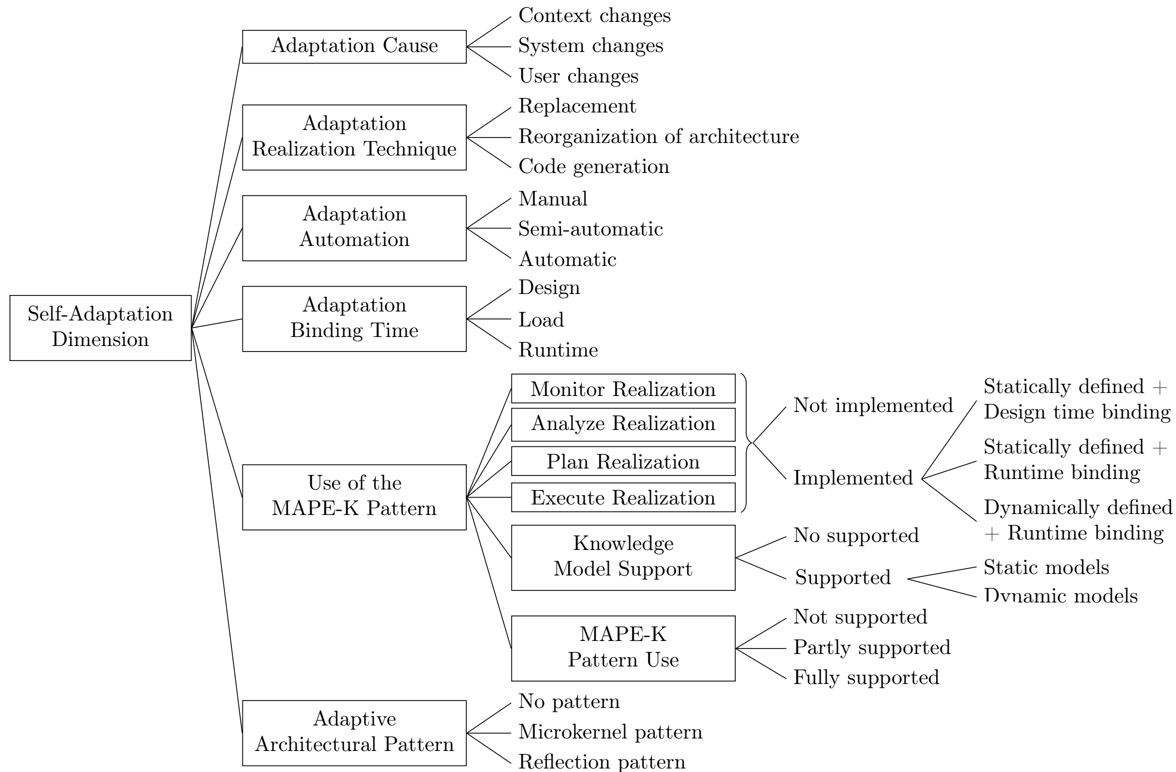

Figure 3: Self-adaptation dimension.

Adaptation Realization Technique. It is related to the implementation approach to realize the adaptation. There are at least three different design solutions for implementing the adaptation: (i) *replacement*, (ii) *reorganization of the architecture*, and (iii) *code generation*. In the first approach, adaptation is achieved by replacing one architectural element with another with a similar interface, without affecting the rest of the system architecture. In the second approach, a reconfiguration of architecture is performed when adaptation occurs, reorganizing the structure of architecture. 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 behavior of the system.

Adaptation Binding Time. It defines when the adaptation binding occurs (Section 2.2.4). There are three different design solutions for the binding occurrance: (i) *at design time*, (ii) *at load time*, and (iii) *at runtime*. In the first approach, design time refers to adaptation performed during development phases, including pre-compile, compile, and link time. In the second approach, load time is defined as when the system is first deployed and loaded into memory. In the third approach, runtime refers to the adaptation performed while the system is running. 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.

Adaptation Automation. Automation refers to the degree of outside intervention during adaptation. There are at least three design solutions for automation: (i) *manual adaptation*, (ii) *semi-automatic adaptation*, and (iii) *automatic adaptation*. In the first case, the adaptation is performed by a human effort. In the second case, the adaptation is performed manually but supported by tools in a semi-automatic way. In the third case, the adaptation process is fully automated with no

human intervention.

Use of the MAPE-K Pattern. Managing subsystems should implement the MAPE-K control loop in order to be considered dynamic solutions, according to Bencomo *et al.* [15]. This requirement should consider at least three design aspects: (i) whether or not the four activities of the MAPE-K are present in the implementation of the managing system as individual functionalities (*realization of individual MAPE-K Activities*), (ii) to what extent the knowledge base supports the representation of models (*model support by the Knowledge Base*, and (iii) whether or not the proposal adheres to *the use of the MAPE-K Pattern*, as shown in Figure 3.

The first aspect identifies whether or not the implementation of the monitor, analyze, plan, and execute functionalities are supported. For instance, some proposals implement a simplified version of the MAPE-K pattern including only the planner and the executor components, while others implement all four activities. 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.

The second design aspect refers to what extent the knowledge base supports model representations and model extensibility at runtime. The knowledge base can support (i) *no models*, (ii) *static models* or (iii) *dynamic models.* On one hand, static models cannot be extended at runtime, that is, a 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 analyzer and/or the planner components. On the other hand, 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 time by the analyzer 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 solution 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 solution explicitly applies the control loop pattern.

Adaptive Architectural Pattern. According to Buschmann *et al.* [20] architectural pattern expresses a fundamental structural organization schema for software systems, providing a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. We consider the architectural patterns for adaptive systems presented by Buschmann *et al.* [20]. Hence, the use of adaptive architectural pattern can be: (i) *no pattern* when the solution does not follow an adaptive architectural pattern ; (ii) *microkernel* when the solutions follows the Microkernel Architectural Pattern; or (iii) *reflection* when the solutions follows the Reflection Architectural Pattern.

