International Journal of Software Engineering
and Knowledge Engineering
Vol. 27, No. 2 (2017) 191–234

DOI: 10.1142/S0218194017500085

World Scientific
www.worldscientific.com

Dynamic Software Product Line Engineering: A Reference Framework

Mahdi Bashari[*,‡,§], Ebrahim Bagheri[†,¶] and Weichang Du[*,‖]

[*]*Faculty of Computer Science, University of New Brunswick*
Fredericton, Canada

[†]*Department of Electrical and Computer Engineering*
Ryerson University, Toronto, Canada
[‡]*mbashari@unb.ca*
[§]*mahdy.bashary@gmail.com*
[¶]*bagheri@ryerson.ca*
[‖]*wdu@unb.ca*

Received 4 March 2015
Revised 22 September 2015
Accepted 24 February 2016

Runtime adaptive systems are able to dynamically transform their internal structure, and hence their behavior, in response to internal or external changes. Such transformations provide the basis for new functionalities or improvements of the non-functional properties that match operational requirements and standards. Software Product Line Engineering (SPLE) has introduced several models and mechanisms for variability modeling and management. Dynamic software product lines (DSPL) engineering exploits the knowledge acquired in SPLE to develop systems that can be context-aware, post-deployment reconfigurable, or runtime adaptive. This paper focuses on DSPL engineering approaches for developing runtime adaptive systems and proposes a framework for classifying and comparing these approaches from two distinct perspectives: adaptation properties and adaptation realization. These two perspectives are linked together by a series of guidelines that help to select a suitable adaptation realization approach based on desired adaptation types.

Keywords: Dynamic software product line; self-adaptive software; runtime adaptation; variability management; survey.

1. Introduction

In order to provide better service in many areas, software systems require behavior modification at runtime particularly in response to users' dynamically varying needs as well as environmental constraints [1]. Such changes could be on the features of the system

or on how the system caters those features. This is referred to as *runtime adaptation*, and a system capable of deciding about and performing runtime adaptation is called a *runtime adaptive* system. Developing runtime adaptive systems has been a significant concern in many research areas such as mobile computing, autonomic computing, robotics and ubiquitous computing [2].

Runtime adaptation is a complex process and runtime adaptive systems are at times prone to be unstable, inefficient and unreliable [3]. Furthermore, not taking a systematic approach toward adaptation design results in a system which is complex and an adaptation mechanism which is hard to modify [1]. Therefore, there have been efforts for the systematic development of runtime adaptive systems by separating the application logic from adaptation logic [4]. Still, many of these approaches suffer from limitations such as being domain specific, having low performance or limited adaptability [3]. Therefore, the research community still tries to introduce more efficient frameworks and processes for the development of runtime adaptive systems [2].

The *Software Product Line Engineering* (SPLE) paradigm [5] suggests an effective way to deal with the variability of similar products especially when satisfying requirements of different operating environments and users. This group of similar products is called a *product family* which is a group of products sharing common features. In SPLE, the development of a system that fits the user-needs and operating environment relies heavily on the reuse of assets through two development phases: *domain engineering* and *application engineering* [5]. In the domain engineering phase, those reusable assets needed for developing products are specified and built. These assets consist of common parts and variation points. In the application engineering phase, the target product for the specific operating environment and user requirements are derived using the reusable assets developed in the domain engineering phase. The binding of these assets into a product is mostly performed at design, compile or link time, after which the system stays the same during its lifetime.

SPLE and runtime adaptive systems have much in common [6]. The software product line paradigm (SPL) is a solution for managing the variability of products in a product family. In SPLE, the variability of the products is captured at the domain engineering phase, and the best variant for a specific operating environment and user requirements is selected during the application engineering phase. As such, designing a runtime adaptive system may be considered to be a variability management problem, where first and foremost the variability of the system is captured at design time, and then the best product variant is selected at runtime according to context requirements. The fact that both of these two paradigms are dealing with variability management, one as the problem and one as a solution, has motivated researchers to consider the synergy of these two paradigms in *dynamic software product line* (DSPL) engineering [7].

DSPL is used for software development in different application areas such as context-aware systems, post-deployment reconfiguration, and runtime adaption. In DSPL engineering for runtime adaptation, SPL models and techniques are used at runtime to manage and decide about the variability of runtime adaptive systems [8].

SPL-inspired models and mechanisms are used at runtime to select the most suitable variant of the system and adapt the system to that variant. In this paper, we focus on DSPL engineering approaches for realizing runtime adaptation by presenting a framework for classifying and comparing adaptation properties and implementation concerns. Additionally, we suggest guidelines for the adoption or design of DSPL engineering approaches using the recommended framework. In order to do so, we first review key adaptation properties. Subsequently, we identify realization concerns for the development of DSPL. Finally, we link adaptation properties to the idenfitied realization concerns by providing guidelines for the selection of suitable models and techniques.

The proposed framework is designed by defining a set of dimensions which answer questions about how runtime adaptation could be realized using DSPL engineering approaches. These dimensions are defined such that they can be generally used to characterize a DSPL engineering approach. In the framework, these dimensions are clearly described, the possible values are discussed, and examples from existing DSPL engineering approaches are provided. In order to organize these dimensions, our framework conceptualizes DSPL adaptation management as a MAPE-K loop [9]. Using this conceptualization, our framework dimensions are organized in a taxonomy in which the steps of the MAPE-K loop are used to classify different dimensions of adaptation realization.

In the following sections, we define what we mean by DSPL and then select seven of the prominent approaches from the literature to which we shall apply the proposed framework (Sec. 2). Subsequently, we describe what we mean by adaptation through defining its properties and by comparing the selected approaches based on the type of adaptation they offer (Sec. 3). Furthermore, the framework and its dimensions are described, and this time the framework will be used to compare adaptation realization in the selected approaches (Sec. 4). After describing the framework, adaptation dimensions and framework dimensions are linked together by a set of guidelines which are used for selecting various values for the dimensions of the proposed framework. The guidelines provide strengths and weaknesses of each possible value according to the adaptation required when either building a new DSPL engineering approach or choosing from already existing approaches (Sec. 5).

2. Dynamic Software Product Lines

Although SPLE models and techniques are utilized for managing variability in DSPL paradigms, SPLE and DSPL engineering paradigms follow different goals, each of which results in products with different nature. Hallsteinsen *et al.* [10] characterize DSPL engineering with a set of properties that distinguishes it from SPL engineering in terms of the goal and properties of their products. For instance, the goal for selecting a different system variant in SPL is satisfying specific market needs; while, the goal in the DSPL paradigm is satisfying the requirements of different running contexts [7]. Hallsteinsen *et al.* [10] also define two types of properties for DSPL engineering

products that make them distinct from SPLE products: *runtime reconfiguration* and *self-management*. Runtime reconfiguration properties represent a product's ability to 'configure and bind variability at runtime', 'bind changes several times during lifetime', 'change in variation points', and 'deal with unexpected changes' [10]. Self-management properties represent a product's ability to be 'context-aware', 'self-adaptive', and function as an 'automatic decision maker' [10]. Runtime reconfiguration properties are considered mandatory for the product, but self-management properties are deemed as optional within the context of DSPL.

Several approaches and frameworks, which may or may not call themselves a DSPL engineering approach, have been proposed that exploit SPL models and techniques in the development of different systems such as service-oriented and context-aware systems. In this study, in order to distinguish DSPL approaches from other engineering approaches, we have adopted the DSPL definition offered by Hallsteinsen *et al.* [10]. Hence, we consider an approach DSPL engineering approach if it uses SPL assets in its development and has runtime reconfiguration properties. Considering this definition as a guideline, we used Google Scholar to locate DSPL studies using search terms 'DSPL', 'Software Product Line Runtime adaptation', and 'Software Product Line Self-adaptive' with no restriction set on the date or place of publication. The results, and other studies cited within these publications, were reviewed to determine whether they matched the Hallsteinsen DSPL definition for inclusion in the study pool. The studies documenting different aspects of a specific approach were grouped together. A list of these study groups can be found in Table A.1 in Appendix A.

We have selected seven approaches from these studies, which will be discussed in a comparative scale using our framework. Our criteria for the selection of these seven approaches were as follows:

- *Select studies which propose a development methodology.* The studies were classified by their focus into four groups: conceptual discussions, development methodologies, models and languages, and other (tools, algorithms,...). Using this classification, only those studies which propose a development methodology are considered for selection.
- *Select studies with greater impact on the trends in the field.* The sum of citation counts provided by Google Scholar for the study group documenting the approach has been selected as the criterion for measuring the impact of the research on the field.
- *Select studies with well-documented adaptation realization procedures.* In the case of development methodologies, the studies were reviewed to determine if they have addressed each of the different steps of adaptation realization and relevant knowledge in the MAPE-K loop (i.e. Monitoring & Analysing, Planning, and Execution).
- *Select studies which propose an original approach.* Approaches were reviewed to see if they were original or built as extensions on other approaches. The latter group were excluded.

The above criteria were used to build a table which is used as reference for the selection of the seven reviewed approaches (Appendix A). Table 1 shows the selected approaches, their relation to SPLE and their application domain. A brief overview of these approaches is as follows:

Baresi *et al.*: This DSPL engineering approach utilizes SPL, Common Variability Language (CVL) [28] and aspect-oriented programming for enabling runtime reconfiguration of business processes represented with Business Process Execution Language (BPEL). BPEL is a language for the formal specification of business process behavior focusing exclusively on Web services. The approach by Baresi *et al.* has been built as an implementation of CVL, which is a suggested generic approach for managing variability in domain specific languages. In this CVL-based approach, the variants of the system are chosen by selecting the desired features from the feature model of the system. The selected features in the feature model are mapped to series of substitutions in the base BPEL code. Base BPEL code is a process which is considered as a reference point for defining variability. A substitution is defined as placing a BPEL aspect from the CVL library into placement location in the base code. The target business process is derived using these placements after which it replaces the current business process. Runtime reconfiguration has been enabled in this approach by extending the ActiveBPEL engine [29] using aspect-oriented programming to modify BPEL at runtime by changing the included aspects [30].

Bencomo *et al.* (Genie): Genie is a model-based approach for developing self-adaptive systems for grid mobile computing and embedded systems. In this

Table 1. Summary of the selected approaches.

Approach	References	Relation to SPLE	Application domain
Baresi *et al.*	[11]	Uses feature models at runtime for variability modeling, feature model configuration for variant selection.	General
Bencomo *et al.* (Genie)	[12–15]	Uses Orthogonal variability modeling (OVM) at runtime.	Grid and mobile computing, embedded systems
Cetina *et al.* (MoRE)	[16–18]	Uses feature models at runtime for variability modeling, feature model configuration for variant selection.	Smarthomes
Floch *et al.* (MADAM)	[7, 19, 20]	Extends configurable product families with runtime reconfiguration.	Mobile and distributed systems
Gomaa *et al.* (REPFLC)	[21, 22]	Extends SPLE with reconfiguration patterns to support runtime adaptation.	General
Morin *et al.* (DiVA)	[23–25]	Uses feature model at runtime for variability modeling, feature model configuration for variant selection, uses aspect model weaving to generate system architecture.	General
Parra *et al.* (CAPucine)	[26, 27]	Uses feature models at runtime for variability modeling, feature model configuration for variant selection, uses aspect model weaving to generate system architecture.	General

approach, two variability dimensions are recognized for a system and linked to each other in the development process: *environment* or *context variability* which represents variability in the environment and *structural variability* which represents variability in the architecture of the running system. The structure of the system can adapt according to changes in the environment by linking these two dimensions. In the Genie approach, the environment is represented by a state transition diagram, where states represent the states of the environment and the transitions represent possible environment state changes which are guarded by conditions over the context of the system. The structural variability of the system is represented with OpenCOM DSL which is a domain specific language that represents system architecture. This approach takes advantage of the OVM [5] for linking those models which represent two dimensions. OVM traditionally is used for tracing variability between models at different levels of abstraction in the software product line. Genie enables runtime reconfiguration using OpenCOM [31] middleware.

Cetina *et al.* (MoRE): MoRE is a reconfiguration engine which incorporates SPLE ideas for developing autonomic pervasive systems such as Smarthomes. This approach uses feature models as a variability space model at runtime. Runtime adaptation takes place in this approach by modifying the current feature model configuration of the system using a set of condition/resolution rules, where these rules define which features are activated/deactivated as context conditions change. The system managed by MoRE should be built using service-oriented architectures, where services and devices communicate using channels. The actual system adaptation is performed by mappings features of the feature model to the services and channels that realize these specific features. Therefore, a feature model configuration is mapped to a configuration of the system, where a set of services and channels are active while the others are inactive. This approach also proposes a technology-independent, domain-specific language named PervML [32] representing system architecture. PervML is used for representing services and devices and how they are connected though channels.

