TECHMIND RESEARCH
ISSN: 2291-2118
International Journal of Research in Business and Technology
Volume 4 No.2 June 2014

Towards a Dynamic Software Product Line: Analysis of the Background and State of the Art

Chiraz BOUZID[1], Naoufel KRAIEM[2], Zuhoor Al Khanjari[3]
[1]RIADI Lab, ENSI, Campus of Manouba Manouba, Tunisia
[2,3] Department of Computer Science, Sultan Qaboos University, Muscat, Oman
[1]Chiraz.bouzid@yahoo.fr
[2]naoufel@squ.edu.om,
[3]zuhoor@squ.edu.om

Abstract- *Dynamic software adaptability is one of the central features leveraged by autonomic computing. However, developing software that changes its behavior at run time in response to dynamically varying user needs and resource constraints is a challenging task. With the emergence of mobile and service oriented computing, such variation is becoming increasingly common, and the need for adaptivity is increasing accordingly. Software product line engineering has proved itself as an efficient way to deal with varying user needs and resource constraints. In this paper, we present a study of different approaches for design and runtime adaptation that can be used in the context of Dynamic Software Product Lines (DSPLs). We propose a classification and a comparison of existing work. Afterwards, we refine our proposal by concretizing the research goals that fulfill the gaps current approaches present.*

Keywords- *Dynamic Software Adaptability; Software Product Line Engineering; Dynamic Software Product Line.*

1. INTRODUCTION

Dynamic Software Product Lines (DSPL) intends to face several challenges related to the continuous changes in software at design and execution. More and more, these modifications become the rule rather than the exception. At design time, developers have to produce a family of software products instead of individual applications for solving one problem. They usually specify the software using a composition of concerns in order to obtain a complete software definition. Such a separation and composition of concerns facilitates either the definition of successive versions in the time, or different variants for different target platforms or user requirements. Moreover, software can also evolve at runtime in order to dynamically consider new requirements or context changes. This last change could be managed by self-adaptive platforms. These platforms enable software systems to add and remove some of its elements at runtime.

The term DSPL was introduced in 2008 by Hallsteinsen et al. [12]. In [12] authors introduce a new trend in research that aims at using the principles of traditional SPL to build products that can be adapted at runtime depending on the requirements of the users and the conditions of the environment. Because of its novelty, literature is yet scarce with regard to concrete DSPL approaches. Nevertheless, DSPL challenges can be faced using already mature approaches in software engineering, especially when dealing with software adaptation (e.g. AOM, service and component-based runtime platforms, MDE, ECA rules). This article aims at studying different approaches for design and runtime adaptation that can be used in the context of DSPLs. We propose a classification and a comparison of existing work. Afterwards, we refine our proposal by concretizing the research goals that fulfill the gaps current approaches present.

The remainder of this paper is organized as follows. In Section 2 we discuss the adaptation and present the two main types of adaptation considered for this survey: adaptation at design time, and adaptation at runtime. In Section 3 we elaborate on the criteria and classification as well as the description of the approaches surveyed. We discuss additional criteria for the comparison that, although not present in all the works surveyed, is worth mention for the relevance in the context of DSPL. At the end of the section we present a summary of the results. Next, in Section 4 we present the research goals to better position the contributions of our work. Sub sections A and B of Section 4 elaborate on the need for a unification of adaptations and the challenges of defining and implementing such unification. Finally we conclude in Section 5 with a brief summary of the survey presented in the chapter.

2. DSPL AND SOFTWARE ADAPTATION

DSPLs focus on the development of software products that can be adapted at runtime depending on the requirements of the users and the conditions of the environment. Indeed with the increasing need of self-managed systems and the emergence of multi-scale environments, software developers need to cope with

variability and adaptations. Software must be developed to be adapted and reconfigured automatically on heterogeneous platforms in accordance with the unavoidable evolution of information and communication technologies. Therefore, the adaptation is now considered as a first-class problem that must be taken into account throughout the software life-cycle. In order to position our work, we start by presenting the definition of adaptation, and its implications at design time and at runtime respectively.