3.2 The DSPL Variability Dimension

Variability Type. We classify the design approaches for supporting variability into three types: (i) *only static variability*, (ii) *only dynamic variability*, and (iii) *both static and dynamic variabilities.* Static variability is performed by the compiler while dynamic variability is performed by the runtime

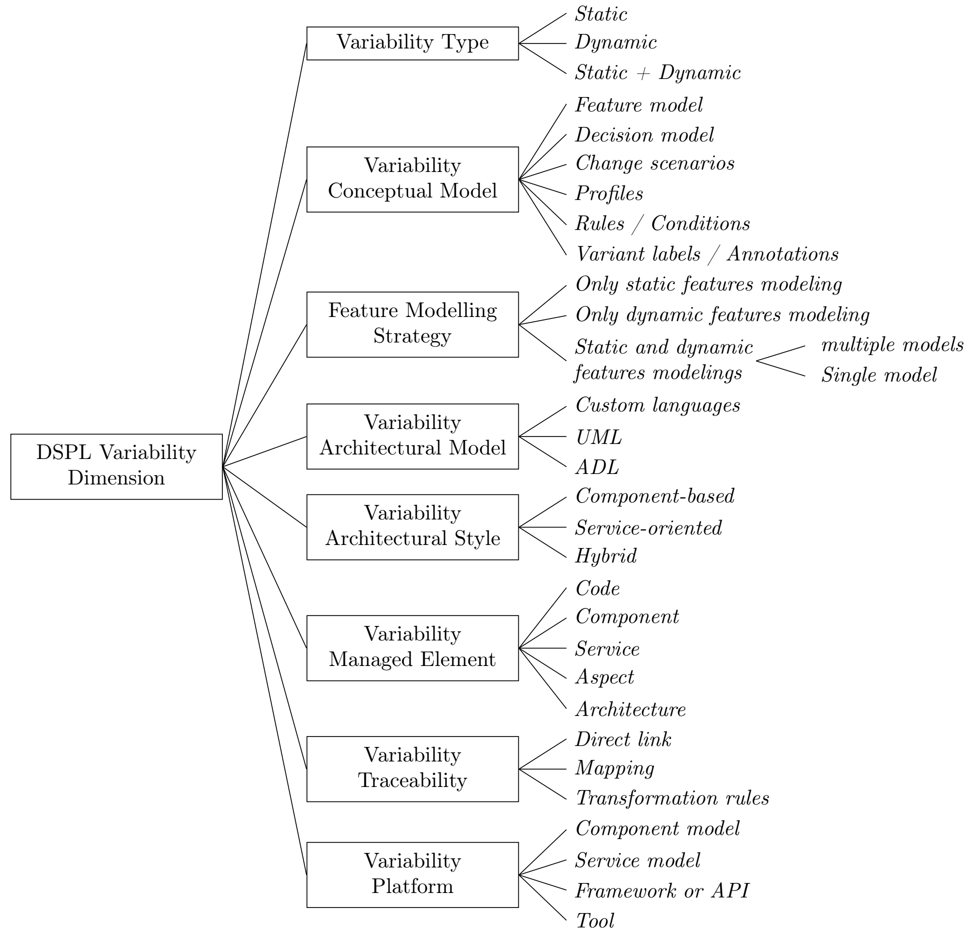

Figure 4: DSPL variability dimension.

system. A DSPL solution should support at least dynamic variability. Some DSPL solutions support only dynamic variability while other ones perform both static and dynamic variabilities.

Variability Conceptual Model. Variability can be represented as: (i) *feature model*, (ii) *decision models*, (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 solutions. 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 modeled to describe events or options that trigger changes in the managed subsystem. In the fourth approach, profiles are created to represent descriptive summaries of artifacts in the environment (as a table, model or a set of expressions). In the fifth approach, a set of rules or conditions is coded into the system in order to support variability. In the sixth approach, variant labels or annotations are added to artifacts that represent the DSPL.

Feature Modeling Strategy. There are at least three design strategies for feature representation: (i) *only static features modeling*, (ii) *only dynamic feature modeling* and (iii) *static and dynamic features modeling*. The first case represents the traditional use of feature models with static binding, and it is not applied to DSPLs. In the second case, the feature model represents only dynamic

features with dynamic binding, encompassing FODA-like feature models, orthogonal variability models (OVM) and common variability language (CVL) (Section 2.2.5). In the third case, the mechanism supports both static and dynamic features at the same time. As mentioned in Section 2.2.5, most of the traditional notations of feature models emphasize 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 bindings times, for instance [42, 70] (Section 2.2.5); and (ii) use of a *single feature model* based on an extended notation to represent both static and dynamic features, for instance [69].

Architectural Model. There are at least three design techniques for representing PLA architectures: (i) using *customized languages*, (ii) using *UML*, and (iii) using *ADL*. In the first case, some DSPL solutions define customized languages or ad hoc models to represent the architectural model. In the second case, UML diagrams or UML profiles are used to represent the architectural model. In the third case, Architecture Description Language (ADLs) are used to define the software architecture. An ADL is any language for use in an architecture description, and can be used by one or more viewpoints to represent identified system concerns within an architecture description [59]. As examples of ADL, we can mention: AADL[1], Rapide [71], Wright[2], and SysML[3].

Architectural Style. It refers to the highly granular entities of the system and how they are connected to each other [9]. There are at least three different architectural styles used by DSPL solutions: (i) *component-based style*, (ii) *service-oriented architecture style*, and (iii) *hybrid architectural style*. In the first approach, component-based architectures provide the systems' 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, a hybrid architectural style, using specifications such as Service Component-Architecture (SCA).

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 types of elements, such as: (i) *code*, (ii) *components*, (iii) *services*, (iv) *aspects*, and (v) *software architectures*. 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 is compared to meet variability, as in [16].

Variability Traceability. It refers to the mapping between the variability and architectural models. The traceability is required when the separated variability orthogonality is chosen. There are at least three strategies for supporting traceability: (i) *direct link*, (ii) *mapping*, or (iv) *transformation*

[1]SAE Architecture Analysis and Design Language. `http://www.aadl.info/aadl/currentsite/` Accessed on April 27, 2017.

[2]Wright ADL. `http://www.cs.cmu.edu/~able/wright/` Accessed on April 27, 2017.

[3]OMG Systems Modeling Language. `http://www.omgsysml.org/` Accessed on April 27, 2017.

rules. In the first approach, a direct link is defined between the variability elements and architectural elements, for instance, using OVM or CVL to relate features to architectural elements or using stereotypes in components of architectural model as a reference to features of feature model. This approach does not require another artifact to specify traceability because it is defined interwoven in the variability model or the architectural model. In the second approach, the traceability is performed in a mapping table or mapping model from the elements of variability model to elements of the architectural model. In the last approach, transformation rules define the traceability between variability and architectural model.

Variability Platform. We classify the technology used to implement the DSPL into four categories: (i) *component model*, (ii) *service model*, (iii) *framework or API*, and (iv) *supporting tools*. Component model defines the standard and conventions imposed on system components in order to describe their functions and how they interact, for instance, OSGi [4], OpenCom [27], and Fractal [19]. Service model defines the standards and protocols used to implement and bind services, for instance REST [43], SOAP/WSDL [53] and Jini[4]. Framework refers to the infrastructure used as a foundation to build the system and generally corresponds to an implementation of a component model or service protocol using a specific programming language, for instance Eclipse Equinox[5] and Java Reflection API[6]. Supporting tools support the construction of the DSPL, for instance ITACA[7] and Genie [14].