Floch *et al.* (MADAM): MADAM is a utility-based approach for building self-adaptive systems for mobile and distributed environments. This approach extends an SPL engineering concept named *configurable product bases* [33] to enable runtime adaptation. Configurable product bases is a type of SPLE approach which requires no product specific development during application engineering. The MADAM approach extends configurable product bases by adding a utility-based planner to it and enabling configurable product bases to reconfigure at runtime. In this utility-based approach, a utility function is defined over the system properties and its context to represent desirability of the current configuration in the current context. Therefore, the goal of adaptation becomes the maximization of the value of this function. The planner functions by taking advantage of property predictor functions, which can predict different properties of the system according to the system configuration and the context. When the utility becomes unacceptable, the planner uses

a brute-force technique to find a configuration with the highest predicted utility and adapts to it.

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

Morin *et al.* (DiVA): This approach builds on ideas from SPLE and aspect-oriented modeling to develop self-adaptive systems. In this approach, the architecture of the system is built in three layers. The bottom layer contains the application logic; the top layer plans the adaptation; and the middle layer creates the link between the top and the bottom layers. The middle layer creates the bottom to top link by analyzing data from sensors and converting the data into context information useful for reasoning. It also creates the top to bottom link by reflecting changes in the system architecture model into the running system. The top layer uses feature models at runtime for managing the variability of system at an abstract level. Feature models are used to decide about the best configuration for the current context using a reasoner. Planning results in a feature model configuration which represents what system features should be available in this context. This feature model configuration is then mapped to the architectural model of the system using aspect model weaving. The approach uses a system variant configuration checker to ensure consistency of the target architecture model at runtime.

Parra *et al.* (CAPucine): In this approach, software product line and service oriented architecture have been used to create a process to build systems that monitor context changes in order to dynamically incorporate required assets at runtime. The variability of the system is kept modeled using a feature model. The approach gathers context information using a context sensing middleware which has been built based on the COSMOS framework. The changes in the context triggers changes in the feature model based on a set of rules. This approach suggests the use of *aspect model weaving* [34] for generating the architecture model of the system from feature model configuration at runtime. In aspect model weaving, features are mapped to aspect models representing different aspects of a given feature in the system architecture. For a feature model configuration, the corresponding aspect models of selected features are woven into the base model of the system to create an integrated model of the architecture of the system. Then the approach uses FraSCAti [35] which is a Fractal-based Service Component Architecture (SCA) [36]

platform with dynamic properties, to reconfigure the system according to the target architecture model at runtime.

Although systems with the ability to adapt at runtime can be built using any of these approaches, the types of adaptation offered by them are different. Therefore, in what follows, we provide a method for characterizing adaptation achieved by each specific approach. Knowing the type of adaptation which is offered by systems developed using each approach can help in the selection of a suitable one according to specific adaptation needed for a problem domain.

3. Characterizing Adaptation Properties in DSPL

Runtime adaptation covers a wide range of runtime changes. In order to arrive at a better understanding of the kind of adaptations which are addressed by existing DSPL engineering approaches, we need to characterize the adaptations that can be performed by each approach. The characteristics of alternative adaptations can be determined by a number of points of difference (also called dimensions). These points refer to the properties of an adaptation itself regardless of techniques and tools which are used to realize it. In [37], 20 dimensions have been proposed for characterizing an adaptation in a specific system. Therefore, we use these dimensions for comparing the adaptation properties that the different approaches offer. Figure 1 shows the dimensions classified in three categories. Those dimension whose values cannot be determined for a DSPL approach were removed from the figure. The dimensions in each of the goal, cause, and mechanism categories represent characteristics of the goal of an adaptation, source of changes that causes an adaptation, and mechanism

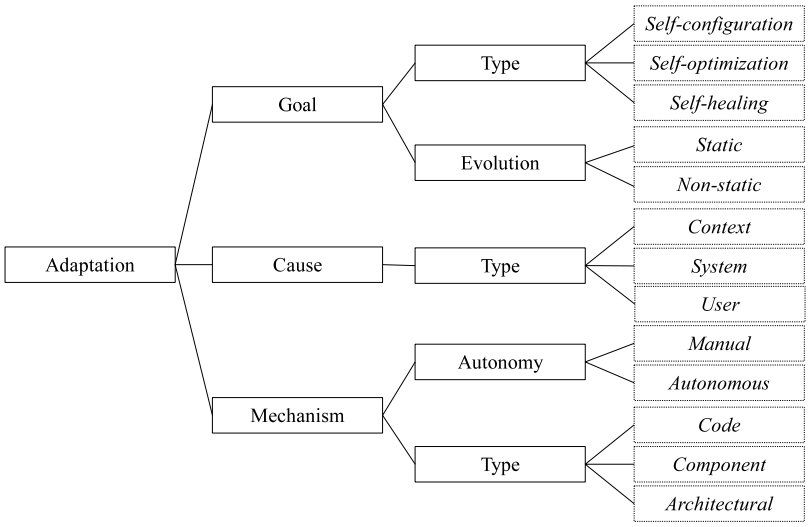

Fig. 1. Dimensions characterizing an adaptation.

Table 2. Adaptation dimension of DSPL engineering approaches.

	Goal		Cause	Mechanism	
Approach	Type	Evolution	Type	Autonomy	Type
Baresi *et al.*	—	Static	User	Manual	Code-level
Bencomo *et al.* (Genie)	Self-configuring	Static	Context	Autonomous	Component
Cetina *et al.* (MoRE)	Self-healing / self-configuring	Static	Context	Autonomous	Component
Floch *et al.* (MADAM)	Self-optimizing	Static	Context/ System	Autonomous	Component
Gomaa *et al.* (REPFLC)	—	Static	—	Autonomous	Component
Morin *et al.* (DiVA)	—	Non-static	—	Autonomous	Component
Parra *et al.* (CAPucine)	Self-configuring	Static	System	Autonomous	Component

used for enabling adaptation, respectively. In Table 2, we have listed these dimensions and their values for different DSPL engineering approaches. Each of these dimensions has been discussed in the following.

3.1. *Goal type*

This dimension is concerned with the intent for adding runtime adaptation. In the definition of autonomic computing [9], Kephart and Chess cite four goal types for self-managing systems: self-configuring, self-optimizing, self-healing and self-protecting. Goals for enabling runtime adaptation can be categorized into these categories. In the following, these goal types have been discussed, except that of the self-protection aspect due to the yet unavailability of a DSPL engineering approach with this goal type.

- *Self-configuring:* The goal here is to enable the system to automatically reconfigure itself as the context or requirements change such that the system satisfies the high level policies. This is similar to product configuration in SPLE, where a product is configured for different situations such that it satisfies stakeholders' needs. The goals can be defined as a condition that should be always satisfied for the corresponding system; for example, the response time of a specific service of the system should be always lower than a specific duration. In [12], the Genie approach was used to develop a system that runs on sensor nodes in a sensor network for flood monitoring, where the system needs to be self-configuring. In this specific problem domain, the goal of adaptation was to enable the system to switch from the battery preserving configuration to the high performance configuration so as to be able to communicate faster when flood was likely to occur.
- *Self-optimizing:* A system supports self-optimizing adaptation when the goal of enabling adaptation is to make the system perform optimally under different environments and contexts. Usually, the optimality of the system is represented with a single utility value which is typically the weighted sum of different quantitative attributes of the system (such as QoS attributes). For example, In [7],

MADAM was used to develop a supporting application for service technicians working in geographically dispersed installations. The application adapts itself to changes in the context (such as connectivity, noise, etc.) to optimize user experience represented with a utility value.

- *Self-healing*: In self-healing adaptation, the goal of adaptation is to have a system that adapts itself after a failure in order to reduce the impact of failure on the system. A self-healing system enables search for alternatives when a component of the system fails. For example, the MoRE approach is used for developing self-healing Smarthomes, where failure of devices is likely to happen [16]. In this example, failure of devices is handled by exploiting alternative ways; for example, alarm failure results in the blinking lights of the house. As an another example, Abotsi *et al.* [38] developed a DSPL engineering approach for developing a Web-based application, where the system is monitored for critical conditions. In case of a critical condition, a suitable variant of the system is identified and selected to recover from the critical condition. For instance variants that use less resource demanding contents are selected when there is lack or dearth of resources.

3.2. *Goal evolution*

This dimension denotes whether the goal of adaptation can change at runtime or not. The change in the goal is usually required when the system faces an unforeseen situation not supported by the adaptation manager of the system. The ability to evolve the adaptation goal can range from static, where the goal of adaptation does not change to dynamic, where the system itself can modify goals of adaptation during the systems lifetime:

- *Static:* In this type of adaptive systems, the goal of adaptation stays the same during the lifetime of the adaptive systems. These systems usually have a fixed adaptation policy and system variants. In the case that new goals are needed, the system should be stopped and modified to support new goals. However, the verification of these systems is usually easier considering their limited flexibility.
- *Non-static:* Here, the system adaptation manager provides a mechanism to introduce new adaptation goals to the system at runtime [23, 39, 40]. This can range from manual change of goals to evolution of the system by itself. The DiVA project [23] has claimed that it has enabled manual change of system goals by providing interfaces for modification of models both at design time and runtime. Another notable research in enabling runtime goal evolution in DSPL proposes an approach for supporting feature model changes as a result of runtime goal changes [39]. In this approach, a *meta variability model* is defined over the variation points of the system, which provides the information required for anticipating goal updates and explicitly expressing how these goal updates are supported in the feature model. For example, this meta model specifies which variation points may have new variants as the systems evolves and what are the requirements of these new variants.

3.3. *Cause type*

The trigger of an adaptation influences how monitoring and planning activities operate in a runtime adaptive system. The types of changes which can trigger adaptation may be generally classified as context changes, system, and user changes.

- *Context:* Context changes are those changes that take place in the external environment of a system. When the system needs to responds to context changes, it requires the right sensors to measure the properties of interest in real-time. The gathered information is then used in planning for adaptation. Therefore, a mechanism should be provided by the engineering approach for capturing the context information. For example, CAPucine uses the COSMOS framework for monitoring the context. In this framework, a component is developed for each sensor named context node, which represents that sensor as a component.
- *System:* System changes are those changes that take place in a system internally. Examples of these changes are failure of a component, performance of a component, and exceptions. For capturing these type of changes, sensors should be integrated within the system implementation. For example in [19], sensors are used as part of the architecture, monitoring the performance of components such as component response time and memory usage. These information are then used by the adaptation manager for planning about possible system adaptations.
- *User:* Changes are alterations in user requirements or priorities at runtime, and therefore the trigger for adaptation is an external entity such as a user. User preferences are usually captured using an interface. The change in requirements is expressed in high level terms such as features, and is realized by an adaptation manager. In the Helleboogh *et al.* approach [39], the user of the system can change the operation mode of an automotive transportation system in a warehouse in response to changes in the operating condition. For example, when a load of goods arrive at the warehouse, the operation mode of the system is changed by the user to efficiently unload the packages. Another example for adaptation as a result of user preference changes is that proposed by Wolfinger *et al.* [41] where the users' preferences are captured by wizard-like dialogues following a decision model. In each dialogue, the user selects from available alternatives in a decision point while the system offers the consequence of selecting each alternative. The adaptation target configuration for the system is built based on user decisions captured by the wizard.

3.4. *Mechanism autonomy*

This dimension defines if an outside entity (mostly humans) is responsible for planning the adaptation. This can range from completely manual to autonomous. In autonomic computing, the degree of a system's autonomy is demonstrated in five levels from adaptations which are completely manual (basic level) to adaptations which are completely automatic and where the human agent only specifies high level

business rules and policies [9]. In the following, we only discuss the two extremes of this range:

- *Manual:* In the manual adaptation mechanism, the system cannot complete an adaptation without help from an external entity. In this type of adaptation, an interface is usually provided for the user to specify the required adaptations of the system. An example for manual adaptation in DSPL engineering is an approach proposed by Baresi *et al.* [11], where a variability designer, in the form of an Eclipse plug-in is used for managing the variability of the system represented by a feature model.
- *Autonomous:* In an autonomous mechanism, the system plans and performs the needed adaptations automatically. The required behavior of the system is usually provided using high-level goals or rules. It is according to these goals or rules that the system makes its adaptation planning. Therefore, the system should be context-aware, which implies that it needs to monitor its internal state as well as the context.

3.5. *Mechanism type*

This dimension defines the type of changes that an adaptation mechanism can make in the structure of a system to perform an adaptation. Gomaa *et al.* [42] classify adaptations into three classes according to the extent by which they affect the architecture of the system:

- *Code-level adaptation:* In this type of adaptation, system behavior changes although its architecture stays the same. This type of adaptation can be implemented by changing runtime entities using parametrization, inheritance, and preprocessor directives in the code [43, 44]. This type of adaptation is simpler to implement; however, it limits system adaptability since system adaptation is limited to the code of the constituents within the system. Additionally, modification and extension of the system is harder because, in most cases, adaptation logic is interwoven with system code.
- *Component adaptation:* Here, adaptation is achieved by substituting a component with another of similar interface though with a different behavior. This substitution provides a plug-in mechanism, where plug-ins are replaced with other plug-ins of the same type as the system adapts. However, some components and also the way these component are connected to each other stay the same. An example for this is systems developed with the MADAM engineering approach [19], where a number of ports are defined in the architecture which can be filled with components of the same type. In order for the system to adapt, the system middleware allows components connected to the ports to be replaced at runtime.
- *Architectural adaptation:* In this type of adaptation, the structure or the architecture of the system changes to modify the collaboration between components or the incorporation of new components in to the system.