2.1 Adaptation

Software adaptation is strongly related to software evolution. Both processes deal with the modification of an application. However, as presented on [22, 23], such processes are complementary with regard to the focus and tasks that they involve. A software evolution is understood as the modifications done to a system over time. The adaptation is more related to the processes needed to modify an application including: detecting events and information that may lead to a change, planning a set of changes, and performing those changes on the application. A well-known reference of this model is the one presented by IBM [13] known as MAPE for the phases it includes: Monitor, Analyze, Plan, and Execute. The IBM model has been defined for control loops at runtime. However, software can be adapted either at the design phase or at the runtime phase. For each phase, dedicated technologies are used to specify and realize the adaptations.

2.2 Static and Dynamic Adaptation

Another characteristic of adaptation is the moment of time in which the business code is adapted. Literature in general refers to two types of adaptation: static and dynamic. Static adaptation refers to the changes that are performed during development, compile or load time. During the development for instance, design languages provide adaptation mechanisms such as inheritance or composition. A slightly different approach is to adapt the application at compile time. One of the better-known examples that allow this type of adaptation is AspectJ [15], an aspect-orientation extension of java. With AspectJ, crosscutting features can be defined and woven with original business code at compile time. Load-time adaptation is also considered as a way of static adaptation. This kind of adaptation consists in waiting until the loading of an application to decide which components are employed. For example, as explained in [20], when an application requests the loading of a new component, decision logic might select from a list of components with different capabilities or implementation, choosing the one that most closely matches current needs.

Dynamic adaptation refers to changes that happen while the applications are being executed. This means that elements of the application such as algorithms or structures can be replaced or modified during execution without necessarily having to halt and restart the application [20]. Typically, at runtime, applications are based on platforms that support dynamic adaptation. For instance, certain CBSE platforms provide APIs to dynamically change connections between running components.

2.3 Adaptation at design time

In our work we intend to face the challenges for the adaptations at design time. From an SPL perspective, it does not matter if the adaptation takes place at the level of models or by modifying the source code because in both cases, the adaptations are part of the derivation of a product from a user-defined configuration. For this reason we group the static adaptations techniques under the same SPL process of application engineering at design time. We consider an adaptation at design time as any modification performed over an application that starts and ends before the application has been deployed ant its execution has actually taken place.

2.4 Adaptation at runtime

In a similar way as for the adaptation at design time, we group the different approaches for achieving dynamic adaptations under the same process of application engineering at runtime. We consider that independently from the approach, all shares the same objective of changing the applications dynamically. Consequently, we define the notion of adaptation at runtime as any modification of the application that takes place during its execution.

3. DESIGN AND RUNTIME ADAPTATION: APPROACHES AND MECHANISMS

In this section, we survey different approaches that are related to the definition and implementation of SPLs for deriving adaptive systems. We classify the approaches based on the type of adaptation they support. There are three main groups: (1) those who specifically deal with design time adaptations, those who specifically deal with runtime adaptations, and (3) those who try to cover both processes at the same time.

3.1 Design time adaptation approaches

This category includes all the approaches where the adaptation takes place before the deployment of the software artifacts that constitute the application. Usually, approaches in this category present a complete derivation process that uses variability and variability constraints for product derivation, as well as mappings and code generation processes for building the concrete artifacts that constitute the software products.

Design time criteria

Each approach in the design time category is classified regarding two main criteria: (1) the mechanisms used, and (2) the scope of the adaptation. These criteria are detailed in Table 1.

Arboleda and al. [2] propose a Software Product Line based on Models. Their approach uses variability and constraint models in combination with AOP to derive products that integrate different concerns into a single product. All the operations to derive a product occur at

design time (merging models and code generation). Regarding the scope of the adaptation, this approach emphasizes on the adaptation at the level of models and code.

Table 1: Comparison criteria for design time approaches.