4 Selection of DSPL solutions

We have searched in the literature for relevant solutions in the intersection of Software Product Lines (SPLs), and Self-Adaptive Software Systems (SASSs). In particular, we have searched for two types of systems: (i) SPLs that support dynamic variability, and (ii) self-adaptive systems that apply SPL techniques to manage variability at runtime. Table 1 lists the selected DSPL solutions that support some form of dynamic variability. This list was defined based on two sources of information: (i) the systematic mapping study developed by the authors to identify proposals that include dependability attributes in DSPLs [37] and (ii) two surveys [15, 9] that identify DSPLs and analyze the use of the MAPE autonomic control loop.

Our systematic mapping study [37] reviewed papers about Dependable DSPLs selecting nine primary studies. Also, we made a comparison of the primary studies regarding the MAPE-K loop activities and the DSPL dimension. The survey developed by Bendomo *et al.* [15] questioned the dynamism level of DSPL solutions compared to MAPE-K loop. They selected nine DSPL solutions and analyzed whether each DSPL meet phases of autonomic control loop at runtime (dynamic) or design-time (static). The survey presented by Bashari *et al.* [9] proposed a conceptual framework for comparing adaptation realization in DSPL based on MAPE-K loop. They also compared seven DSPL approaches using the proposed framework.

Table 2 shows the value of each design aspect in our self-adaptive dimension and Table 3 shows the value of each design aspect in our DSPL variability dimension for each of the approaches that have been reviewed.

In the following, we use Tables 2 and 3 to compare each of the selected approaches:

[4]Jini Service Oriented Architecture. `http://river.apache.org/release-doc/current/spec-index.html`. Accessed on April 05, 2017.

[5]Eclipse Equinox. `http://www.eclipse.org/equinox/`. Accessed on April 05, 2017.

[6]Java Reflection API. `https://docs.oracle.com/javase/tutorial/reflect/`. Accessed on April 05, 2017.

[7]ITACA. `http://itaca.gisum.uma.es/`. Accessed on March 09, 2017.

Table 1: Selected DSPL solutions.

Approach	**Year**	**References**
Abbas *et al.* (ASPL)	2010	[2, 1]
Abotsi *et al.*	2013	[3]
Baresi *et al.*	2012	[8]
Bencomo *et al.* (Genie)	2008	[14, 13]
Casquina *et al.* (Cosmapek)	2016	[22]
Cetina *et al.* (MoRE)	2008	[23, 24]
Cubo *et al.* (Dynamic DAMASCo)	2013	[28, 29]
Fuentes and Gamez	2008	[45, 46]
Gomaa and Hashimoto	2011	[51]
Hallsteinsen *et al.* (MADAM)	2006	[55, 44, 50]
Lee *et al.*	2012	[69, 60]
Morin *et al.* (DiVA)	2008	[77, 76, 75]
Nascimento *et al.* (ArCMAPE)	2014	[80, 79]
Parra *et al.* (CAPucine)	2009	[86, 85]
Rosenmüller *et al.*	2008	[92, 90, 91]

Abbas *et al.* (ASPL): Autonomic Software Product Lines (ASPL) [2, 1] is an SPL that uses the concepts of autonomic application. The approach of Abbas *et al.* is based on reflection architectural pattern and MAPE-K pattern, performing all MAPE-K activities. ASPL uses information about context and system to trigger adaptations, realizing the replacement of components at load and runtime. Abbas *et al.* uses the OVM to feature modeling, representing only dynamic variability, and its features are mapped components of dRS model (Domain Responsibility Structure). The dRS model uses a custom language based on UML component diagrams with provide/requires interfaces. The dRS models decisions regarding architectural components, how they are structured, and the responsibilities assigned to the components. ASPL is proposed using a component-based architecture, and its implementation is based on OSGi and related technologies.

Abotsi *et al.*: They proposed a theoretical approach [3] that leverage the internal variability concept of SPL paradigm to support self-healing mechanisms. Their approach is based on MAPE-K, but they do not detail the implementation of all activities. The knowledge base captures and modeling the elements that are critical to fault detection and recovery decisions. They proposed a hybrid architecture style using components and services. The variability is represented using OVM only with dynamic variability and is performed by replacement of services. When a service invocation fails, a new service is selected.

Baresi *et al.*: This approach [8] proposes the convergence of SOA and DSPL by combining BPEL process [82] with CVL and using a dynamic version of BPEL. Their approach is based on DyBPEL, a tool that extends ActiveBPEL [39] so that the service composition (process) can adapt to changes during its operation. DyBPEL exploits aspect-oriented programming to change the features bound to variation points dynamically. In this approach, the adaptation planning is performed by a human through a user interface. DyBPEL uses as input a variability designer based on Eclipse CVL plugin. Their CVL variability model represents only dynamic features.

Bencomo *et al.* (Genie): Bencomo *et al.* propose the Genie [13, 14], a tool that uses architecture models to support the generation and execution of self-adaptive systems for grid mobile computing and embedded systems. Genie has a component-based reflective architecture. In this approach,

Table 2: Self-adaptive dimension of DSPL solutions.

Design Aspect \ DSPL Solution		**Abbas *et al.*** (ASPL)	**Abotsi *et al.***	**Baresi *et al.***	**Bencomo *et al.*** (Genie)	**Casquina *et al.*** (Cosmapek)	**Cetina *et al.*** (MoRE)	**Cubo *et al.*** (Dynamic DAMASCo)	**Fuentes & Gamez**	**Gomaa & Hashimoto**	**Hallsteinsen *et al.*** (MADAM)	**Lee *et al.***	**Morin *et al.*** (DiVA)	**Nascimento *et al.*** (ArCMAPE)	**Parra *et al.*** (CAPucine)	**Rosenmüller *et al.***
Adaptation Cause		Context System	System	User System	Context	Context System	Context	Context	Context	Context	Context System	Context System	Context System	Context System	System	Context
Adaptation Realization Technique		Repl.	Repl.	Repl.	Reorg.	Reorg.	Repl.	Reorg.	Reorg.	Repl.	Repl.	Reorg.	Reorg.	Reorg.	Reorg.	Code gen.
Adaptation Automation		Auto	Auto	Semi	Auto	Auto	Auto	Auto	Auto	Auto	Auto	Auto	Auto	Auto	Auto	Auto
Adaptation Binding Time		Load Runtime	Runtime	Runtime	Runtime	Design Load Runtime	Load Runtime	Design Load Runtime	Load Runtime	Runtime	Load Runtime	Design Runtime	Runtime	Runtime	Runtime	Design Load Runtime
Use of MAPE-K Pattern	Monitor realization	✓	≈	−	✓	≈	✓	≈	−	✓	✓	✓	✓	≈	✓	✓
	Analyze realization	✓	−	≈	≈	≈	−	✓	≈	≈	✓	✓	✓	≈	≈	≈
	Plan realization	✓	−	≈	≈	≈	≈	✓	≈	≈	✓	≈	✓	≈	≈	≈
	Execute realization	✓	≈	✓	≈	≈	≈	✓	≈	✓	✓	≈	✓	✓	✓	✓
	Knowledge model support	Dynamic models	Static models	Static models	Dynamic models	Static models	Static models	Dynamic models	Static models	Static models	Dynamic models	Dynamic models	Dynamic models	Static models	Static models	Static models
	MAPE-K use	Fully	Not	Not	Not	Fully	Partly	Not	Not	Partly	Partly	Not	Partly	Fully	Partly	Partly
Adaptive Architecutral Pattern		Reflection	*No pattern*	*No pattern*	Reflection	Reflection	Reflection	*No pattern*	Micro-kernel	*No pattern*	Reflection	*No pattern*	Reflection	Reflection	*No pattern*	Reflection

✓implemented and dynamically defined with runtime binding,
≈ implemented and statically defined with runtime binding,
∗ implemented and statically defined with design time binding,
− not implemented

Table 3: DSPL variability dimension of DSPL solutions.