These dimensions can be used to characterize the type of adaptation which is required for a system. In Table 2, the value of these dimensions for systems developed using the seven DSPL engineering approaches covered in this paper are presented. In the next section, we will focus on the dimensions characterizing the way adaptation is realized in a DSPL engineering approach.

4. Characterizing Adaptation Realization in DSPL

From the early efforts made in the area of systematic implementation of runtime adaptive systems, the separation between parts of the system focusing on adaptation logic and application logic has been promoted [45]. Adaptation logic is concerned with selecting the best possible variant of the running system based on the current context while the application logic provides the required functionalities of the system. This practical distinction makes the development of a runtime adaptive system easier and results in a more reliable system [3]. The separation is usually implemented as a two-layered structure. In this structure, an adaptation management layer is implemented over the application layer. The adaptation management layer accommodates adaptation logic. The application layer enables runtime reconfiguration and also provides the interfaces for monitoring and reconfiguring the system for the adaptation management layer as well as accommodating the application logic. Figure 2 shows the adaptation manager layer and the application layer.

The focus of DSPL engineering approaches is mostly on building adaptation managers for runtime adaptive systems. In order to break down the concerns for building an adaptation manager, we conceptually represent an adaptation manager as a MAPE-K loop model [9] similar to what Bencomo *et al.* [8] have proposed. In their research, they have proposed a conceptual model for DSPL adaptation management process in which the system adaptation manager can be viewed as a

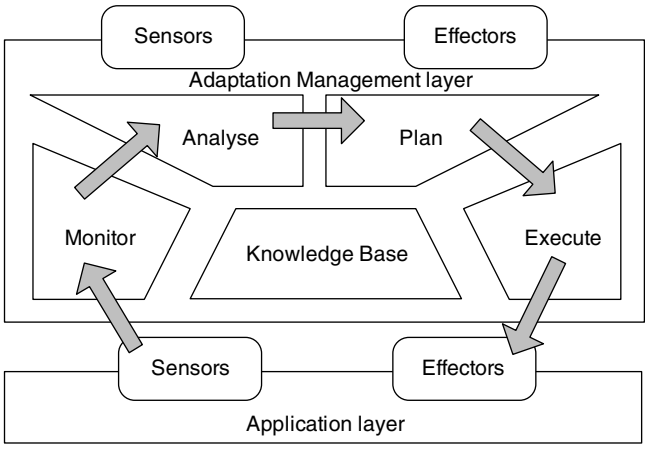

Fig. 2. MAPE-K loop [9].

MAPE-K loop. However, they do not provide the details of how this process can be mapped to the MAPE-K loop. Here, we adopt their conceptual model for adaptation management and provide detailed mapping between DSPL adaptation management process and the MAPE-K loop. The MAPE-K loop (Fig. 2) model is used in autonomic computing to represent the main concerns for building an autonomic manager.

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

We have extracted the dimensions that can be used to distinguish between existing DSPL engineering approaches. In order to design our framework adaptation realization taxonomy tree, we posit that all dimensions that characterize DSPL adaptation manager can be attributed to one of the MAPE-K steps. With this assumption, a one level taxonomy tree was defined with its first level nodes representing different steps of the MAPE-K loop (i.e. Monitoring, Analysing, Planning, Executing). Subsequently, relevant dimensions for each step are identified and those dimensions are placed as a child nodes for that step in the taxonomy tree. For a dimension related to the MAPE-K knowledge(K), the dimension was added to the step which mainly works with that knowledge.

Figure 3 shows the resulting taxonomy. In this figure, dimensions related to knowledge are shown with gray boxes while other dimensions are shown with white boxes. In the remainder of this section, these dimensions are described in detail. Each subsection describes one of the steps of the MAPE-K loop. For each dimension, the aspect of DSPL engineering approach that is characterized by that dimension is described and a number of possible values for that dimension is listed. The dashed line between dimensions means values selected for the dimension at the beginning of the arrow directly affects possible selectable values for the dimension at the end of the arrow. For each of these values, its description and supporting examples from the literature are provided.

4.1. *Monitoring and analysis*

These two steps of MAPE-K loop are responsible for making adaptation manager context-aware. This is essential where adaptation manager is expected to autonomously respond to context changes. This step captures internal or external context through a *context sensing* mechanism. The information captured through the

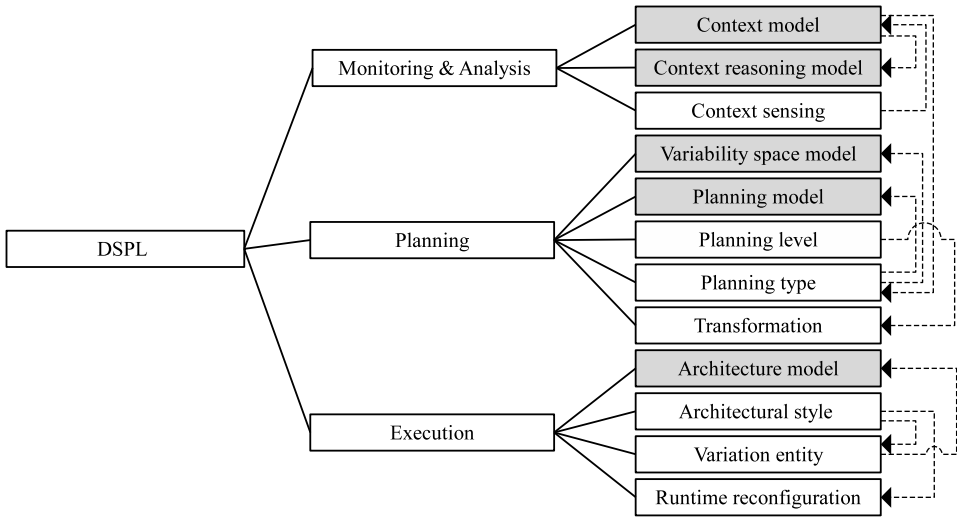

Fig. 3. The taxonomy of the proposed framework.

context sensing mechanism is represented *context model*. Since the information captured by context sensing are usually too low level to be directly usable by the planner, they are usually extended by defining more complex context conditions using *context reasoning models*. Figure 4 shows these dimensions and some of their possible values. In the following, these dimensions are reviewed:

(i) **Context model:** A context model is used to represent the knowledge acquired by the monitoring and analysis steps. This is a minimal representation of the system context that is important for planning. Two types of common context models are presented in the following:

- *Property-set:* In this type of context model, the context of the system is represented by a set of properties crucial for planning. The values of these properties are updated at runtime through monitoring and analysis which may lead to the adaptation of the system. For example in the MUSIC framework

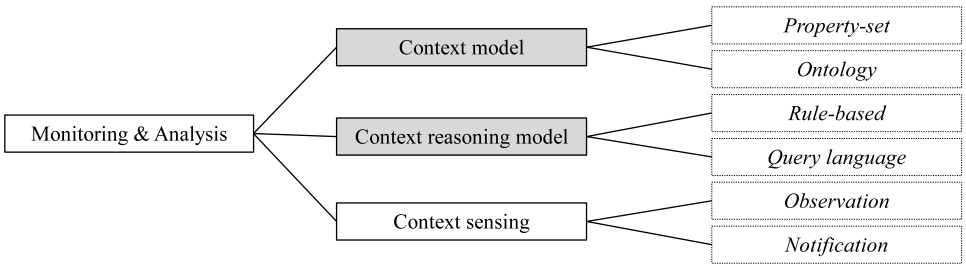

Fig. 4. The taxonomy for dimension related monitoring step.

[46, 47], the information about the environment (e.g. ambient noise) as well as information about internal state of the system (e.g. device memory) and QoS properties (e.g. network quality) are kept and updated as values to a number of context variables. The planner accesses these context variables to decide about system adaptation.

- *Ontology.* In this type of context model, ontology representation languages such as OWL [48] are adopted to relate (together) the important properties of the system using semantic relationships in the ontology. The use of an ontology allows a more structured representation of the context. It also enables reasoning over the context model using existing semantics reasoners and sharing of the context model between different systems. For example in [49], the OWL language is used for expressing the context information in a way that allows for reasoning. In this framework, the properties are organized using classes and their relations are expressed using semantic relations. In another case study [50], the OWL language is used to represent the context model of a smart hotel. As an another example in [51], the global context which is represented with an ontology is mapped to the feature model which represents the variability of the system. Using this mapping, a local context of each variant of system is found based on the features in that configuration. This will exclude irrelevant information from reasoning in each specific context which results in more efficient adaptation.

(ii) **Context reasoning:** Context reasoning extend information acquired using context sensing by producing more contextual information using existing ones. A context reasoner accepts one or more existing context information and produces new more abstract information. For example, it may take information from ambient noise and motion sensors and specify the value for the home occupancy information. Use of context reasoning allows providing more abstract information for the planner which facilitates the design of reasoning in a planner. Additionally, context reasoning can be used for alleviating the effects of noise and uncertainty by working with multiple sources of information.

- *Rule-based logic:* In this type of reasoning, the new context conditions is defined by creating simple rules over existing context information (e.g. propositional logic). For example in Genie [13], conditions which result in transition from one state to another state is defined using propositional logic over existing information. Use of rule-based logic provides a simple and performance efficient way of finding new information over existing ones. However, in the cases, where temporal characteristics as well as quality of information (e.g. certainty) should be considered in context reasoning, simple rule-based methods may not be efficient.
- *Query languages:* The captured information by context sensing usually provides a stream of observed values (with different timestamps, confidence, and etc). Therefore, query languages and their corresponding reasoners are used in

some context managers. These query languages facilitate expression of more complex context condition. For example, the context manager in DiVA project provides a query language named EQL which is a language similar to SQL which works on a sliding window of sensor events. Similarly, MoRE uses an ontology as its context model and SPARQL is used for context reasoning.

(iii) **Context sensing:** In order to be context-aware, a DSPL engineering approach should provide a way for defining context sensing mechasnism which enables capturing situational information (e.g. internal state, environmental context). This separates activities for capturing information from other adaptation activities and prevents other entities in adaptation management from becoming involved in the internal process for obtaining these information. Context sensing mechanisms are usually implemented by enforcing a standard for designing sensors (e.g. defining standard interfaces for sensor components) which provide a unified way for capturing contextual information. This enables the use of a central middleware for management of contextual information. Additionally, the middleware provides functionalities such as organizing, storing, or querying context information. For example, in [27], the sensors are realized as components named *context nodes* which are organized in a hierarchy. Although these components may have different properties such as active or passive, they all provide same interface which allow them to be accessed by the context manager in a similar way. Context sensing middleware also allow reuse of general sensors such as CPU load sensors. For example, context manager library for the DiVA project provides a library of sensors which can be reused in different projects. This context sensing is usually provided in two ways: observation and notification.

- *Observation:* The context sensing mechanism which enables observation allows the planner to query contextual information. Therefore, the planner is responsible for getting new context changes (e.g. in a specific interval) in order to get new context information. This is a simple type of monitoring but it may cause overhead as it is needed to query new information regularly in order to have model of the context.
- *Notification:* Context sensing mechanisms which enable notification allow the planner to be notified of changes in certain contextual information as the they take place. Therefore, the planner gets notified as soon as an interesting situation takes place and reduces the overhead. However, it requires interfaces in the middleware to be called when those changes happen.

4.2. *Planning*

In a typical software product line process, the application engineers develop a suitable variant of the product for a specific user by selecting and instantiating assets developed during domain engineering with the help of variability models. In the context of DSPL engineering, variant selection should be performed at runtime, usually without

any external help. Therefore, the system should take the role of the application engineer to select and instantiate new assets to generate a new variant of the system. We refer to this process as planning. Most efforts in the DSPL community have been in the area of developing planning approaches [8]. A planner uses situational information provided in the context model to decide if an adaptation is necessary, and if so, to find the desired variant for the current situation. The possible variants of the system are represented in a *variability space model*. The planner uses a *planning model* to select the next variants of the system from the variability space model. The planning model represents how the system selects the desired variant according to the current context and variant of the system. Variability space models may define possible variants of the system in terms of their features or architectural structure. Since the planner works by selecting variants from the variability space model, the level used for variability space model specifies the *planning level*. Planning level specifies the aspect of the system on which planners reason (i.e. feature or architecture). If feature-level planning is used, the desired variant of the system will be represented by its features which needs to be transformed to architectural structure of the system using a *transformation* mechanism. This architectural structure is then used in the execution step to reconfigure the running system to the selected target system. Figure 5 shows these dimensions and some of their possible values which will be reviewed below.