Criteria	Definition
Mechanisms	The mechanisms used for defining and implementing the adaptation. This includes but is not limited to models and model transformations, aspect oriented modeling and model merging, and feature diagrams.
AOM, AOP	AOP and AOM are also commonly used across the different approaches as a way to achieve modularization and adaptation based on the composition of multiple modules. Aspects can be combined with feature diagrams as a way to deal with variability and constraints among several products in software product lines.
MDE	MDE is widely accepted in design adaptations. One common strategy among several approaches is the use of model transformations and code generation to automate the development of applications
Variability Management	Several approaches are based on SPL and variability management to configure and build families of similar products. Adaptation is achieved by switching across several product configurations. Typically, variability management is combined with other mechanisms like models or aspect oriented programming.
Scope	By scope we mean the granularity of the adaptation, it varies from fine grained granularity, as modifying methods and parameters, to coarse grained granularity, when doing architectural modifications like changing component bindings.
Model	Several approaches use models to represent applications at both design (most MDA approaches) and runtime (models at runtime). We say that the scope of the adaptation is a model if the elements that get modified because of the adaptation are models.
Architecture	For approaches where applications are based on architectural paradigms like components, services, processes (e.g. CBSE, SOA, BPEL), we evaluate if the adaptation has an impact on the structure or behavior of the elements that constitute the architecture of the application.
Code	Finally, we say that the code is the scope of the adaptation when parts of the source code (e.g. classes, methods, attributes) implementing the applications are impacted by the adaptation. For example, AOP approaches define explicit pointcuts on the source code, to extend them with added functionality.

The adaptation modifies the models used to represent the product. Besides, since they use AOP, source code is also the target of modifications during the derivation process. In terms of mechanisms employed, the approach defines the variability of the family of products, and uses MDE to define intermediate models and AOP to compose model transformation rules.

Clarke [8] discusses about composition mechanisms needed in particular in the UML metamodel to align requirements and objects. She proposes to add a specific composition relationship among elements, so that, common elements in different models (regarding the same requirements) can be identified and composed. Using this composition relationship, she discusses two ways of performing composition: merging and override. As in the previous case, the composition takes place at design time, among the different UML models. Since the result of the composition proposed are new UML models, the scope of this approach are the models. Regarding the mechanisms used, the approach uses MDE for representing the models and for defining the composition mechanisms that correspond to the same requirements.

Czarnecki and Antkiewicz [7] propose a mapping from feature models to application models. The idea is to allow the modeler to view directly the assets related to each feature and estimate the impact of selecting/deselecting a given variant. With a particular configuration, a template instance is obtained which represents the selection of the modeler. A template corresponds to design elements like UML diagrams. The approach focuses on design-time derivation since the results of the configuration corresponds to a UML model. Regarding the scope, the mapping of feature models to application models implies that the models and the architecture of the application are modified. Indeed, authors deal with both models and templates at the same time. While models are used mainly to represent variability, templates are used to represent design elements like UML diagrams which define the architecture of the applications being derived. The mechanisms used by the authors combine mainly MDE for

modeling the applications and variability management to map such models to features.

Kienzle and al. [14] define aspects over UML diagrams. They use class diagrams for structure modeling, as well as sequence and state diagrams for behavior modeling. Afterwards, their approach proposes a weaving that composes such models, including dependency chains among them. Since the result of the weaving is a new model that can be used for simulation or code generation the approach is centered on design-time adaptation. The scope of the approach is the models that get composed thanks to the weaving of the aspects defined. The mechanisms used by Kienzle et. al. combine AOM for defining the aspects, and MDE for the creation of class, sequence, and state diagrams.

Perrouin and al. [25] propose a model-based approach at design time for product derivation in SPLs. They start with a feature model, and for each feature, there is a partial model. A merging operation takes place in order to merge the partial models of the different features selected for a particular configuration. The adaptation target corresponds to the models, since the merging that combines different features results in a merged model. Regarding the mechanisms, Perrouin et. al. combine variability for defining the feature model with MDE for defining the models and the merging operation.

Reddy and al. [27] present an aspect based approach for model composition. They introduce a base algorithm and different directives. The directives are used when the composition algorithm yields to incorrect results. Directives modify default composition rules, so that developers have finer-grained control on how the elements of the models are composed. Their approach focuses on aspect models and design composition. The adaptation scope in the approach are the models obtained after modifying the composition rules. The approach is mainly based on AOM and MDE for defining the elements to compose, the base composition algorithm, and the directives that modify the rules of such algorithm for the cases where there is a conflict.

Sánchez and al. [28] define a language for composition of assets in SPL called VLM4. This language can be used to generate model transformation rules that automate the derivation process at design time. The approach aims at creating model transformations that in the end produce as a result a model that represents the SPL configuration. Authors use variability for defining the assets to compose and MDE transformations that are generated from their own language.

Voelter and Groher [30] propose a complete model driven SPL where aspects are used to realize variability. AOM and AOP are both used to introduce variability at the level of models, and later at the level of generated code. The scope of this approach covers both the models when using AOM, and code when the adaptation takes place through AOP.