Design Aspect \ DSPL Solution		**Abbas *et al.*** (ASPL)	**Abotsi *et al.***	**Baresi *et al.***	**Bencomo *et al.*** (Genie)	**Casquina *et al.*** (Cosmapek)	**Cetina *et al.*** (MoRE)	**Cubo *et al.*** (Dynamic DAMASCo)	**Fuentes & Gamez**	**Gomaa & Hashimoto**	**Hallsteinsen *et al.*** (MADAM)	**Lee *et al.***	**Morin *et al.*** (DiVA)	**Nascimento *et al.*** (ArCMAPE)	**Parra *et al.*** (CAPucine)	**Rosenmüller *et al.***
Variability Type		*D*	*D*	*D*	*D*	*SD*	*D*	*SD*	*D*	*D*	*D*	*SD*	*D*	*D*	*D*	*SD*
Variability Conceptual Model		Feature model	Feature model	Feature model	Feature model	Feature model	Feature model	Feature model	Feature model	Feature model	Variant label	Feature model	Feature model	Feature model	Feature model	Feature model
Feature modeling Strategy		*FMD* (OVM)	*FMD* (OVM)	*FMD* (CVL)	*FMD* (OVM)	Single model	*FMD*	Multiple models	*FMD*	*FMD*		Single model	*FMD*	*FMD* (CVL)	*FMD*	Multiple models
Variability Architectural Model		*CL* / dRS model	*not specified*	*CL* / BPEL, CVL	*ADL* / Open-COM DSL	*UML*	*CL* / PervML model	*CL* / CA-STS model	*ADL* / xADL	*not specified*	*UML*	*CL* / Feature model	*ADL* / meta model	*UML* / CVL base model	*not detailed*	*not specified*
Variability Architectural Style		*CB*	Hybrid	*SO*	*CB*	Hybrid	*SO*	Hybrid	*CB*	Hybrid	*CB*	*SO*	Hybrid	Hybrid	Hybrid	*not specified*
Variability Managed Element		Comp.	Comp. Service	Code Aspect	Comp.	Comp. Service	Service	Comp. Service	Comp. Aspect	Comp. Service	Comp.	Service	Comp. Service Aspect	Comp. Service	Comp. Service Aspect	Code
Variability Traceability		*direct*/ OVM to dRS	*not detailed*	*not detailed*	*direct*/ OVM to Transit. model	*direct*/ FArM, XML	*mapping model* / Realiz. model	*mapping table* /	*not detailed*	*mapping table*	*not detailed*	*direct*/ services in FM	*not detailed*	*direct*/ FArM, CVL	*not detailed*	*direct*/ FM to code
Variability Platform	Component Model	OSGi	-	-	Open-COM	DyCos-mos	OSGi	-	CAM/ DAOP	OSGi	J2ME	-	Fractal, Open-COM, OSGi	COS-MOS*, OSGi	Fractal	-
	Service Model	-	-	-	-	REST	-	SOAP/ WSDL	-	SOAP/ WSDL	-	Jini	-	SOAP/ WSDL	SOAP/ WSDL, JavaRMI	-
	Framework or API	-	-	-	-	Java Refl. API	-	-	CAM/ DAOP	Apache CXF, Eclipse Swordfish	-	Apache River	-	Eclipse Equinox	FraSCAti, COS-MOS	-
	Tool	-	-	DyBPEL	Genie	-	FAMA	ITACA	-	-	-	-	-	-	-	Feature-C++

S — Static variability, *D* — Dynamic variability, *SD* — Static and Dynamic variability, *FMD* — Feature model only with dynamic feature.
CB — Component-based, *SO* — Service-oriented.

context changes as the physical location can trigger an adaptation. This approach uses the OVM for feature modeling, dealing only with dynamic features. The OVM has a direct link to transition diagrams, which guides the reconfiguration and adaptation process. Genie uses the OpenCOM DSL to allow the construction of the architectural model that is used by the OpenCOM [27] middleware. The OpenCOM DSL used by Bencomo *et al.* can be seen as an Architecture Description Language (ADL) with generative capabilities [14].

Casquina *et al.* (Cosmapek): Casquina *et al.* propose the Cosmapek [22], an adaptive deployment infrastructure that uses techniques of SASS as a means to achieve dynamic deployment of DSPL. Cosmapek relies on a reflective architecture and implements the main MAPE-K activities. However, they perform the MAPE-K activities based on static models using runtime binding. They represent the variability in a feature model, representing static and dynamic features in a single model with its own notation, that is translated to XML for runtime use. The static features are used at design time, to generate a dynamic product. The dynamic features are used at load and runtime in order to provide variability reconfiguration at runtime. The traceability is performed by a direct link from the features of feature model to the components of the architectural model. They use UML component diagram to build the architectural model. They also proposed the DyCosmos, a language-independent dynamic component-based implementation model which extends the COSMOS* component model [49], such that the resulting components and connectors are reconfigurable at runtime allowing the reflection. The Cosmapek was implemented using DyCosmos component model and Java reflection API.

Cetina *et al.* (MoRE): Cetina *et al.* propose MoRE [23, 24], a reconfiguration engine for developing pervasive systems using SPL concepts and techniques. MoRE manages DSPLs with service-oriented architectures, where services and devices communicate using channels. Cetina *et al.* use the FAMA tool [11] to feature modeling and analysis. This approach also proposed a realization model and an architectural model called PervML. The PervML model describes pervasive systems focusing on specifying services in concrete physical environments. The PervML is technology-independent domain specific language for representing services and devices and how they are connected through channels. The realization model is an extension of Atlas Model Weaving (AMW) [41] to relate the features (feature model) with the PervML elements. AMW is a model for establishing relationships between models. MoRE uses the OSGi [4] to implement the reconfiguration.