(i) **Variability space model:** The variability space model represents the alternative variants of the system and their difference points. The model can represent variability at different abstraction levels. For example, the variability

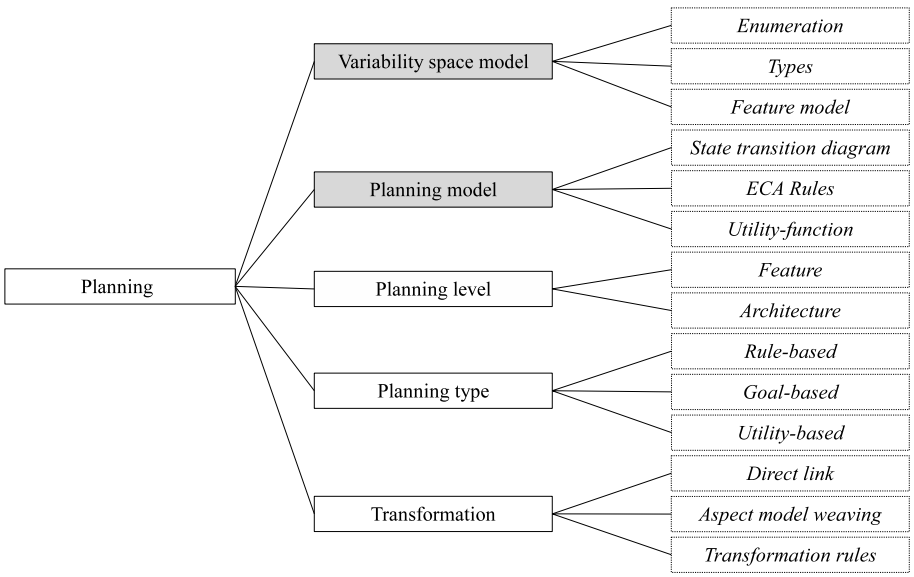

Fig. 5. The taxonomy for dimension related planning step.

of a system can be represented by what are the properties of different contexts the system can operate in [51, 52] or what components are different between the variants of the system.

- *Enumeration:* In this type of variability modeling, the possible variants of the system and their properties are enumerated as a set of possible variants [53]. The planner selects the most suitable variant of the system for the current context from the enumerated set and adapts to it. Those systems that model their variability using this type of modeling are generally simpler to design and validate. However, it is likely that the size of the possible variants of the system increases rapidly and becomes unmanageable as the system grows [23]. This type of variability modeling is usable when the possible set of variants of the system is possible to define. For example, in a case study of using DSPL [15], the enumeration model is used to model three possible variants of a network discovery agent. These variants showed what kinds of roles the agent can have in the service discovery process at runtime.
- *Variation points:* In this type of variability space modeling, difference points in a base system are specified and called variation points. A type is defined for each variation point within the system, where different alternatives of the same type can be selected for each variation point. The adaptation in the system takes place by the replacement of alternatives with the same type in the variation points of the system. The current variant of the system can be specified by knowing the alternatives that have been selected for the variation points. An example for this type of variation management is the MADAM approach [19], where ports are defined in the architecture of the system. Ports are the locations where different components with the same interface can be substituted with one another. Variation points provide a simple and manageable solution for managing variability in a system [7]. However, one problem facing this type of variability modeling is that it does not support cross-cutting variations, where the selection of alternatives in one variation point puts constraints such as exclusion or inclusion on other variation points [7].
- *Feature modeling:* Feature models [54, 55] are widely used in SPL engineering for representing the variability of the product family [17, 23, 27, 56, 57]. A feature model provides a hierarchical structure that represents the organization of the features which are 'increments in product functionality' [58]. When decomposing a feature into sub-features, the sub-features may be optional or mandatory, or may form Xor-, Or-, or And-groups. These relations express the possible variability of the system. Feature models also represent cross-cutting variations using integrity constraints. Feature models are commonly used as a variability space model in DSPL engineering approaches because of their power to represent complex variability of a system.
- *Context variability model:* In this type of variability space modeling, a context variability model which represents the dominant contextual drivers for

variation (e.g. different locations) is defined beside the feature model. This model is linked to feature model in a way that selection of different dimensions of it constrains the feature model variability space [59–62]. This creates an orthogonal link between context and features which facilitates relating different context conditions to an appropriate feature configuration by eliminating irrelevant features during feature model configuration. For example in [59] another feature model is used as context variability model. This context variability model is linked to the main feature model using feature model integrity constraints.

(ii) **Planning model:** The adaptation planner of a system is usually implemented using a reasoner which considers the variability model, the context, and the current variant of the system as inputs. It then develops the most suitable variant for the current context according to its planning knowledge [1]. The planning knowledge could be hard-coded into the reasoner or expressed explicitly by a model which is available at runtime and used by the reasoner. Using models for representing the planning knowledge eliminates the need for the development of a new reasoner from scratch for every system developed with a DSPL engineering approach. A planning model specifies the policy of the system for selecting the suitable variant of the system by having access to the current variant of the system and the current context at runtime [6]. The type of planning model is determined by the planning method that is used in the system. Here, we cover the planning models used by the reviewed systems in this study.

- *State transition diagrams:* In these models, which can only be adopted when enumeration is used for representing the variation space model, the adaptation policy of the system is represented using a state transition diagram. The states are variants of the system and the transitions are the possible adaptations. The guard condition of a transition defines the situation in which that specific adaptation could take place. One problem associated with the state transition diagram is that they are not easy to maintain after they are initially designed because of their relatively complex structure [23]. An example of using this approach to define the adaptation policy is a case study of the Genie framework [12], where a state transition diagram is used to specify when a flood monitoring agent should change its behavior according to its context and what target variant it should adapt to.
- *ECA rules:* Event-Condition-Action rules represent what actions the system should perform in response to an event when certain conditions hold [23]. Events could be changes in the context (e.g. network traffic) or system internal state (e.g. processor usage); Conditions are defined over the current environment or the current variant of the system; and finally Actions can be a set of activations and deactivations over the system components or similar changes in system features. Expressed using different languages and models, the ECA rules usually have the advantage of being readable [63]. The

planning model of the system can also be modified easily by adding and removing ECA rules. However, a problem with ECA rules is that when the number of rules are large, conflicts are likely to arise between the rules. Detecting such conflicts are difficult [63]. In such a case, it is ambiguous how the system should act, and a conflict resolution mechanism is essential, for example, giving priorities to some actions. An example for using ECA rules to define adaptation policy is a case study where the ECA rules are used in a dynamic access control product line to specify the changes in the available functionality of the system as users with different roles use the system [64]. In this system, the ECA rules specify access to what functionality should be granted to a user according to his role.

- *Utility functions:* In this type of planning modeling, the planning model consists of a utility function which is usually a heuristic representation of the desirability of adapting to a variant of the system or performing an action in a specific context [63]. This utility function is used to select the most desirable variant of the system to adapt or determine the action that should be performed in each context of the system. An example for this is the MADAM approach [19], where a utility function represents the desirability of each system variant which is a weighted sum of the important properties of the system. In another example [65], features in the feature model are annotated with quality attribute and value for a quality attribute of a configuration is calculated using aggregated functions. The desirability of a configuration is defined by a single value which equals the weighted sum of all quality attributes for that configuration. Using these information, an optimization method is used to find a configuration between possible configurations of the partially configured feature model which maximizes this value. One problem associated with the utility function is the difficulty usually encountered in the design of such function that would precisely model the desirability of the system [7].

(iii) **Planning level:** The planning level dimensions specifies if planning is performed over the features of the system or the architectural entities of the system. The planner may see the system as a number of components collaborating with each other and therefore decides which components should be active, their configuration and how they should collaborate (e.g. [7, 12]). Similarly, the planner can view a system according to its feature, therefore deciding what features should be active in the system according to its context (e.g. [16, 23]). The selection of appropriate planning level removes unnecessary details while preserving the required information for planning.

- *Feature:* Similar to feature model configuration in SPLE, the planner could decide at the feature level which would abstract away realization details. Planning on the features promotes the separation of concerns by dissociating the context requirements from realization details [66]. However, using planning on features necessitates the implementation of a transformation mechanism

which converts the feature level configuration to the architectural configuration [23]. For example, in a robotic case study [67], the adaptation planner for the robot software selects required features according to the context. The feature model configuration is used by a weaver to compose related architectural models to create architectural configuration model of the system.

- *Architecture:* Here, planning is performed on architectural entities (such as components). Planning on architectural entities is usually harder than planning on features, because reasoning becomes more complicated when context requirements and implementation constraints are taken into consideration at the same time. For example, in [19], the requirements of the system are defined on the properties of the components and the planner tries to find the components which satisfy these requirements best.

(iv) **Planning type:** Planning type represents the kind of planning method are employed for selecting the most suitable variant of the system. Selecting the most suitable variant of the system is usually expressed informally in SPLE [6], but it should be specified explicitly and formally in DSPL in order to automate the selection process. Generally, planners can be categorized into three groups [68]:

- *Rule-based:* In rule-based planning, adaptation takes place by following a set of rules which specify what actions should be performed in each particular context [8]. In this planning type, the planning strategy is usually modeled using ECA rules or state-transition diagrams and the strategies are then enforced by a reasoner. In this type of planning, adaptation rules are usually defined at design time which requires thorough knowledge of the operating environment at design time [8]. However, some rule-based planners allow the modification of the rules at runtime [23]. In a case study of MoRE engine used in the context of Smarthomes [16], adaptation rules are organized as a set of condition/resolution rules. The conditions are specified using propositional logic over context variables. The planner is made of a reasoner which checks if a condition holds for any of the rules and performs the resolution for those rules whose condition holds.
- *Goal-based:* In goal-based planning, the high level goal of the adaptation is defined, and finding how to achieve the goal is left to be figured out by the planner. In goal-based planning, unlike rule-based planning where the actions of the system in different contexts are specified at design time, the most suitable action to be performed is figured out at runtime by using a planning method. Usually in this planning type, the effects of different actions on the adaptation goals are formally defined. Subsequently, the problem of finding the most suitable action is reduced to a satisfiability problem (SAT) [51, 69] or a constraint satisfaction problem (CSP) [70–72] which is then solved using a SAT-solver or a CSP-solver. For example, in Parra *et al.* [70], a CSP solver is used at runtime to find the most suitable variant of the system which

satisfies the goals of adaptation. The goals of adaptation are expressed through constraints over system quality of service properties.

- *Utility-based:* Goal-based planners categorize the current configuration of the system into two possible states: desirable when the goals of the adaptation are satisfied and undesirable when the goals are not satisfied. However, it is not always possible to satisfy all goals of the adaptation at the same time or exactly specify the goals of adaptation. Therefore, the goal of planning can be changed to keeping the system at the most desirable state. In such cases, a utility which is a quantitative value representing the desirability of the current system variant in the current context is defined for the system [73]. This utility is approximated by a utility function over the properties of the system and the system context. Similarly, the best action is usually chosen using a heuristic which approximates the effect of each action on the utility of the system. For example, in [46], the planner of an adaptive travel assistant mobile application uses a utility function to approximate the fulfillment of user preferences in different situations. This prediction function is used to find the configuration of the system which has the highest predicted utility. In another example [74], the problem of feature configuration is represented as multiobjective optimization with various criteria and then different multiobjective evolutionary algorithms are used in order to find the best configuration for that feature model.

(v) **Transformation:** In the DSPL adaptation managers which perform planning on the feature level, the planner specifies the target configuration of the system in terms of features other than architectural configuration which necessitates a linking strategy between feature and architectural configuration of the system. This mapping is usually addressed by a transformation mechanism which can create target architectural configuration of the system based on the selected features. A similar situation arises in the SPLE product configuration, where target product configuration is specified in terms of features. There are some work in SPLE which focus on relating the variability space model of the system to system code [75, 76] or its architectural model [77–80], or its domain specific modeling language in the design of a planner [81]. DSPL engineering approaches have been inspired by such SPLE approaches in their development of a transformation mechanism between feature configuration and the architectural configuration at runtime. Here are three approaches used for this purpose:

- *Direct link:* One simple way of linking models at two levels is to create a direct mapping between the features in the high level model to the architectural model in system implementation [82]. Consequently, the architecture model of the system after adaptation can be built by selecting the fragments of the architecture model that are mapped to the selected features of the system. However, this type of linking is not practical in certain cases, because direct mapping between features and the architectural model fragments would not

exist [82]. In these cases, the linking between the feature model and the architecture model can only be enabled if the feature model is built considering the architecture. However building the feature model by considering the architecture does not provide proper separation of concerns. An example of using direct link for relating these two models is the MoRE approach [16], where the link between features and architecture model of the system is defined by a superimposition operator which takes features as input and returns the components and connectors which should be active.