Wagelaar [31] proposes a way to take modularization to the level of rule-based model transformations. He proposes a composition of rules so that different independent transformations can be combined and scale up to a larger model transformation. Since combining transformation rules is equivalent to modify the model that results from executing them, we consider that the scope of the approach are the models. Wagelaar focuses on MDE techniques and particularly in the combination of model to model transformations.

Van der Storm [11] proposes a formal model to bridge domain and solution models in product line engineering. His approach is based on dependency graphs that map concepts from feature diagrams to software artifacts. As with the previous approaches, the domain and solution models are used at design time during the development process of the applications. The approach by Van der Storm has also the models as its scope, since its main goal is to define a formal model which allows adaptation based on feature selection (domain problem) into the software artifacts (solution). Regarding the mechanisms, Van der Storm approach is based on variability management and SPL techniques.

Finally, Lee and al. [16] work on product derivation by means of an aspect oriented solution to the problem of feature dependencies. Aspects are used as a way to separate feature dependencies from feature implementations. This approach attacks directly the source code of the applications being implemented, by defining aspects that are woven depending on the feature dependencies. Aspects are combined with feature diagrams as a way to deal with variability and constraints among several products in software product lines.

Summary of design time approaches

As we have shown, work on adaptation at design time is prolific. Different scopes are defined as well as different mechanisms for achieving such adaptations. The results of this first group are summarized in Table 2. The first column contains the reference of the work. We have two main columns for **Scope**, and **Mechanisms**. Each main column contains their respective criteria subgroups. Additionally, we have added an extra column called **Domain** to indicate, if available, the kind of domain of application (e.g., mobile computing, embedded systems, smart houses, and multimedia). We use a check mark (X) if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

From this first group of approaches that focus on design time, we can observe that most of them include complete derivation processes by defining the variability of the products at early stages of the development. All of them can be used (at least partially) to produce families of products from different product configurations. They combine variability management with concrete techniques for software development like MDE in the case of a PIM to PSM transformation chain, AOM when modularization and composition are employed at the level of models, or AOP when aspects are woven directly to the source code. However, due to the lack of support for dynamic adaptations, these approaches only face a subset of the

challenges for DSPLs. Concretely, there is no support for adaptations at runtime. This implies that the configuration defined for each product at design time does not exist when the product is executed. We consider that a complete approach for DSPL should not only deal with the design adaptations this group of approaches support, but also with the requirements for adaptations during the execution of the applications.

Table2: Summary of the design time adaptation approaches.

Reference	Scope			Mechanism				Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	Mobile Computing	Embedded systems	Smart House	Multimedia
Arboleda and al.	–	X	X	X	X	X	X	X	–	X	–
Clark	X	–	–	–	–	X	–	–	–	–	–
Czarnecki and Antkiewiez	X	X	–	–	–	X	X	–	–	–	–
Kienzle and al.	X	–	–	X	–	X	–	–	–	–	–
Perrouin and al.	X	–	–	–	–	X	X	–	–	–	–
Reddy and al.	X	–	–	X	–	–	–	–	–	–	–
Sanchez and al.	X	–	–	–	–	X	X	–	–	X	–
Voelter and Groher	X	–	X	X	X	X	X	–	–	X	–
Wagelaar	X	–	–	–	–	X	–	–	–	–	–
Van der Storm	X	–	–	–	–	X	X	–	–	–	–
Lee and al.	–	–	X	–	X	–	X	–	–	–	–

3.2 Runtime adaptation approaches

This category includes all the approaches where the adaptation takes place during the execution of the application. Usually, approaches in this category aim at defining adaptation rules and at taking advantage of technologies that allow for runtime modifications of the applications.

Runtime reconfigurations are performed in different ways and using a variety of tools (i.e., introspection and intersection, meta object protocols, models at runtime, runtime platforms based on services and/or components).

Runtime adaptation criteria

To properly identify the different mechanisms used in this category, we have slightly modified the criteria. In addition to the elements previously identified for the design time approaches that remain valid, we have added the ECA rules in the Mechanisms criteria for approaches that are based on rules, and conditions. Additionally, we have added a new criterion called Context Awareness, that is used to classify the approaches that use context information to trigger the process of adaptation dynamically. Table 3 details the new criteria for runtime approaches.