Cubo *et al.* (Dynamic DAMASCo): Cubo *et al.* propose the Dynamic DAMASCo [28, 29], an extension of DAMASCo framework [30] that promotes the safe reuse of services in service-based systems. Dynamic DAMASCo has a hybrid architectural style, combining services and components. Also, Dynamic DAMASCo performs service discovery at runtime. Cubo *et al.* use BPEL process [82] or WF workflows[8] to complement the feature model to deal with static and dynamic variability using a multiple model feature modeling technique. This approach represents a family of services from a business process specification, using an intermediate interface model that can be generated from BPEL process or WF workflows. They also proposed the CA-STS model with Context-Aware Symbolic Transition Systems (CA-STSs). CA-STSs are extracted from the BPEL services or WF workflows, which implement the client and services, through a model transformation process. They defined CA-STS as an extension of Labelled Transition Systems (LTS) [7]. Dynamic DAMASCo was implemented in Python as a set of tools integrating DAMASCo in the toolbox ITACA.

[8]Windows Workflow Foundation. `https://msdn.microsoft.com/en-us/library/jj684582.aspx` Accessed on January 27, 2017.

Fuentes and Gamez: Fuentes and Gamez [45, 46] propose a family of aspect-oriented middleware platforms, able to enable, disable or replace the version of some services modeled as aspects depending on the available resources of small devices. They propose middleware feature model, a microkernel feature model, and a base services feature model, to represent the variability of the middleware platform family. Their approach also includes a mapping between the middleware feature model and an aspect-oriented architecture. Fuentes and Gamez uses an extension of an ADL called xADL [35] with aspect and variability to represent the architecture. This approach is based on the microkernel architectural pattern and uses the CAM/DAOP to implement its solution. CAM/-DAOP[9] [87] is a component and aspect based model and platform that applies the components and aspects as first class entities to dynamically weave at runtime;

Gomaa and Hashimoto: They [51] propose a dynamic software adaptation approach and an environment for service-oriented product lines, extending the SASSY Framework [73]. This approach is compatible with the three-layer reference architecture model for self-management [67] and is based in a hybrid architectural style, managing components, and services. They represent the variability through feature model, using the PLUSS notation and representing only dynamic features. PLUSS method [40] is used to map features to components and fulfill a mapping table, that record this traceability. The SASSY monitoring service captures context information to adapt the system, by replacing a component or service by another. The SASSY framework was developed using Eclipse Swordfish[10] that is an open-source, extensible ESB (Enterprise Service Bus) based on OSGi[4], and Apache CXF that is an open-source web services framework which supports JAX-WS, including SOAP and WSDL.

Hallsteinsen *et al.* (MADAM): Hallsteinsen *et al.* propose MADAM [55, 44, 50], an approach to building self-adaptive systems as component-based systems families with explicit modeling of variability. MADAM aims to build self-adaptive systems for mobile and distributed environments, extending the configurable product bases [100] to enable runtime adaptation. Configurable product bases are a type of SPL, in which product derivation does not involve product specific development and is often highly automated [100]. This approach provides a UML profile for modeling requirements models where the variability is made explicit. They also use the provided UML profile to include stereotypes for modeling variations of a component's functional properties of the architectural model. Nonetheless, they argue the provided UML profile can be used for modeling variations of a component's functional properties that are related to the feature-oriented techniques of FODA. MADAM relies on a reflective architecture that capture context and system information to trigger an adaptation. Although not an MAPE-K based approach, MADAM meets all MAPE-K activities, performing a control loop. To validate this approach, they implemented a pilot mobile application using Java J2ME/CDC.

Lee *et al.*: Lee *et al.* [69, 60] propose a feature-oriented product line engineering approach that is based on the feature analysis technique to support the services identification. They present a service-oriented architecture and represents the architecture by an extended notation of feature model that represents dynamic compositions and services [69]. Their proposed notation is an extension of FODA notation and represents in a single model the static and dynamic variability. This notation distinguishes the type of composition using dashed line to represent the dynamic composition and keeping static composition as a straight line. Furthermore, this approach represents dynamic service

[9] CAM/DAOP. `http://caosd.lcc.uma.es/CAM-DAOP.htm`. Accessed on April 06, 2017.
[10] Eclipse Swordfish. `https://projects.eclipse.org/projects/soa.swordfish`. Accessed on April 29, 2017.

composition into the feature model. This approach considers a feature in feature model as a service, representing three types of services: (i) *workflow services* define service transactions (behaviors), (ii) *molecular services* represent functionalities to be developed in-house as reusable assets, and (iii) *dynamic services* constitute third-party services the system uses at runtime.

They also propose a QoS framework to support the management of the dynamic services. Workflows orchestrate product lines, while a continuous monitoring, renegotiation, and acceptability loop maintains QoS and service composition integrity at runtime. They developed the QoS framework using the Apache River, which has implementation-specific connection interfaces that make it flexible enough to be applied to other SOAs such as Web services. Apache River[11] is the implementation of Jini[12] and defines a programming model which both exploits and extends Java technology to enable the construction adaptive network systems. The approach proposed by Lee *et al.* [69] has a significant advantage because the notation joins in a single model: (i) the static and dynamic variability, and (ii) the variability and architecture, performing the integrated orthogonality and not requiring traceability between the variability model and the architectural model. However, this approach is focused on service-oriented architectures, not considering component-based or hybrid architectures.

Morin *et al.* (DiVA): Morin *et al.* propose DiVA [77, 76, 75], an approach that uses model-driven engineering (MDE) and aspect-oriented modeling (AOM) techniques to support runtime variability. DiVA relies on a reflective architecture built in three levels. The bottom level contains the application logic and uses feature models at runtime for managing the variability of the system. The top level plans the adaptation. Also, the middle level creates the link between the top and the bottom level, by analyzing data from sensors and converting the data into context information useful for reasoning and by reflecting changes in the running system. The variability is represented employing feature models, mapped to an aspect of architecture. To represent the architectural model, they use an own metamodel that can be seen as a dynamic ADL to describe the running system. Nonetheless, they argue the architectural model can be build using any metamodeling language, as UML or any architectural description language (ADL). Although not an MAPE-K based approach, DiVA meets all MAPE-K activities. DiVA can be implemented using OSGi [4], Fractal [19] or OpenCom [27].

Nascimento *et al.* (ArCMAPE): Nascimento *et al.* propose ArCMAPE [80, 79], an SPL-based solution, which explores the different software fault tolerance techniques based on design diversity. In ArCMAPE approach, changes in context as the quality of network and changes in the system as faults or service unavailability triggers the adaptation. Nascimento *et al.* create an initial feature model using the notation proposed by Ferber *et al.* [42] complementing with CVL (Common Variability Language) variability model to deal with variability at runtime. This approach also encompasses two other models, CVL base model, and CVL resolution model, to decide the variability at runtime. The CVL base model is represented by UML component diagram. They use CVL resolution model and the FArM method [97] to direct link the CVL variability model (feature model) to CVL base model (UML component diagram). ArCMAPE relies on a reflective architecture based on MAPE-K. The components were implemented according to COSMOS* [49], a component implementation model, and using SOAP protocol for connecting services. They implement the reflection using the Eclipse Equinox[13], an implementation of OSGi [4].