- *Aspect model weaving:* Model-Driven Development (MDD) techniques allow for the automated generation of detailed architecture models from high level designs. Specially for creating a detailed architecture model from the configured feature model, aspect model weaving can be used as presented in SMARTADAPTERS [83] and Kompose [84]. In these approaches, system architectural commonalities are represented in a base model. To create the final model of the system, each feature is mapped to a set of aspect models which are then woven into the base model if that feature is selected in the feature model configuration of the system. For example in [85], this approach was used to build an adaptive home automation system for dependent people. In their system, the authors defined a *pointcut model* representing, where an aspect could be woven into the system and an *advice model* which represents what can be woven at joint points. When the system adapts, the advices are woven into the pointcut model according to the *weaving directives*.
- *Transformation rules:* In this linking strategy, in addition to the feature model of the system, an alternative feature model is defined over the variability of the architecture model such that all possible variants of the architecture model could be represented by that feature model configurations. Subsequently, this feature model is linked to the main feature model of the system using a set of transformation rules. The configuration for the feature model over the architectural model can be found using these transformation rules whenever a feature model configuration is available. For example, in [86], this approach was used to link between features of the system and the architecture model in an adaptive video surveillance system. The system was modeled using two feature models one on the features and another on the components. These two models were linked using a set of rules represented using propositional logic. When the system adapts, the planner finds a configuration for the feature model over components satisfying the rules using a SAT solver.

4.3. *Execution*

In traditional software product lines, the target system is developed and deployed after its configuration has been specified. But adaptation manager in DSPL should reconfigure a running instance such that current variant gives its place to the target variant while ensuring system consistency. The execution step is responsible for this

part of system adaptation. The planner provides the *architecture model* of the target system to the execution steps. The architecture model specifies the configuration of architectural entities (e.g. components, services, connectors) and how they are put together. The execution step uses a *runtime reconfiguration* mechanism to safely introduce, remove, or modify *variation entities*. The variation entities are those architectural entities by which adaptation is made possible through adding, removing, or modifying them. The selection of architecture model, variation entity, and runtime reconfiguration mechanism is affected by the *architectural style* which has been selected for the system. Figure 5 shows these dimensions and some of their possible values which we review in the remainder of this sub-section.

(i) **Architecture model:** When an adaptation occurs, the planner should provide the configuration of the target variant of the system to be used for the actual reconfiguration. This model is an abstract view of the system with which all possible variability of the system can be represented. In other words, it is detailed enough so that every possible adaptation which the planner can perform can be reflected in it. This model is used by the reconfiguration mechanism to find points in the system which should change as a result of adaptation.

- *Custom languages:* Since this model of the system is available at runtime, it should be abstract to enhance memory and processing efficiency. Therefore, approach-specific models are defined in some cases [12, 16]. For example, in MoRe [16], an architecture model named PervML [32] has been proposed and adopted, where only the component and their connections are represented as a graph. Some other approaches [24, 27] use the architecture model which is used by the adopted reconfiguration middleware to define the system

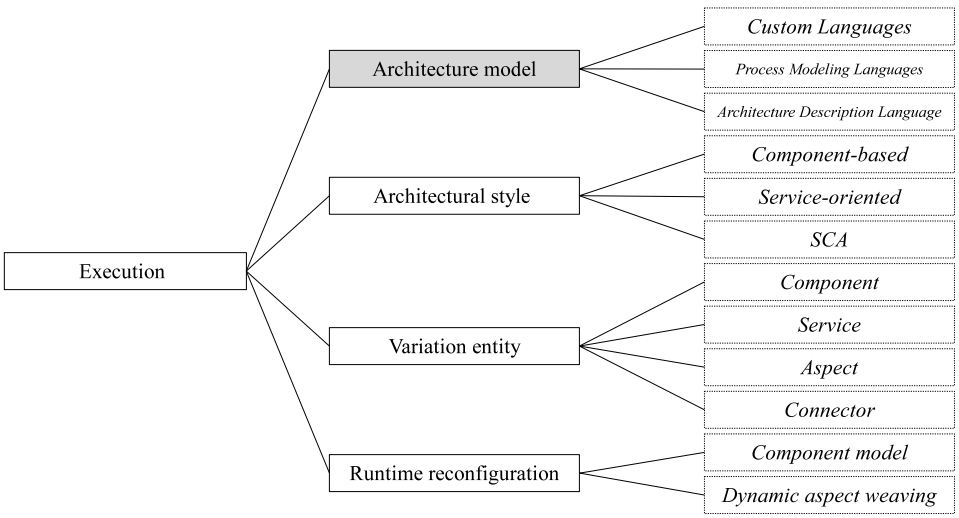

Fig. 6. The taxonomy for dimension related execution step.

architecture. Fore example, in [12], OpenCOM DSL which is used by the OpenCOM middleware to specify system architecture is used.

- *Process modeling languages:* Business process modeling languages are widely used for defining how services interact in order to fulfill the requested functionality of the system. This models is usually abstract enough for representing structure of the system for planning and used in some approaches as architecture model of system. For example, the system adapts dynamically by changing the process that defines the service interaction in [11], which is expressed using the BPEL.
- *Architecture description languages* (ADL) this type of formalism is used for conceptually specifying architecture of the system. This model is used by system architect to design the system in the high level and therefore can be used for representing system architecture by a planner. For example, in [87], an ADL is proposed for dynamic service-oriented product lines which allows representing variability of architecture as well as structural elements and configuration of it.

(ii) **Architectural style:** Architectural style defines the highly granular entities of the system and how they are connected to each other. Although, selection of architecture style is not directly related to adaptation, the selected architecture style affects the selection of architecture model, variation entity, and runtime reconfiguration mechanism. Therefore, we have considered it as one of the dimensions. In the following, we discuss the suitability of two of the most common architectural styles for use in DSPL engineering approaches:

- *Component-based:* In component-based architecture, components communicate through connectors to provide the required functionalities of the system. Although components are built to work independently from their context, they are not usually designed to be reconfigurable at runtime. Thus, architectural pattern have been proposed to facilitate runtime variability [1, 88]. Gomma *et al.* [22] propose a process for developing a component-based DSPL, where the architecture is built using reusable architectural patterns (such as master/slave, client/server), and for each of these architectural patterns, a specific reconfiguration pattern is defined. Modeled with state and scenario-based models, these reconfiguration patterns provide a safe mechanism for reconfiguration.
- *Service-oriented:* Both service-oriented architectures and software product lines share the common goal of reusing existing assets for developing new products. A number of approaches have already been proposed for developing service-oriented product lines [89–92]. Service-oriented architecture has also been used for developing dynamic reconfigurable systems because, it eases the dynamic composition of self-contained and loosely-coupled services [2, 46, 93]. Considering the motivations for using service-orientation both for software product lines and dynamic reconfigurable systems, there have been a number of works that developed service-oriented DSPLs [11, 16, 27, 42, 94–96].

Cetina *et al.* [16] propose a service-oriented DSPL engineering approach for pervasive systems (e.g. Smarthomes). Pervasive systems are highly dynamic in nature since new entities are frequently introduced to the system or removed. In their approach, Cetina *et al.* define two categories of components: devices and services which implement system functionality. These components are connected using communication channels. The system adapts by activating/deactivating both services and communication channels.

- *Service component architecture:* This hybrid architecture tries to combine components-based and service-oriented architecture in order to take the benefits of both of them. In this architecture, the functionalities are provided by components which their interfaces are defined as services. Therefore components interact by calling each other's services. This architecture takes advantage of service-oriented systems benefits such as well-defined communication interfaces, simpler integration, and dynamism in context of component-based environment. For example [27] adopts this approach in order to facilitate dynamic binding and unbinding of components at runtime.

(iii) **Variation entity:** Variation entities are parts of the system that change when the adaptation is carried out. The type and the granularity of the variation entities affect the type of adaptation which is supported by the system.

- *Components:* According to the definition of a component by Szyperski [97], components can be deployed independently, and their context dependencies are always specified explicitly. This disconnection of the component from its environment allows the component to be used as the variation entity in the adaptation. However, it has been argued that components do not have the same granularity as features most of the time [69] and they may be more coarse or fine grained. In order to support feature realization with a higher granularity than that seen in components, reconfiguration can be defined over a component collection. For example, in the Genie approach, a collection of components which address a specific area of concern is called a *component framework*. Component frameworks act as the variation entity in adaptations. One concern with using components as variation entities of the system is that certain features do not have corresponding components at the component level. For example, the implementation of an authorization feature may be distributed between multiple components. Another concern with using components as variation entities is in maintaining states of the components as they are replaced with other components while the system adapts.
- *Aspects:* Aspect-oriented programming and dynamic aspect weaving can be used in order to enable runtime reconfiguration in a DSPL adaptation platform. However, efforts for using aspect-oriented programming in both software product lines and DSPL have shown that this approach may lead to certain problems [11, 98]. Features are more abstract than aspects, and a feature may be represented by many aspects; therefore, handling the linking between the features of

the system and the aspects may become unmanageable. In an example of using aspects for realizing features [99], each feature is implemented by a set of aspects. These aspects are implemented following a set of design patterns in order to enable both static and dynamic binding of the features in a system.

- *Services:* Services can be simply deployed into a system and easily composed with each other. They also have characteristics such as being loosely coupled, self-describing and having well-defined interfaces which make them viable candidates for runtime adaptive systems. For example, Rouvoy *et al.* [46] used services as the variation entities for ubiquitous environments, where new services constantly enter and leave the system. Additionally, alternative third-party services can be discovered and added to the system so they enable open adaptation, where the system can evolve into new variants which have not been considered at design time. However, a problem of using services as reconfiguration entities appears when we attempt to define the links between services and system features. Similar to components and aspects, services are not the most accurate representatives of the features in a system.
- *Connectors:* Connectors define how the constituent elements of a system are connected together. The connectors in a system could be glue codes, communication channels, or workflows. The connectors usually define the flow of the data in the system. Therefore, they are selected as the variation entity when the flow of data defines the behavior of the system. Examples of this function for the connectors may be encountered in data processing systems (e.g. video filters [65, 86]) and business processes [88, 100]. An example of using connectors at runtime adaptation is the MoRe framework [22], where connectors activate/deactivate in order to introduce new services and devices in a Smarthome application.

(iv) **Runtime reconfiguration:** In order to execute the actual adaptation actions, the execution platform should provide the ability to modify the architectural configuration of the running system (such as the binding and unbinding of the components) and observe the running system's architectural configuration. These functionalities are usually provided by an application-independent middleware which frees the system from dealing with actual adaptation concerns by taking responsibility for performing the adaptation while providing interfaces for managing that. These middleware usually work by implementing a reflection mechanism. Reflection in the middleware [101] is the ability to provide introspection and intercession. A reflective middleware provides a model of the system which allows for any system change to be reflected on the model and likewise for any change on the model to be reflected on the system. Therefore, the planner can access the representation of the architectural configuration of the system using this model (introspection). It can then reconfigure the system by modifying this model (intercession). In this dimension, two of the mechanisms used by middleware that enable runtime reconfiguration in the system is reviewed.

- *Component model:* A component model is a definition of the semantics and syntax of components which show what they do, how they are structured and how they are composed [102]. A system can reconfigure at runtime if the architecture and its component conform to a component model that allows runtime reconfiguration. For example, OpenCOM middleware [31] uses a component model for enabling runtime reconfiguration in the system. This middleware is used in the Genie tool for developing DSPLs. OpenCOM is a reflective middleware which supports hierarchical implementation of the system using component frameworks. Component frameworks are sets of components addressing a specific feature of the system. The Fractal component model [103] is another hierarchical component model that can be extended and customized. The OSGi framework [104] is yet another component model which works on services and supports runtime installation, starting, restarting and uninstallation of services.
- *Dynamic aspect weaving:* Dynamic aspect weaving can be utilized to modify the aspects of the system at runtime. This ability can be used to adapt the system when the variation entities of the system is aspects [105]. For example, in [106], features are implemented using dynamic aspects in AspectJ [107]. These dynamic aspects are added or removed at runtime using dynamic aspect weaving. In another work, Baresi *et al.* [11] used dynamic aspect weaving in a business process to add/remove features at runtime, which were represented with aspects of BPEL code.

4.4. *Comparing DSPL engineering approaches using the proposed framework*

Table 3 shows the value of each dimension in our framework for each of the approaches that have been reviewed. In this table, columns show the MAPE-K loop steps and their respective model and mechanism dimensions and rows indicate the reviewed DSPL engineering approaches.

In the following, we use Table 3 to compare each of the selected approaches:

Baresi *et al.* In this work adaptation planning is performed by a human expert through a user interface. As a result, monitoring and planning steps are not applicable to this approach. In addition, considering BPEL code as the variation entity is a distinctive feature. This has necessitated the use of dynamic aspect weaving as the runtime reconfiguration mechanism. However, the adoption of BPEL as variation entity restricts this engineering approach's application domain to systems which can be designed and implemented through business processes.

Bencomo *et al.* (Genie): The notable design decision in this approach is the use of two variability space models. Enumeration is used for defining the context variability whereas variation points are used for determining the architectural variability of the system. These two variability models are linked directly using the OVM. Furthermore, planning is different from other approaches as it is defined by a state transition

Table 3. Features of different DSPL engineering approaches.