Table 3: Comparison criteria for runtime approaches.

Criteria	Definition
Mechanisms	In the same way as for the design adaptations, the runtime mechanisms cover the different techniques and approaches used to achieve the adaptation.
ECA Rules	Event condition-action (ECA) rules are mechanisms employed when it is necessary to trigger a particular action in response to events. This type of mechanisms are mainly used to model an adaptation in response to changes in the execution context.
Context Awareness	Context awareness refers to the capability of the systems to react to changes in their environment [5, 34]. Context information refers to all the information available in the environment when applications are being executed, and that may affect the structure or behavior of them. Examples of context information include location, temperature, hardware constraints, user preferences and personal information,

	time, etc. For this criteria we identify the approaches that effectively use context information as input in the decision making process particularly in the case of runtime adaptations.

In [3] Batista et al. introduce their framework called PLASTIK. It allows both the definition of runtime components as well as their dynamic reconfiguration. It is a combination of an ADL for describing architectures, with a reflective component model. Runtime adaptation is achieved through reconfigurations that can be of two types: programmed reconfigurations, which are foreseen at design time, and ad-hoc reconfigurations, which cannot be foreseen at design and that are controlled with the help of invariants in the specification of the system. PLASTIK uses models at runtime together with ECA rules for achieving the adaptation. The component-based platform called OpenCOM and the ADL with extensions allow developers to define ECA-like conditions on which reconfiguration actions take place.

In [6], Bencomo et al. propose software product lines for adaptive systems. In their approach, a complete specification of the context and supported changes has to be provided thanks to a state machine. Each state then represents a particular variant of the system and transitions between states define dynamic adaptations that are triggered by events corresponding to context changes. The work of Bencomo et al. defines reconfiguration policies that take the form of on-event-do-action, where actions are changes to component configurations and events represent the notifications arriving from the environment and processed by a context engine.

David and Ledoux [9] present SAFRAN, an extension of the FRACTAL component model in order to modularize dynamic adaptations using aspects. The aspects represent reactive adaptation policies which trigger reconfigurations based on evolutions of the context. The adaptation is defined using FScript, a language developed to write Fractal component reconfigurations. They useWildCAT [35] to detect external events. WildCAT models context as a set of domains. Each domain represents a particular aspect of the context information. The information itself is modeled as pairs (name, value) inside every domain. The information changes over time and these changes generate events, which are used by SAFRAN to trigger the adaptation process.

Pessemier et al. [26] introduce the Fractal Aspect Components (FAC). FAC is a model for software evolution that benefits from Aspect Oriented Software Development (AOSD) and Component Based Software Engineering (CBSE) [37]. In FAC, there can be aspect components, which are regular Fractal components that embody an advice code. The adaptation takes place by adding or removing components (aspect or regular) to running applications. The runtime reconfiguration is, as in the previous case based on the support provided by the FRACTAL component model. Since FAC is based on Fractal components and use Fractal dynamic capabilities to define adaptations at the architecture level, the scope of the adaptation corresponds to the architecture of the component-based application that gets modified through FRACTAL reconfigurations.

Zhang and Cheng [33] introduce a model driven process for the development of dynamic programs. Formal models are created for the behavior based on states. They separate adaptive from non-adaptive behavior in programs, making the models easier to specify. Trinidad and al. [29] propose a mapping from feature models onto component models. Basically, for each feature, there is one component who implements it. There is additionally a component called the configurator which is in charge of creating the bindings to form the desired architecture that represents a particular feature configuration. The configurator acts at runtime and is able to activate components linked to non-core features. The approach focuses on the relationship of features and software components. Adaptation takes place thanks to a configurator component that modifies the architecture of the applications components and bindings at runtime.

Finally, Dinkelaker et al. [10] propose a dynamic software product line using aspect models at runtime. They use aspect models to define features and feature constraints. Their approach mixes SPL principles of product derivation with the notion of dynamic variability. They distinguish static variability from dynamic variability, and for the latter one, they use dynamic AOP for the implementation. Their approach links what they call dynamic features, representing late variation points in an SPL, to dynamic aspects.