[11] Apache River. `http://river.apache.org/`. Accessed on April 06, 2017.
[12] Jini Service Oriented Architecture. `http://river.apache.org/release-doc/current/spec-index.html`. Accessed on April 05, 2017.
[13] Eclipse Equinox. `http://www.eclipse.org/equinox/`. Accessed on April 05, 2017.

Parra *et al.* (CAPucine): Parra *et al.* propose CAPucine [86, 85], a context-aware DSPL to define a service-oriented context-aware product derivation that allows adaptation at runtime according to its context of use. The changes in the context trigger adaptation on the running system based on a set of rules. This approach uses aspect model weaving [61] for generating the architectural assets from feature model configuration at runtime. A metamodel is used at runtime to represent the architectural assets. In aspect model weaving, features are mapped to aspect models representing different aspects of a given feature in the architectural assets. The corresponding aspect models of selected features are woven into the architectural assets to create the architecture of the system. This approach exploits SCA (Service Component Architecture) models, and the dynamic binding and unbinding of referenced services provided by the FraSCAti runtime environment. FraSCAti [95] is a Fractal-based Service Component Architecture (SCA) [81] platform with dynamic properties. In order to obtain information from the environment, the authors use COSMOS [93], a context-aware framework connected to the environment through appropriate sensors.

Rosenmüller *et al.*: Rosenmüller *et al.* [92, 90, 91] propose and approach to use static and dynamic compositions to statically generate a tailor-made DSPL from a highly customizable SPL. They demonstrate how to bind features of an SPL dynamically or statically using the same code base. This approach can choose a distinct binding time per feature after development. They achieve this by statically composing the features that are used in combination with a dynamic binding unit, which is bound at runtime as a whole. Dynamic binding units are similar to components but are generated at compile time from a user-defined set of features. This approach generates binding units statically (at design time) to achieve fine-grained customizability while maximizing performance. Also, this approach dynamically applies the binding units at runtime. At runtime, the adaptation is performed by code transformation, code generation, and applying binding units. This approach relies on a reflective architecture and meets all MAPE-K activities, though it does not follow the MAPE-K. Rosenmüller *et al.* support the dynamic composition process by representing features as classes, called feature classes. Feature classes are generated in the FeatureC++ code transformation process. FeatureC++[14] is a C++ language extension to support Feature-Oriented Programming (FOP).

5 Conclusions and Future Work

This paper has focused on providing a comprehensive taxonomy for comparing DSPL approaches. We studied these approaches from two dimension namely self-adaptive dimension and DSPL variability dimensions. Using this taxonomy, fifteen prominent DSPL solutions were compared.

Regarding the Self-Adaptation Dimension, we observed most of selected DSPL solutions are autonomous, that is, in their adaptation process, there is no outside intervention. Likewise, the cause or trigger of adaptation is the context or the system, and there is only one user-triggered adaptation. Concerning binding time, a DSPL requires runtime binding, although a DSPL solution also can perform the design and/or load time binding. Besides, DSPL solutions can perform the replacement or reorganization of architecture as realization technique of adaptation. As well as the analysis by Bencomo *et al.* [15], concerning MAPE-K activities, we also conclude: (i) some solutions are not dynamic as they should be, because these solutions partially (or do not) implement autonomic activities; and (ii) among the analyzed solutions, only nine solutions perform all the activities of the MAPE-K loop, of which only three studies explicitly implement the MAPE-K loop.

[14]FeatureC++. `http://fosd.de/fcc`. Accessed on April 05, 2017.

In the context of DSPL variability dimension, DSPL solutions can deal only with dynamic variability, or with static and dynamic variability together. Most of the solutions use feature model to represent the variability, however, to represent the architectural model there are a diversity of options that include UML, ADL, and many approaches-specific models. Regarding the managed element, the most commonly used are components; however, services and aspects are also used. However, the diversity of architectural models used also reflects the diversity of traceability techniques between the variability and the architectural elements. Among the analyzed solutions, only six solutions perform traceability by a direct link from variability model to architectural model elements, commonly using OVM, and three others solutions use mapping table or mapping model. About feature modeling strategy we notice a direct relationship between the type of variability and the feature model used. When the DSPL solution represents only the dynamic variability, any of the options for modeling feature can be chosen. Conversely, when the DSPL solution represents static and dynamic variability together, then it requires the use of feature model with multiple models or a single model by an extended notation of feature model.

References

[1] Nadeem Abbas and Jesper 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*, page 117, New York, New York, USA, 2013. ACM Press. doi:10.1145/2499777.2500718.

[2] Nadeem Abbas, Jesper Andersson, and Welf Löwe. Autonomic Software Product Lines (ASPL). In *Proceedings of the Fourth European Conference on Software Architecture Companion Volume - ECSA '10*, volume 46, pages 324–331, New York, New York, USA, 2010. ACM Press. doi:10.1145/1842752.1842812.

[3] Komi S. Abotsi, S. Tonny Kurniadi, Hamad I. Alsawalqah, and Danhyung 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*, New York, New York, USA, 2011. ACM Press. doi:10.1145/2019136.2019171.

[4] OSGi Alliance. OSGi - The Dynamic Module System for Java. Available: https://www.osgi.org/.

[5] Vander Alves, Daniel Schneider, Martin Becker, Nelly Bencomo, and Paul Grace. Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems. In *Proceedings of the Third International Workshop on Variability Modelling of Software-intensive Systems*, pages 9–17, Sevilla, Spain, 2009.

[6] Jesper Andersson, Rogerio de Lemos, Sam Malek, and Danny 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)*, volume 5525 LNCS, pages 27–47. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-02161-9_2.

[7] André Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[8] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Service-Oriented Dynamic Software Product Lines. *Computer*, 45(10):42–48, 2012. Available:

http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6269872http://ieeexplore.ieee.org/document/6269872/, doi:10.1109/MC.2012.289.

[9] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Dynamic Software Product Line Engineering: A Reference Framework. *International Journal of Software Engineering and Knowledge Engineering*, 27(1):1–44, 2017. doi:10.1142/S021819401700894X.

[10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. doi:10.1016/j.is.2010.01.001.

[11] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *First International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 129–134, 2007.

[12] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering (17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Proceedings)*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005. doi:10.1007/11431855_34.

[13] Nelly Bencomo, Gordon Blair, Carlos Flores, and Pete Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *Second International Workshop on Variability Modelling of Software-Intensive Systems*, pages 141–150, 2008. doi:10.1.1.149.4822.

[14] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon 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*, page 811, New York, New York, USA, 2008. ACM Press. doi:10.1145/1368088.1368207.

[15] Nelly Bencomo, Jaejoon Lee, and Svein 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)*, pages 61–68, 2010.