	Monitoring & Analysis			Planning					Execution			
Approach	Context model	Context reasoning model	Context sensing	Variability space model	Planning model	Planning level	Planning type	Transformation	Architecture model	Architecture style	Variation entities	Middle-ware
Baresi *et al.*	N.A.	N.A.	N.A.	Feature model	N.A.	Feature-level	N.A.	Direct Link	BPEL	Service-oriented	Aspect	Dynamic aspect weaving
Bencomo *et al.* (Genie)	Property-set	Rule-based	N.S.	Enumeration/ Variation points	Transition diagram	Feature-level	Rule-based	Direct Link	OpenCOM DSL	Component-based	Subsystem	OpenCOM
Cetina *et al.* (MoRE)	OWL	Query Language	N.A.	Feature model	ECA rules	Feature-level	Rule-based	Direct Link	PervML	Service-oriented	Service/ Connector	OSGi framework
Floch *et al.* (MADAM)	Property-set	N.S.	MADAM context manager	Variation points	Utility function	Architecture	Utility-based	N.A.	N.S.	Component-based	Subsystem	Configurable Product-bases
Gomaa *et al.* (REPFLC)	N.S.	N.S.	N.S.	N.A.	ECA Rules	Architecture	Rule-based	N.A.	N.A.	Component-based	Component	N.S.
Morin *et al.* (DiVA)	Property-set	Query Language	WildCAT	Feature model	N.S.	Feature-level	N.S.	Aspect model weaving	Fractal/ OpenCOM DSL	SCA/ Component-based	Component	Fractal/ OpenCOM
Parra *et al.* (CAPucine)	Property-set	Rule-based	COSMOS	Feature model	ECA rules	Feature-level	Rule-based	Aspect model weaving	Fractal	SCA	Component	Fractal

Note: N.A.: Not applicable, N.S.: Not specified Knowledge.

diagram. These two features create an intiutive and simple adaptation management mechanism. As a drawback, however, the management of context variability and its link with the architectural variability becomes more difficult as the system grows.

Cetina *et al.* (MoRE): In this approach, OWL is used to develop the context model of the system. The approach enables complex context reasoning using the SPARQL language. Another notable dimension of design in this system is the use of connectors as variation entities. This is specially useful in the context of Smarthomes, where new devices are introduced at runtime.

Floch *et al.* (MADAM): The use of utility-based planning is a distinctive feature for this approach. The planning model for this approach is a utility function which is defined over the architectural configuration of the system. The application of the utility-based approach liberates the adaptation manager designer from the task of explicitly defining adaptation actions such as ECA rules and transition diagrams. However, defining a precise utility functions is rather complex.

Gomaa *et al.* (REPFLC): The distinctive feature of this approach is that it uses ECA rule planning such that the rule actions work directly on the architecture of the system. These rules are part of reconfiguration patterns which are defined in the domain engineering process. As these reconfiguration patterns are directly applied to the architecture itself, defining a transformation mechanism or architecture model becomes unnecessary. However, the design of these reconfiguration patterns is an arduous task since they directly impact the system's architecture.

Morin *et al.* (DiVA): The approach suggests a general architecture with flexibility for planing and execution. This allows the selection of appropriate planning and execution mechanisms based on the condition of the application domain. Additionally, this model uses the WildCAT context-sensing middleware which provides EQL, a language capable of querying the values for context reasoning.

Parra *et al.* (CAPucine): This approach uses the COSMOS framework in order to enable context-awareness. Additionally, as its important feature, this approach uses aspect model weaving to create the architecture model of the target system. This allows the system developer to define more complex relationships between the feature model and the architecture model of the system. However, the same also necessitates the design of rather more complicated (as a result, harder-to-maintain) aspect models. One other notable design decision, in this approach, is the selection of SCA as the architectural style, which takes advantage of several benefits of both service-oriented and component-based architectural style.

5. Guidelines for Selection of a DSPL Approach

Our proposed framework can be used for aiding one in selecting or designing a DSPL engineering approach. In order to adopt or develop a DSPL, one can select appropriate values for different dimensions of the proposed framework according to the specific needs of the problem domain. The selection of possible values for this framework is mostly reliant on the type of adaptation sought. This can be called expected

DSPL *specification*. Then, the DSPL specification can be compared with the specification of existing approaches to find exact or partial matches, where models and mechanisms can be borrowed in order to build a new DSPL engineering approach.

In the following, we provide general guidelines for the selection of appropriate values for different dimensions of the framework organized by the steps of the MAPE-K loop. Additionally, we discuss how selecting different values for the dimensions affect adaptation properties of the final product whenever possible.

- **Monitoring and analysis design**
  - *Context model:* Design of a context model is mainly determined by properties of the context and aspects of those properties that are significant for the planner. These aspects include context properties history, quality, and interrelation, among others. Using property-sets for representing context provides simple yet efficient solution which can be simply extended to represent those aspects. However, using ontologies provides a new set of capabilities for the context model which cannot be easily achieved using property-sets. For example, it allows semantic reasoning on the context model and enables possible sharing of context information between different systems which can be useful in the case of decentralized adaptation when multiple agents work together to plan an adaptation.
  - *Context reasoning model:* Since the reasoning model is used to analyze the information represented by the context model, the selection of a reasoning model is essentially affected by the context model. For example, using ontologies for representing the context model calls for using semantic reasoning and therefore semantic reasoning models. However, the complexity of a context reasoning mechanism and as a result its reasoning model depends on the analysis required for the operation of the planner. This could range from a simple rule-based mechanism, where reasoning model would be simple rules such as thresholds, to complex machine learning mechanisms where a large number of context properties are considered at the same time to speculate about the context situation.
  - *Context sensing:* Generally, the design or adoption of a context sensing mechanism is mostly reliant on the sources of the changes resulting in an adaptation (e.g. context, system, or user), its type (e.g. functional, non-functional or technological) and the system platform (e.g. software or hardware) [37]. We refer the interested reader to guidelines and taxonomies currently existing in the literature [93, 108, 109].
- **Planner design**
  - *Variability space model:* Variability space model can be used to represent a system variability in terms of its context, its features, or its architecture. Two or more variability space models can be used in a system to show its variability from two aspects at the same time. Different variability space models may suit different aspects of the system. For example, in [13], enumeration is used to represent context variability while variation points are used for architecture

variability. One other notable factor affecting the selection of a variability space model is the complexity of variability in the system. For example, although enumeration works effectively when the number of system variants is limited, it is not efficient in systems with higher number of variants.

- *Planning model:* Since the type of planning model is mainly specified by the planning type used in a DSPL engineering approach, we have incorporated discussion about it in the planning type part.
- *Planning level:* Deciding on the features of the system separates planning concerns from implementation requirements. Such separations will result in a more elegant and readily understandable adaptation logic, which allows easier extension as it is less likely to be faulty. Furthermore, planning at the feature level is closer to the procedures in an SPL engineering process and has the benefit of easier adoption when an SPL engineering approach is extended to DSPL. However realizing this separation will require a mechanism to link high-level features to the architecture level details. Developing a mechanism that creates this link is not generally effortless and will result in overhead at runtime.
- *Planning type:* If planning is not autonomous and the variants of the system are selected by an external entity such as the user, planning does not necessarily need to be implemented in the system. Otherwise, the type of planning for the system depends largely on the type of the goal that is pursued by the adaptation. In cases, when the system adapts in a limited way and this adaptation can be specified at design time, rule-based approaches seem to be the better option for adoption. In circumstances, when the adaptation of the system may not be specified at design time and can only be figured out at runtime according to specific situations, goal-based and utility-based approaches are the more appropriate choice. Goal-based approaches are suitable when the goal of the adaptation is to enable the system to self-configure and self-heal, and it can be specified formally. Utility-based approaches seem to be the better fit for a self-optimizing context when the desirability of the system can be specified with a quantitative value.

  Another important aspect in selecting the planning approach is the impact of the approach on the regular operations of the system. Factors such as criticality, overhead, reliability and resilience of adaptation should be taken into account when a planning approach is selected. For instance, rule-based approaches usually have less overhead on the system which should be considered in the running environments when system resources are limited. Given their easy verification at design time, rule-based approaches are generally more reliable. However, it should be noted that verifiers do exist that can be used to validate adaptation at runtime using methods such as simulation or model checking [23, 110].
- *Transformation:* The direct link between a system's feature model and its architectural model is simple to understand and establish. Since the features and architecture part of the system have different natures, this mapping does not always exist. However, the limitation can be bypassed if the feature model is built by considering system architecture. In more complicated situations, more

complex approaches such as transformation rules and aspect model weaving should be used. It is sometimes necessary to run a SAT solver at runtime when transformation rules are used. Running the SAT solver at runtime increases the adaptation overhead. Furthermore, developing and maintaining consistent aspect models which are used by aspect model weaving is itself an arduous task considering that aspect models are hard to understand. One other difference which should be considered when selecting a mechanism is that the direct link and aspect model weaving mechanisms do not address the implementation of cross-cutting variation (e.g. inclusion of a component excludes the existence of another one), while cross-cutting variation can be addressed using integrity constraints over feature models when transformation rules are used.

- **Execution design**
  - *Architectural model:* The architectural model works as a basis for changes performed by runtime reconfiguration middleware of a system. Use of an architectural model understandable by the reconfiguration middleware facilitates the adaptation process. Otherwise, additional transformation from architectural model produced by the planner to a model or a set of actions understandable by the runtime reconfiguration middleware is necessary. One other factor which should be considered in the selection or adoption of an architectural model is selection of appropriate level of abstraction, where all possible architectural variants of system can be expressed while unnecessary implementation related details are removed. Using appropriate level of abstraction for architectural models reduces the complexity of the planner.
  - *Architectural style:* The dynamism of a system is influenced by the architectural platform upon which the system is built. In systems with a component-based architecture, behavioral and component adaptation can be performed by changing component parameters or by replacing a component with another component with the same interface. Yet, the component dependency in a component-based architecture usually prevents architectural adaptation. Therefore, an architectural pattern (e.g. C2 [111], Weaves [112]) should be used in order to enable architectural adaptation, which may not always be feasible. Furthermore, the service-oriented architecture is built on loosely-coupled services by explicitly defining the manner in which these services interact using workflow services, business process languages, or communication channels. Similarly, component adaptation can be achieved by replacing a service with another service with the same interface while architectural adaptation can be performed by changing services interactions.
  - *Variation entity:* Similar to the architectural platform, the selection of a variation entity depends on the degree of the dynamism required for the system. In the problem domains, where component adaptation satisfies adaptation requirements, the variation entity is usually a component or a service. In cases, where an architectural adaptation is required, connectors such as channels

should be adapted in addition to components and services. In the context of DSPL engineering and feature-based planning, aspects have smaller granularity than features, and implementing features using aspects would be effective. However, in order to adapt the system at feature-level at runtime, aspects can be used to modify code of the connectors (e.g. workflow specification languages such as BPEL which work on services). Another important factor in selecting the variation entity of the system is state preservation; components usually have a state which should be preserved if the component is replaced with a new component. Component state preservation requires a supporting mechanism (such as [22, 113]). However, services can be implemented in a stateless manner such that any adaptation over them does not affect the state of the system.

– *Runtime reconfiguration:* The selection of runtime reconfiguration mechanisms is affected by the variation entity as well as the dynamism required for the adaptation. Dynamic aspect weaving is used when the variation entity is an aspect. Otherwise, other mechanisms are used to enable runtime reconfiguration (e.g. component models). Considering that the planner in a DSPL usually operates on the model of the running system, the middleware which is used for enabling runtime reconfiguration should support reflection or allow extension for supporting reflection. Extendibility of the middleware and its component model is usually essential. The component model usually needs to be extended to allow for sensors to be placed within the system or to implement mechanisms for getting and setting component state information.

6. Summary

With the burgeoning demand for runtime adaptation in emerging applications, the field witnesses considerable increase in research. However, developing runtime adaptive systems is considered a complex and challenging task, and existing knowledge in the field requires more research for a wide range of approaches. DSPL engineering approaches suggest the use of extensive product line engineering variability models and mechanisms for the management of runtime adaptation. Research in this area has shown that DSPL is a promising path for developing new solutions.

This paper has focused on providing a comprehensive framework for comparing different DSPL approaches as well as a set of guidelines used to select the appropriate one. We studied these approaches from two perspectives namely adaptation properties and adaptation realization. Different properties of adaptation for a DSPL engineering approach were then reviewed. A framework for comparing adaptation realization in DSPL was consequently proposed. In this framework, the adaptation management was seen as a MAPE-K loop, and a taxonomy for adaptation realization was developed. Using this framework, seven prominent DSPL engineering approaches were compared. Additionally, guidelines were introduced to create the link between the adaptation and its realization properties by outlining dimension values for realization scenarios which best suit different types of adaptations.

Appendix

Citation information have been collected on December 1, 2015.

Table A.1. Reviewed papers.