Summary of runtime adaptation approaches

Table 4 summarizes the approaches of the second category. We have added the ECA rules criterion for the mechanisms, and the context awareness. In the same way as for the previous group, a check mark (X) indicates if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

In this second category we find approaches that offer great support for dynamic adaptations. They are usually based on platforms with reflective capabilities that they use to modify applications at runtime. Some of the approaches use context information and event rules to trigger adaptations, whenever a context occurs. However, such approaches do not offer support for design adaptations. Their starting point is usually a set of applications already developed (by hand in most cases), and they are not interested on automating the development process before the execution. A DSPL approach can take advantage of the dynamic capabilities and runtime adaptations offered by the approaches in this category. Nevertheless, the lack of adaptations at design time, make us consider that this second category of approaches are only suitable to face a subset of the challenges for DSPLs. They have to be complemented to offer support for the initial development process that takes place before the execution.

Table 4: Summary of the runtime adaptation approaches.

Reference	Scope			Mechanism					Context	Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	ECA Rules		Mobile Computing	Embedded Systems	Smart House	Multimeedia
Bastia and al.	–	X	–	–	–	X	–	X	X	X	–	–	–
Bencomo and al.	–	X	–	–	–	X	–	X	X	–	–	–	–
David and Ledoux	–	X	–	–	X	–	–	–	X	–	–	–	–
Pessemier and al.	–	–	–	–	–	–	–	–	–	–	–	–	–
Zhang and Cheng	X	–		–	–	X	–	–	–	X	–	–	–
Dinkelaker and al.	–	–		–	X	–	X	–	–	–	–	–	–
Trinidad and al.	–	X	–	–	–	–	X	–	–	–	X	–	X

3.3 Mixed adaptation approaches

In this category, we analyze a third group of approaches that propose mixed solutions. Such approaches include some of the characteristics we have found separatedly in the two previous groups, but combined in order to provide support for adaptations to be performed at design time and at runtime.

Adaptation criteria

Since this category is a combination of the two previous categories, we use the same criteria specified for the previous groups.

In this category, we find the work by Bastida et al. [4]. The authors introduce an approach for context-aware service composition. They propose a methodology of six processes aimed at defining an executable model composed of several services with a particular workflow, which represents a set of variants chosen for several variation points. Afterwards, the composition can take place at runtime based on ECA rules, in order to connect to new services. Regarding the mechanisms for runtime, the authors use context information defined as a dynamic property that may depend on an underlying protocol. The property is used in a predicate which is expressed in their particular ADL. This ADL associates a programmed reconfiguration action to the property. This leads to context-based reconfigurations that are triggered through a change in the dynamic property.

Apel et al. [1] introduce the notion of Aspectual Feature Modules. They aim at combining feature oriented programming (FOP) and AOP to implement feature models when required. Their approach uses classic feature modules for non cross-cutting concerns and AOP for special cases. Although not specified, their approach could eventually use dynamic AOP, making adaptation possible at both design and runtime. The scope of the adaptations in this approach is the source code where aspects are woven. Regarding the mechanisms, Apel et al. base their approach on the combination of variability to build families of similar products, and AOP for the cross-cutting concerns.

Lundesgaard and al. [17] propose an approach formed by two parts: an MDA transformation chain for building adaptive applications, and a middleware system to make decisions about adaptation based on Quality of Service (QoS) information. At runtime, the model is causally connected with the application it represents. The adaptation takes place by choosing the right configuration by modifying the application model. Then, the application absorbes the changes of the model. No details are given as to how this last process actually takes place. The approach uses QuAMobile, a context and QoS-aware middleware that identifies and chooses the best variant configuration for the current context and available resources. Such middleware works as a set of plugins that can be plugged in and out. In particular there is one plugin called Context Manager in charge of managing context information.

In [18], the authors present K@RT, a framework for dynamic product lines based on aspects and models. They use models at runtime for dynamic variability and deal with the combinatorial explosion. In [36] a close related work presents the strategy for dynamic adaptation. Models are kept at runtime as part of the application being executed, then, the target configuration is calculated for the current conditions of the executing environment. Having both models, current and target model, a difference is computed, and from this difference, a reconfiguration script is generated that takes the current configuration to the target configuration. Also from the same authors, Morin et. al. [19] present an evolved version of this approach when aspects are formed by advice, pointcut, and

weaving directives. They use dynamic reconfigurations to modify the model of the application, the architecture of the application itself, and the code generated from the model. With respect to the scope of these approaches, the main element of the adaptation is the model. However, using the models as starting point, they also introduce strategies for source code generation. Regarding context awareness, in [36] authors define an adaptation model which captures all the information about the dynamic variability and adaptation of their adaptive system. Among the elements that conform to such a model, they include a context model which is a minimal representation of the environment used to define adaptation rules. In a similar way, in [19] authors define aspect models which may include a context that they describe as a slice of the environment. No further details are given concerning the context management or the frameworks used for context aggregation.