[16] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004. doi:10.1016/S0164-1212(03)00080-3.

[17] Jan Bosch, Rafael Capilla, and Rich Hilliard. Trends in Systems and Software Variability. *IEEE Software*, 32(3):44–51, may 2015. doi:10.1109/MS.2015.74.

[18] Yuriy Brun, Giovanna Di, Marzo Serugendo, Cristina Gacek, Holger Giese, and Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, 5525:48–70, 2009. doi:10.1007/978-3-642-02161-9_3.

[19] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. Available: http://dx.doi.org/10.1002/spe.767.

[20] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 1996.

[21] Rafael Capilla and Juan C. Dueñas. Modelling variability with features in distributed architectures. In Frank van der Linden, editor, *Software Product-Family Engineering: 4th International Workshop, PFE 2001 Bilbao, Spain, October 3–5, 2001 Revised Papers*, pages 319–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi:10.1007/3-540-47833-7_28.

[22] Junior Cupe Casquina, Jane Dirce Alves Sandim Eleuterio, and Cecilia Mary Fischer Rubira. Adaptive Deployment Infrastructure for Android Applications. In *12th European Dependable Computing Conference Adaptive*, pages 218–228, 2016. doi:10.1109/EDCC.2016.25.

[23] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying Software Product Lines to Build Autonomic Pervasive Systems. In *2008 12th International Software Product Line Conference*, pages 117–126. Universidad Politécnica de Valencia, IEEE, 2008. doi:10.1109/SPLC.2008.13.

[24] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42(10):37–43, 2009. doi:10.1109/MC.2009.309.

[25] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-02161-9_1.

[26] Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series. Addison-Wesley, 1 edition, 2001.

[27] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42, 2008. doi:10.1145/1328671.1328672.

[28] Javier Cubo, Nadia Gamez, Lidia Fuentes, and Ernesto 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*, volume 7925 of *Lecture Notes in Computer Science*, pages 326–342. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38977-1_25.

[29] Javier Cubo, Nadia Gamez, Ernesto Pimentel, and Lidia Fuentes. Reconfiguration of Service Failures in DAMASCo Using Dynamic Software Product Lines. In *2015 IEEE International Conference on Services Computing*, pages 114–121. IEEE, 2015. doi:10.1109/SCC.2015.25.

[30] Javier Cubo and Ernesto Pimentel. DAMASCo: A Framework for the Automatic Composition of Component-Based and Service-Oriented Architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture (5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings)*, volume 6903 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-23798-0.

[31] Krysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[32] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002. `doi:10.1007/3-540-45821-2_10`.

[33] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In Robert L. Nord, editor, *Software Product Lines (Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings)*, volume 3154 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004. `doi:10.1007/978-3-540-28630-1_17`.

[34] Jackson Raniel F. da Silva, Francisco Airton P. da Silva, Leandro M. do Nascimento, Dhiego A. O. Martins, and Vinicius C. Garcia. The dynamic aspects of product derivation in DSPL: A systematic literature review. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 466–473. IEEE, 2013. `doi:10.1109/IRI.2013.6642507`.

[35] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, 2005. `doi:10.1145/1061254.1061258`.

[36] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems : A Second Research Roadmap. *Software Engineering for Self-Adaptive Systems II*, 7475:1–32, 2013. `doi:10.1007/978-3-642-35813-5_1`.

[37] Jane Dirce Alves Sandim Eleuterio, Felipe Nunes Gaia, Andrea Bondavalli, Paolo Lollini, Genaina N. Rodrigues, and Cecília Mary Fischer 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)*, pages 323–330, 2016. `doi:10.1109/SEAA.2016.40`.

[38] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, page 7. ACM Press, 2010. `doi:10.1145/1882291.1882296`.

[39] Active Endpoints. Communicating with the ActiveBPEL Server Administration Interface via Web Services. Available: `http://www.activevos.com/content/developers/education/sample{_}active{_}bpel{_}admin{_}api/doc/index.html`.

[40] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations. In *Software Product Lines. 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005. Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin Heidelberg, 2005. `doi:10.1007/11554844_5`.

[41] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006. `doi:10.1.1.100.7255`.

[42] Stefan Ferber, Jürgen Haag, and Juha Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In Gary J. Chastek, editor, *Software product lines - Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings*, volume 2379 of *Lecture Notes in Computer Science*, pages 235–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. `doi:10.1007/3-540-45652-X_15`.

[43] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California, 2000.

[44] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006. `doi:10.1109/MS.2006.61`.

[45] Lidia Fuentes and Nadia 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*, pages 11–16. ACM, 2008. `doi:10.1145/1408620.1408623`.

[46] Lidia Fuentes and Nadia Gámez. Modeling the Context-Awareness Service in an Aspect-Oriented Middleware for AmI. In *3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008*, volume 51, pages 159–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. `doi:10.1007/978-3-540-85867-6_19`.

[47] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in Software Systems – A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014. `doi:10.1109/TSE.2013.56`.

[48] David Garlan, Shang-Wen Wen Cheng, An-Cheng Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004. `doi:10.1109/ICAC.2004.1301377`.

[49] Leonel Aguilar Gayard, Cecilia Mary Fischer Rubira, and Paulo Asterio de Castro Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, Institute of Computing, University of Campinas, February 2008.

[50] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, R. Fricke, E. Gjorven, Svein Hallsteinsen, G. Horn, M. U. Khan, Alessandro Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and Erlend Stav. A comprehensive solution for application-level adaptation. *Software: Practice and Experience*, 39(4):385–422, 2009. `doi:10.1002/spe.900`.

[51] Hassan Gomaa and Koji Hashimoto. Dynamic software adaptation for service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference on - SPLC '11*, 2, page 1, New York, New York, USA, 2011. ACM Press. `doi:10.1145/2019136.2019176`.

[52] M.L. Griss, J. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings. Fifth International Conference on Software Reuse*, pages 76–85, Victoria, BC, 1998. IEEE. doi:10.1109/ICSR.1998.685732.

[53] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Simple Object Access Protocol (SOAP), Version 1.2, W3C Recommendation, 27 April 2007, 2007. Available: https://www.w3.org/TR/soap12/.

[54] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008. doi:10.1109/MC.2008.123.

[55] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using Product Line Techniques to Build Adaptive Systems. In *10th International Software Product Line Conference (SPLC'06)*, pages 141–150. IEEE, 2006. doi:10.1109/SPLINE.2006.1691586.

[56] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *2008 12th International Software Product Line Conference*, pages 139–148. IEEE, sep 2008. doi:10.1109/SPLC.2008.25.

[57] Andreas Hein, Michael Schlick, and Renato Vinga-Martins. Applying Feature Models in Industrial Settings. In *Software Product Lines: Experience and Research Directions*, volume 576 of *The Springer International Series in Engineering and Computer Science*, pages 47–70. Springer US, Boston, MA, 2000. doi:10.1007/978-1-4615-4339-8_3.