		Subject				Areas covered				
Study group	Total citations	Conceptual discussions	Development methodologies	Models and Languages	Other (tools, algorithms, ...)	Process	Monitoring and analysing	Planning	Execution	Extension on
[114, 115]	12		x					x		
[38]	18				x					
[86]	19			x						
[49]	32		x			x	x	x	x	
[57]	26		x			x	x	x	x	[16]
[6]	10	x								
[11]	39		x					x	x	
[12, 13]	150		x					x	x	
[7, 19]	389		x					x	x	
[8]	28	x								
[116]	21	x								
[16–18], [50]	190		x					x	x	
[96, 117]	6		x					x		
[64]	1		x					x		
[106]	35			x						
[46, 47, 118]	229		x				x	x	x	[7]
[119]	12				x					
[39]	20			x						
[120]	41	x								
[23–25, 67]	397						x	x	x	
[56]	24	x								
[94]	12		x			x				
[89]	45		x			x				
[121]	2				x					
[122]	9			x						
[26, 27]	99		x			x	x	x	x	
[69]	25			x						
[95]	12	x								
[82]	50			x						
[123]	21		x			x		x		
[124]	4				x					

Table A.1. (*Continued*)

		Subject				Areas covered				
Study group	Total citations	Conceptual discussions	Development methodologies	Models and Languages	Other (tools, algorithms, ...)	Process	Monitoring and analysing	Planning	Execution	Extension on
[125]	66				x					
[41]	54		x			x			x	
[21, 22]	149		x			x		x	x	
[71]	25		x					x		
[126]	2	x								
[127]	1		x						x	
[65]	1		x					x		
[128]	23	x								
[52]	4		x				x			
[100]	0		x			x				
[129]	5		x			x				
[87]	0			x						
[74]	4		x					x		
[99]	0		x						x	
[51]	0			x						
[130]	0	x								
[40]	1		x					x		
[131]	0				x					
[72]	0		x				x			
[110]	1		x					x	x	
[132]	0	x								

References

1. P. Oreizy, N. Medvidovic and R. N. Taylor, Architecture-based runtime software evolution, in *Proc. 20th Int. Conf. Software Engineering*, 1998, pp. 177–186.
2. B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun and B. Cukic, Software engineering for self-adaptive systems: A research roadmap, in *Software Engineering for Self-Adaptive Systems*, 2009, pp. 1–26.
3. D. Weyns, M. U. Iftikhar, S. Malek and J. Andersson, Claims and supporting evidence for self-adaptive systems: A literature study, in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2012, pp. 89–98.
4. Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Mller, M. Pezz and M. Shaw, Engineering self-adaptive systems through feedback loops, in *Software Engineering for Self-Adaptive Systems* (Springer, 2009), pp. 48–70.
5. K. Pohl, G. Bockle and F. V. D. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Vol. 10 (Springer, 2005).
6. V. Alves, D. Schneider, M. Becker, N. Bencomo and P. Grace, Comparative study of variability management in software product lines and runtime adaptable systems, in *Workshop on Variability Modelling of Software-intensive Systems*, 2009, pp. 9–17.
7. S. Hallsteinsen, E. Stav, A. Solberg and J. Floch, Using product line techniques to build adaptive systems, in *Software Product Line Conference*, 2006, pp. 10–150.
8. N. Bencomo, S. Hallsteinsen and E. Almeida, A view of the landscape of dynamic software product lines, *Computer* **45**(10) (2012) 36–41.
9. J. O. Kephart and D. M. Chess, The vision of autonomic computing, *Computer* **36**(1) (2003) 41–50.
10. S. Hallsteinsen, M. Hinchey, S. Park and K. Schmid, Dynamic software product lines, *Computer* **41**(4) (2008) 93–95.
11. L. Baresi, S. Guinea and L. Pasquale, Service-oriented dynamic software product lines, *Computer* **45**(10) (2012) 42–48.
12. N. Bencomo, P. Sawyer, G. Blair and P. Grace, Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems, in *2nd Int. Workshop on Dynamic Software Product Lines*, 2008, p. 40.
13. N. Bencomo, P. Grace, C. Flores, D. Hughes and G. Blair, Genie, in *Int. Conf. Software Engineering*, 2008, pp. 811–814.
14. N. Bencomo and G. Blair, Using architecture models to support the generation and operation of component-based adaptive systems, in *Software Engineering for Self-adaptive Systems* (Springer, 2009), pp. 183–200.
15. N. Bencomo, G. S. Blair, C. A. Flores-Cortes and P. Sawyer, Reflective component-based technologies to support dynamic variability, in *Workshop on Variability Modeling of Software-intensive Systems*, 2008, pp. 141–150.
16. C. Cetina, P. Giner, J. Fons and V. Pelechano, Autonomic computing through reuse of variability models at runtime: The case of smart homes, *Computer* **42**(10) (2009) 37–43.
17. C. Cetina, J. Fons and V. Pelechano, Applying software product lines to build autonomic pervasive systems, in *Software Product Line Conference*, 2008, pp. 117–126.
18. C. Cetina, P. Giner, J. Fons and V. Pelechano, Using feature models for developing self-configuring smart homes, in *Fifth Int. Conf. Autonomic and Autonomous Systems*, 2009, pp. 179–188.
19. J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund and E. Gjorven, Using architecture models for runtime adaptability, *IEEE Softw.* **23**(2) (2006) 62–70.

20. M. Mikalsen, J. Floch, N. Paspallis, G. A. Papadopoulos and P. A. Ruiz, Putting context in context: The role and design of context management in a mobility and adaptation enabling middleware, in *Int. Conf. Mobile Data Management*, 2006, pp. 76–76.
21. H. Gomaa and M. Hussein, Software reconfiguration patterns for dynamic evolution of software architectures, in *Proc. Working IEEE/IFIP Conf. Software Architecture*, 2004, pp. 79–88.
22. H. Gomaa and M. Hussein, Dynamic software reconfiguration in software product families, in *Software Product-Family Engineering* (Springer, 2004), pp. 435–444.
23. B. Morin, O. Barais, J. M. Jzquel, F. Fleurey and A. Solberg, Models@ run. time to support dynamic adaptation, *Computer* **42**(10) (2009) 44–51.
24. B. Morin, F. Fleurey, N. Bencomo, J. M. Jzquel, A. Solberg, V. Dehlen and G. Blair, An aspect-oriented and model-driven approach for managing dynamic variability, in *Model Driven Engineering Languages and Systems*, Vol. 5301 (Springer, 2008), pp. 782–796.
25. B. Morin, O. Barais, J.-M. Jezequel *et al.*, K@ rt: An aspect-oriented and model-oriented framework for dynamic software product lines, in *Proc. 3rd Int. Workshop on Models@ Runtime*, 2008.
26. C. Parra, X. Blanc, A. Cleve and L. Duchien, Unifying design and runtime software adaptation using aspect models, *Sci. Comput. Program.* **76**(12) (2011) 1247–1260.
27. C. Parra, X. Blanc and L. Duchien, Context awareness for dynamic service-oriented product lines, in *Proc. 13th Int. Software Product Line Conference*, 2009, pp. 131–140.
28. O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen and A. Svendsen, Adding standardized variability to domain specific languages, in *Int. Software Product Line Conference*, 2008, pp. 139–148.
29. A. Endpoints, Activebpel: The open source BPEL engine, http://www.activevos.com/learn/open-source.
30. A. Charfi and M. Mezini, Ao4bpel: An aspect-oriented extension to BPEL, *World Wide Web* **10**(3) (2007) 309–344.
31. G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee and J. Ueyama, A component model for building systems software, in *IASTED Software Engineering and Applications*, 2004.
32. J. Munoz and V. Pelechano, Applying software factories to pervasive systems: A platform specific framework, in *ICEIS (3)*, 2006, pp. 337–342.
33. F. V. D. Linden, J. Bosch, E. Kamsties, K. Knsl and H. Obbink, Software product family evaluation, in *Software Product Lines* (Springer, 2004), pp. 110–129.
34. J.-M. Jzquel, Model driven design and aspect weaving, *Softw. Syst. Model.* **7**(2) (2008) 209–218.
35. O. Consortium, Frascati project, http://frascati.ow2.org.
36. O. C. Oasis, Service component architecture (SCA), 2011, http://oasis-opencsa.org/sca.
37. J. Andersson, R. D. Lemos, S. Malek and D. Weyns, Modeling dimensions of self-adaptive software systems, in *Software Engineering for Self-adaptive Systems* (Springer, 2009), pp. 27–47.
38. K. S. Abotsi, S. T. Kurniadi, H. I. Alsawalqah and D. Lee, A software product line-based self-healing strategy for web-based applications, in *Proc. 15th Int. Software Product Line Conference*, Vol. 2, 2011, p. 31.
39. A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout and W. V. Betsbrugge, Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines, in *Proc. Third Int. Workshop on Dynamic Software Product Lines*, 2009, pp. 18–27.
40. L. Baresi and C. Quinton, Dynamically evolving the structural variability of dynamic software product lines, in *10th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems*, 2015, p. 7.

41. R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher and H. Prahofer, Supporting runtime system adaptation through product line engineering and plug-in techniques, in *Int. Conf. Composition-Based Software Systems*, 2008, pp. 21–30.
42. H. Gomaa and K. Hashimoto, Dynamic software adaptation for service-oriented product lines, in *Proc. 15th Int. Software Product Line Conference*, Vol. 2, 2011, p. 35.
43. M. Goedicke, C. Kllmann and U. Zdun, Designing runtime variation points in product line architectures: Three cases, *Sci. Comput. Program.* **53**(3) (2004) 353–380.
44. M. Svahnberg, J. V. Gurp and J. Bosch, A taxonomy of variability realization techniques, *Softw.-Pract. Exp.* **35**(8) (2005) 705–754.
45. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, An architecture-based approach to self-adaptive software, *Intell. Syst. Their Appl.* **14**(3) (1999) 54–62.
46. R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli and U. Scholz, Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments, in *Software Engineering for Self-adaptive Systems* (Springer, 2009), pp. 164–182.
47. J. Floch, C. Fr, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis and H. Rahnama, Playing music: Building context-aware and self-adaptive mobile applications, *Softw.-Pract. Exp.* **43**(3) (2012) 359–388.
48. D. L. McGuinness and F. V. Harmelen, Owl web ontology language overview, *W3C Recommend.* **10**(2004–03) (2004) 10.
49. G. Alfrez, V. Pelechano, R. Mazo, C. Salinesi and D. Diaz, Dynamic adaptation of service compositions with variability models, *J. Syst. Softw.* **91** (2014) 24–47.
50. C. Cetina, P. Giner, J. Fons and V. Pelechano, Designing and prototyping dynamic software product lines: Techniques and guidelines, in *Software Product Lines: Going Beyond* (Springer, 2010), pp. 331–345.
51. S. Neskovic and R. Matic, Context modeling based on feature models expressed as views on ontologies via mappings, *Comput. Sci. Inf. Syst.* **12**(3) (2015) 961–977.
52. A. Murguzur, R. Capilla, S. Trujillo, S. Ortiz and R. E. Lopez-Herrejon, Context variability modeling for runtime configuration of service-based dynamic software product lines, in *Proc. 18th Int. Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools*, Vol. 2, 2014, pp. 2–9.
53. J. Zhang and B. H. Cheng, Model-based development of dynamically adaptive software, in *Proc. 28th Int. Conf. Software Engineering*, 2006, pp. 371–380.
54. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, Form: A feature-oriented reuse method with domain-specific reference architectures, *Ann. Softw. Eng.* **5**(1) (1998) 143–168.
55. K. Czarnecki, U. Eisenecker, R. Glck, D. Vandevoorde and T. Veldhuizen, Generative programming and active libraries, in *Generic Programming* (Springer-Verlag, 2000), pp. 25–39.
56. P. Istoan, G. Nain, G. Perrouin and J.-M. Jzquel, Dynamic software product lines for service-based systems, in *Ninth IEEE Int. Conf. Computer and Information Technology*, Vol. 2, 2009, pp. 193–198.
57. G. H. Alferez and V. Pelechano, Context-aware autonomous web services in software product lines, in *Int. Software Product Line Conference*, 2011, pp. 100–109.
58. D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Corts, Fama: Tooling a framework for the automated analysis of feature models, in *Proc. First Int. Workshop on Variability Modelling of Software-Intensive Systems*, 2007, pp. 129–134.
59. H. Hartmann and T. Trew, Using feature diagrams with context variability to model multiple product lines for software supply chains, in *12th Int. Software Product Line Conference*, 2008, pp. 12–21.