Phung-Khac [24] proposes the adaptive medium approach for developing adaptive distributed applications. His approach proposes a development process where business logic is separated from the adaptation aspects of the applications. The business logic is refined in different configurations that are treated as different members of the system family. Like this, the adaptive medium approach extends the feature modeling method. On the other side, the adaptation aspects define architectural models are and in charge of adapting the business logic dynamically. In his approach, the desired adaptive application is specified at a high abstraction level and then is refined towards the implementation level. For the runtime adaptation, the approach uses FRACTAL software components. The architectural models generated by the refinement process are embedded into the adaptation control to perform the reconfigurations. Finally, since the applications are component-based, we consider that their adaptation scope corresponds to the architecture of the applications that get modified during the reconfigurations, and additionally, to the generated code that is obtained from their refinement process.

Finally, the project ECaesarJ [21] represents an approach to have design and runtime adapatations based on the programing language. ECaesarJ is an aspect oriented language that is based in its predecessor CaesarJ [3]. The language aims at facing the challenges of feature decomposition. To do so, it improves modularity of object oriented programming languages by providing extension and composition mechanisms. At the core of ECaesarJ there is the concept of virtual classes. A virtual class can be redefined in subclasses by adding new methods fields and inheritance relationships. This allows, for example, to define features as extensions of other features. For the composition of features, ECaesarJ supports mixings. It represents a form of multiple inheritances when all inherited declarations of virtual classes with the same name are composed. ECaesarJ also offers support for events and state machines. The events are used to represent explicitly behavioral abstractions. An event is composed of a source and destinations. Examples of such events include implicit join points of the source code for example method calls, attribute value changes, but can also be explicitly defined by the programmer. The state machines are supported in ECaesarJ to make it possible to organize the event handling. Because it is based on CaesarJ, the weaving of aspects takes place through several deployment process which include design time and runtime deployments. At design, the aspects are woven in a similar way as in any AspectJ-based approach. At runtime, the aspect which is defined as in java, can be instantiated at any moment. However for the weaving to actually take place the aspects deployment act as a wrapper that intercepts the calls to the business objects to enrich them with the advice code. Since the language is basically focused on java code, their mechanisms include aspect oriented techniques as well as direct code manipulation. The scope of the adaptation achievable with ECaesarJ is the business code itself that gets modularized and composed by the aspects defined with the ECaesarJ language.

Summary of mixed approaches

Table 5 summarizes the approaches of the third category. The criteria are the same that we have used for the runtime approaches. In the same way as before, a check mark (X) indicates if the approach proposes solutions or deals with the different criteria, and a dash (–) in the opposite case.

This third category of approaches is the most interesting one for the development of DSPLs. The approaches in this category offer support for both design and runtime adaptations. Some of them use variability management and context information as well as models at runtime, reflective platforms, or dynamic aspects that allow them to have both source code manipulation processes for the design adaptations and dynamic reconfigurations for the runtime adaptations. Some of them also use variability management for modularizing and defining adaptations, and context information to define concrete events at runtime for adaptations. There are other approaches based on programming languages that focus on modularity and propose constructs tailored for feature decomposition.

However, for a complete DSPL, we consider that there are still two main issues missing. First

of all, the approaches in this group do not offer a complete development life cycle from feature modeling and configuration to runtime adaptations. This means that design and runtime adaptations are realized through completely independent process that does not have many elements in common. Moreover, to the best of our knowledge, none of the approaches offers a unified representation of adaptations. Assets used for building applications are defined and treated in a different manner than assets used to achieve reconfigurations dynamically.

Table 5: Summary of mixed adaptation approaches.