[58] IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 36(June):34, 2006. doi:10.1021/am900608j.

[59] ISO/IEC/IEEE. Systems and software engineering – Architecture description. Technical Report ISO/IEC/IEEE 42010:2011, IEEE, 2011. doi:10.1109/IEEESTD.2011.6129467.

[60] Jaejoon Lee and Gerald Kotonya. Combining Service-Orientation with Product Line Engineering. *IEEE Software*, 27(3):35–41, 2010. doi:10.1109/MS.2010.30.

[61] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008. doi:10.1007/s10270-008-0080-5.

[62] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[63] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. doi:10.1023/A:1018980625587.

[64] Kyo C. Kang and Hyesun Lee. Variability Modeling. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, pages 25–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36583-6_2.

[65] Kyo C. Kang and Hyesun Lee. Variability Modeling. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, chapter 4, pages 25–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36583-6_2.

[66] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. doi:10.1109/MC.2003.1160055.

[67] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE, may 2007. doi:10.1109/FOSE.2007.19.

[68] Kim Lauenroth and Klaus Pohl. Principles of Variability. In *Software Product Line Engineering – Foundations, Principles, and Techniques*, chapter 4, pages 57–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/3-540-28901-1_4.

[69] Jaejoon Lee, Gerald Kotonya, and Daniel Robinson. Engineering Service-Based Dynamic Software Product Lines. *Computer*, 45(10):49–55, 2012. doi:10.1109/MC.2012.284.

[70] Jaejoon Lee and Dirk Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12):55, dec 2006. doi:10.1145/1183236.1183266.

[71] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, 1995. doi:10.1109/32.385971.

[72] Pattie Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, dec 1987. doi:10.1145/38807.38821.

[73] Sam Malek, Naeem Esfahani, Daniel A. Menasce, Joao P Sousa, and Hassan Gomaa. Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models. In *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 62–69. IEEE, 2009. doi:10.1109/PESOS.2009.5068821.

[74] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H C Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004. doi:10.1109/MC.2004.48.

[75] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009. doi:10.1109/MC.2009.327.

[76] Brice Morin, Olivier Barais, Gregory Nain, and Jean-marc Jezequel. Taming Dynamically Adaptive Systems using models and aspects. In *2009 IEEE 31st International Conference on Software Engineering*, pages 122–132, Vancouver, BC, 2009. IEEE. doi:10.1109/ICSE.2009.5070514.

[77] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *Model Driven Engineering Languages and Systems*, volume 5301 LNCS of *Lecture Notes in Computer Science (LNCS)*, pages 782–796. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-87875-9_54.

[78] Hausi A. Müller, Holger M. Kienle, and Ulrike Stege. Autonomic Computing Now You See It, Now You Don't. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 32–54. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-540-95888-8_2.

[79] Amanda S. Nascimento, Cecilia M. F. Rubira, Rachel Burrows, and Fernando Castor. A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL. In *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*, pages 119–128. IEEE, 2013. doi:10.1109/SBCARS.2013.23.

[80] Amanda S. Nascimento, Cecilia M.F. Rubira, and Fernando Castor. ArCMAPE: A Software Product Line Infrastructure to Support Fault-Tolerant Composite Services. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 41–48. IEEE, 2014. doi:10.1109/HASE.2014.15.

[81] OASIS. Service Component Architecture (SCA), 2007. Available: http://www.oasis-opencsa.org/sca.

[82] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[83] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering - ICSE Companion '08*, pages 899–910, Vancouver, BC, 2008. ACM. doi:10.1145/1370175.1370181.

[84] Peyman Oreizy, Nenad Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, volume 1998, pages 177–186. IEEE Comput. Soc, 1998. doi:10.1109/ICSE.1998.671114.

[85] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Science of Computer Programming*, 76(12):1247–1260, 2011. doi:10.1016/j.scico.2010.12.005.

[86] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In *SPLC '09 Proceedings of the 13th International Software Product Line Conference*, pages 131–140, San Franscisco, CA, 2009. ACM.

[87] Mónica Pinto, Lidia Fuentes, and José María Troya. A Dynamic Component and Aspect-Oriented Platform. *The Computer Journal*, 48(4):401–420, may 2005. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/bxh083, doi:10.1093/comjnl/bxh083.

[88] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering – Foundations, Principles, and Techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/3-540-28901-1.

[89] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *6th World Conference on Integrated Design & Process Technology*, pages 23–27, Pasadena, CA, USA, 2002. Society for Design and Process Science.

[90] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, jun 2011. doi:10.1007/s10515-011-0080-5.

[91] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *Proceedings of the 10th ACM international conference on Generative*

programming and component engineering - GPCE '11, pages 3–12, New York, New York, USA, 2011. ACM Press. doi:10.1145/2047862.2047866.

[92] Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven 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*, page 3, New York, New York, USA, 2008. ACM Press. doi:10.1145/1449913.1449917.

[93] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software Architecture Patterns for a Context-Processing Middleware Framework. *IEEE Distributed Systems Online*, 9(6):13, 2008. doi:10.1109/MDSO.2008.17.

[94] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009. doi:10.1145/1516533.1516538.

[95] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *2009 IEEE International Conference on Services Computing*, pages 268–275. IEEE, 2009. doi:10.1109/SCC.2009.27.

[96] Liwei Shen, Xin Peng, Jindu Liu, and Wenyun Zhao. Towards Feature-oriented Variability Reconfiguration in Dynamic Software Product Lines. *Top Productivity through Software Reuse*, 6727:52–68, 2011. doi:10.1007/978-3-642-21347-2_5.

[97] P. Sochos, M. Riebisch, and I. Philippow. The feature-architecture mapping (FArM) method for feature-oriented development of software product lines. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pages 9 pp.–318. IEEE, 2006. doi:10.1109/ECBS.2006.69.

[98] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005. doi:10.1002/spe.652.

[99] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *2011 15th International Software Product Line Conference*, pages 191–200. IEEE, aug 2011. doi:10.1109/SPLC.2011.53.

[100] Frank van der Linden, Jan Bosch, Erik Kamsties, Kari Känsälä, and Henk Obbink. Software Product Family Evaluation. In Robert L. Nord, editor, *Third International Conference, SPLC 2004*, pages 110–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-28630-1_7.

[101] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE Comput. Soc, 2001. doi:10.1109/WICSA.2001.948406.

[102] Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II*, pages 265–293. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-35813-5_11.

[103] Danny Weyns, Sam Malek, and Jesper Andersson. FORMS: a FOrmal Reference Model for Self-adaptation. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, page 205. ACM Press, 2010. doi:10.1145/1809049.1809078.

[104] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On Patterns for Decentralized Control in Self-Adaptive Systems. *Lecture Notes in Computer Science*, 7475:76–107, 2013. doi:10.1007/978-3-642-35813-5_4.