60. T. T. Tun, Q. Boucher, A. Classen, A. Hubaux and P. Heymans, Relating requirements and feature configurations: A systematic approach, in *Proc. 13th Int. Software Product Line Conference*, 2009, pp. 201–210.
61. K. Lee and K. C. Kang, Usage context as key driver for feature selection, in *Software Product Lines: Going Beyond* (Springer, 2010), pp. 32–46.
62. M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan and J.-P. Rigault, Modeling context and dynamic adaptations with feature models, in *Proc. 4th Int. Workshop Models@ run. time*, 2009.
63. F. Fleurey and A. Solberg, A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems, in *Model Driven Engineering Languages and Systems* (Springer, 2009), pp. 606–621.
64. M. Derakhshanmanesh, M. Salehie and J. Ebert, Towards model-centric engineering of a dynamic access control product line, in *Proc. 16th Int. Software Product Line Conference-Volume 2*, 2012, pp. 151–155.
65. L. E. Sanchez, J. A. Diaz-Pace, A. Zunino, S. Moisan and J.-P. Rigault, An approach for managing quality attributes at runtime using feature models, in *Eighth Brazilian Symp. Software Components, Architectures and Reuse*, 2014, pp. 11–20.
66. J. V. Gurp, J. Bosch and M. Svahnberg, On the notion of variability in software product lines, in *Proc. Working IEEE/IFIP Conf. Software Architecture*, 2001, pp. 45–54.
67. J. F. I. Romero, C. V. Chicote, B. Morin and O. Barais, Using models@ runtime for designing adaptive robotics software: An experience report, in *1st Int. Workshop on Model Based Engineering for Robotics: RoSym10, Artist Consortium*, 2010.
68. M. C. Huebscher and J. A. McCann, A survey of autonomic computingdegrees, models, and applications, *ACM Comput. Surv.* **40**(3) (2008) 1–28.
69. M. Rosenmller, N. Siegmund, M. Pukall and S. Apel, Tailoring dynamic software product lines, in *Proc. 10th ACM Int. Conf. Generative Programming and Component Engineering*, 2011, pp. 3–12.
70. C. Parra, D. Romero, S. Mosser, R. Rouvoy, L. Duchien and L. Seinturier, Using constraint-based optimization and variability to support continuous self-adaptation, in *Proc. 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 486–491.
71. P. Sawyer, R. Mazo, D. Diaz, C. Salinesi and D. Hughes, Using constraint programming to manage configurations in self-adaptive systems, *Computer* **45**(10) (2012) 56–63.
72. L. Dounas, R. Mazo, C. Salinesi and O. E. Beqqali, Runtime requirements monitoring framework for adaptive e-learning systems, in *Int. Conf. Software & Systems Engineering and Their Applications*, 2015.
73. K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli *et al.*, A comprehensive solution for application-level adaptation, *Softw.-Pract. Exp.* **39**(4) (2009) 385–422.
74. G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes and A. Egyed, Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications, *J. Syst. Softw.* **103** (2015) 392–411.
75. M. Voelter and I. Groher, Product line implementation using aspect-oriented and model-driven software development, in *Int. Software Product Line Conference*, 2007, pp. 233–242.
76. S. Apel and C. Kstner, An overview of feature-oriented software development, *J. Object Technol.* **8**(5) (2009) 49–84.
77. G. Perrouin, J. Klein, N. Guelfi and J.-M. Jzquel, Reconciling automation and flexibility in product derivation, in *Int. Software Product Line Conference*, 2008, pp. 339–348.

78. K. Czarnecki and M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in *Generative Programming and Component Engineering* (Springer, 2005), pp. 422–437.
79. T. Ziadi and J.-M. Jzquel, Software product line engineering with the UML: Deriving products, in *Software Product Lines* (Springer, 2006), pp. 557–588.
80. I. Schaefer, Variability modeling for model-driven development of software product lines, in *Proc. 4th Int. Workshop on Variability Modelling of Software-intensive Systems*, 2010, pp. 85–92.
81. J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale and D. C. Schmidt, Improving domain-specific language reuse with software product line techniques, *IEEE Softw.* **26** (4) (2009) 47–53.
82. P. Trinidad, A. Ruiz-Corts, J. Pena and D. Benavides, Mapping feature models onto component models to build dynamic software product lines, in *Int. Workshop on Dynamic Software Product Line*, 2007.
83. P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais and J.-M. Jezequel, Introducing variability into aspect-oriented modeling approaches, in *Model Driven Engineering Languages and Systems* (Springer, 2007), pp. 498–513.
84. F. Fleurey, B. Baudry, R. France and S. Ghosh, A generic approach for automatic model composition, in *Models in Software Engineering* (Springer, 2008), pp. 7–15.
85. B. Morin, O. Barais, G. Nain and J.-M. Jezequel, Taming dynamically adaptive systems using models and aspects, in *Proc. 31st Int. Conf. Software Engineering*, 2009, pp. 122–132.
86. M. Acher, P. Collet, P. Lahire, S. Moisan and J. P. Rigault, Modeling variability from requirements to runtime, in *16th IEEE Int. Conf. Engineering of Complex Computer Systems*, 2011, p. 77.
87. S. Adjoyan and A. Seriai, An architecture description language for dynamic service-oriented product lines, in *27th Int. Conf. Software Engineering and Knowledge Engineering*, 2015.
88. R. N. Taylor, N. Medvidovic and P. Oreizy, Architectural styles for runtime software adaptation, in *European Conf. Software Architecture*, 2009, pp. 171–180.
89. J. Lee and G. Kotonya, Combining service-orientation with product line engineering, *IEEE Softw.* **27**(3) (2010) 35–41.
90. F. M. Medeiros, E. S. de Almeida and S. R. de Lemos Meira, Towards an approach for service-oriented product line architectures, in *Proc. Workshop on Service-Oriented Architectures and Software Product Lines*, 2009, pp. 1–7.
91. B. Mohabbati, M. Hatala, D. GaÅevi, M. Asadi and M. BoÅkovi, Development and configuration of service-oriented systems families, in *Proc. ACM Symp. Applied Computing*, 2011, pp. 1606–1613.
92. M. Abu-Matar and H. Gomaa, Variability modeling for service oriented product line architectures, in *Int. Software Product Line Conference*, 2011, pp. 110–119.
93. M. Salehie and L. Tahvildari, Self-adaptive software: Landscape and research challenges, *ACM Trans. Auton. Adap. Syst.* **4**(2) (2009) 14.
94. J. Lee, G. Kotonya and D. Robinson, Engineering service-based dynamic software product lines, *Computer* **45**(10) (2012) 49–55.
95. H. Shokry and M. A. Babar, Dynamic software product line architectures using service-based computing for automotive systems, in *Proc. Second Int. Workshop on Dynamic Software Product Lines*, 2008.
96. J. Cubo, N. Gamez, L. Fuentes and E. Pimentel, *Composition and Self-Adaptation of Service-Based Systems with Feature Models, Safe and Secure Software Reuse* (Springer, 2013), pp. 326–342.

97. C. Szyperski, *Component Software: Beyond Object-Oriented Programming* (Pearson Education, 2002).
98. C. Kastner, S. Apel and D. Batory, A case study implementing features using aspectj, in *Int. Software Product Line Conference*, 2007, pp. 223–232.
99. R. Andrade, M. Ribeiro, H. Rebêlo, P. Borba, V. Gasiunas and L. Satabin, Assessing idioms for a flexible feature binding time, *Comput. J.* **59**(1) (2015) 1–32.
100. R. dos Santos Rocha, M. Fantinato, L. H. Thom and M. M. Eler, Dynamic product line for business process management, *Bus. Process Manag. J.* **21**(6) (2015) 1224–1256.
101. B. C. Smith, Reflection and semantics in a procedural language, Ph.D. thesis, MIT (1982).
102. K.-K. Lau and Z. Wang, Software component models, *IEEE Trans. Softw. Eng.* **33**(10) (2007) 709–724.
103. E. Bruneton, T. Coupaye, M. Leclercq, V. Quma and J. Stefani, The fractal component model and its support in Java, *Softw.-Pract. Exp.* **36**(1112) (2006) 1257–1284.
104. D. Marples and P. Kriens, The open services gateway initiative: An introductory overview, *IEEE Commun. Mag.* **39**(12) (2001) 110–114.
105. A. Popovici, T. Gross and G. Alonso, Dynamic weaving for aspect-oriented programming, in *Proc. 1st Int. Conf. Aspect-oriented Software Development*, 2002, pp. 141–147.
106. T. Dinkelaker, R. Mitschke, K. Fetzer and M. Mezini, A dynamic software product line approach using aspect models at runtime, in *5th Domain-Specific Aspect Languages Workshop*, 2010.
107. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, An overview of AspectJ, in *ECOOP 2001: Object-Oriented Programming* (Springer, 2001), pp. 327–354.
108. N. Delgado, A. Q. Gates and S. Roach, A taxonomy and catalog of runtime software-fault monitoring tools, *IEEE Trans. Softw. Eng.* **30**(12) (2004) 859–872.
109. J. Ehlers and W. Hasselbring, A self-adaptive monitoring framework for component-based software systems, in *Software Architecture* (Springer, 2011), pp. 278–286.
110. M. Lochau, J. Brdek, S. Hlzle and A. Schrr, Specification and automated validation of staged reconfiguration processes for dynamic software product lines, *Softw. Syst. Model.* (2015) 1–28.
111. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr, J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow, A component- and message-based architectural style for gui software, *IEEE Trans. Softw. Eng.* **22**(6) (1996) 390–406.
112. M. M. Gorlick and R. R. Razouk, Using weaves for software construction and analysis, in *Int. Conf. Software Engineering*, 1991, pp. 23–34.
113. J. P. A. Almeida, M. Wegdam, M. V. Sinderen and L. Nieuwenhuis, Transparent dynamic reconfiguration for corba, in *Proc. 3rd Int. Symp. Distributed Objects and Applications*, 2001, pp. 197–207.
114. N. Abbas, Towards autonomic software product lines, in *Proc. 15th Int. Software Product Line Conference*, Vol. 2, 2011, p. 44.
115. N. Abbas, J. Andersson and W. Lowe, Autonomic software product lines (aspl), in *Proc. Fourth European Conf. Software Architecture: Companion Volume*, 2010, pp. 324–331.
116. J. Bosch and R. Capilla, Dynamic variability in software-intensive embedded system families, Dynamic Variability in Software-Intensive Embedded System Families, *Computer* **45**(10) (2012) 28–35.
117. J. Cubo, N. Gamez, E. Pimentel and L. Fuentes, Reconfiguration of service failures in damasco using dynamic software product lines, in *IEEE Int. Conf. Services Computing*, 2015, pp. 114–121.
118. R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen and E. Stav, Composing components and services using a planning-based adaptation middleware, in *Software Composition* (Springer, 2008), pp. 52–67.

119. S. Hallsteinsen, S. Jiang and R. Sanders, Dynamic software product lines in service oriented computing, in *3rd Int. Workshop on Dynamic Software Product Lines*, 2009, p. 5.
120. M. Hinchey, S. Park and K. Schmid, Building dynamic software product lines, *Computer* **45**(10) (2012) 22–26.
121. J. Lee, J. Whittle and O. Storz, Bio-inspired mechanisms for coordinating multiple instances of a service feature in dynamic software product lines, *J. Univ. Comput. Sci.* **17**(5) (2011) 670–683.
122. S. Nakajima, An architecture of dynamically adaptive php-based web applications, in *Asia Pacific Software Engineering Conference*, 2011, pp. 203–210.
123. Y. Wang, A. Kobsa, A. V. D. Hoek and J. White, Pla-based runtime dynamism in support of privacy-enhanced web personalization, in *10th Int. Software Product Line Conference*, 2006, p. 10.
124. D. Weyns, A. Helleboogh, T. Holvoet, K. Schelfthout and W. V. Betsbrugge, Towards a software product line for automated transportation systems, in *2nd Int. Workshop on Dynamic Software Product Lines*, 2008, pp. 45–52.
125. J. White, D. C. Schmidt, E. Wuchner and A. Nechypurenko, Automating product-line variant selection for mobile devices, in *Int. Software Product Line Conference*, 2007, pp. 129–140.
126. J. R. F. da Silva, F. A. Pereira da Silva, L. M. do Nascimento, D. Martins, V. C. Garcia *et al.*, The dynamic aspects of product derivation in dspl: A systematic literature review, in *IEEE 14th Int. Conf. Information Reuse and Integration*, 2013, pp. 466–473.
127. D. Kramer, S. Oussena, P. Komisarczuk and T. Clark, Using document-oriented guis in dynamic software product lines, in *ACM SIGPLAN Notices* **49** (2013) 85–94.
128. R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Corts and M. Hinchey, An overview of dynamic software product line architectures and techniques: Observations from research and industry, *J. Syst. Softw.* **91** (2014) 3–23.
129. H. Moens and F. D. Turck, Feature-based application development and management of multi-tenant applications in clouds, in *Proc. 18th Int. Software Product Line Conference*, 2014, pp. 72–81.
130. J. D. Eleutrio, F. N. Gaia, G. N. Rodrigues and C. M. Rubira, Dependable dynamic software product linea systematic mapping study, Technical Report IC-15–04, University of Campinas, 2015.
131. J. C. Munoz-Fernandez, G. Tamura, I. Raicu, R. Mazo and C. Salinesi, Refas: A ple approach for simulation of self-adaptive systems requirements, in *Proc. 19th Int. Conf. Software Product Line*, 2015, pp. 121–125, http://doi.acm.org/10.1145/2791060.2791102.
132. G. Guedes, C. Silva, M. Soares and J. Castro, Variability management in dynamic software product lines: A systematic mapping, in *2015 IX Brazilian Symp. Components, Architectures and Reuse Software*, 2015, pp. 90–99.