Reference	Scope			Mechanism					Context	Domain			
	Model	Architecture	Code	AOM	AOP	MDE	SPL Variability	ECA Rules		Mobile Computing	Embedded Systems	Smart House	Multimedia
Bastida and al.	–	X	–	–	–	–	–	X	X	X	–	–	–
Lundesgaard and al.	X	–	–	X	–	X	–	–	X	–	–	–	X
Morin, Barais and Jezequel	X	X	–	X	–	X	–	–	X	–	–	–	–
Morin and al.	X	X	–	X	–	X	–	–	X	X	–	–	–
Appel and al.	–	–	X	–	X	–	X	–	–	–	–	–	–
Phung-Khac	–	X	X	–	–	X	X	–	–	–	–	–	–
Ecaesar Project	–	–	X	–	X	–	–	–	X	–	–	–	–

3.4 Summary

We can now summarize the results of the approaches reviewed with regard to the challenges they face and the strengths and weaknesses of each group. In table 6 we summarize the findings of each of the three categories previously discussed. We have three main criteria. First, we illustrate if the category support design time adaptations, and if it uses variability management for the derivation process. Second, we illustrate if the approach support runtime adaptations, and if it uses context information for the decision making. Finally, we add a last criteria indicating if the approach offers a unified representation of design and runtime adaptations. In the next section, we further discuss the unification and revisit the challenges for a complete DSPL approach that successfully manages design time and runtime adaptations.

Table 6: Synthesis of approaches for DSPL.

Approach	Design Adaptation	Variability	Runtime Adaptation	Context Awareness	Unified Representation
Focus on Design and Product Derivation	Yes	Yes	No	No	No
Focus on Runtime Reconfigurations	No	No	Yes	Yes/No	No
Mixed Approaches	Yes	Yes/No	Yes	Yes/No	No

4. RESEARCH GOALS

From the results of Table 6, we have concluded that the main missing issue relates to the lack of unification between adaptations at design time and adaptations at runtime. For each one, dedicated technologies are used to **specify** and **realize** the adaptations as it has been presented in all of the categories previously reviewed. In addition, both adaptation processes can be understood as the modification of the product being derived by adding and/or removing a certain group of features. It would be desirable to have a unified representation of this modification, so that it can be used at design as well as at runtime.

4.1 The Need for Unification

We claim that design and runtime are similar in their definition and their using process, but not in their implementation. Hence, in order to define a complete approach for DSPL, we need a unified representation of adaptations that combines design and runtime in a coherent development process. Design adaptations are often considered to be of completely different nature than runtime adaptations. Design adaptations are motivated by design goals whereas runtime adaptations are motivated by changes of the software environment. Moreover, design adapftations are considered as permanent adaptations that cannot be rolled back whereas runtime adaptations are considered as impermanent. However, whatever the technology and whatever the phase, a software adaptation is always initiated by a particular motivation and is always realized through modifications of some software artifacts. Therefore, from a specification point of view, design and runtime adaptations are not that different. We argue that a single unified language should be provided to specify both of them. Based on this language, a platform should be realized to derive the software products at design time and runtime transparently.

Having only one unified language to specify design and runtime adaptations offers several advantages. First, it formalizes similarities and differences that exist between the two kinds of adaptation. Second, it may serve as a basis to transform design adaptations into runtime ones and vice versa. Transforming design adaptations into runtime one

allows one to delay the realization of some design adaptations to the runtime phase. Transforming runtime adaptation into design one prevents the realization of adaptation mechanisms that have been defined regarding specific environment state that may not arise at runtime. Third, unifying the specification of modifications done by both aspects is the first step to compute analysis such as dependency analysis between aspects.

Having a platform that derives the software products at design time and at runtime transparently offers several advantages. First, it supports the whole life cycle from the initial creation of the product (driven by feature selection) to its dynamic adaptation (driven by changes of its environment). Second, it establishes the link between the motivations (feature selection or changes of the product environment) and the adaptations of the software artifacts. Third, it can be used as a way to achieve flexibility in the tradeoff between development cycles that are fully design oriented (without any runtime adaptation), and development cycles that are fully runtime oriented (without any feature selection).

4.2 Challenges for DSPL

Having the unification in mind, we can now precise the goals of our approach. We investigate on software engineering techniques for developing and adapting software. Our main goal is to implement dynamic software product lines through the **unification of software adaptations** that allows developers to define and implement adaptations both at design time and at runtime.

Let us now refine the goals of our approach and group them properly according to the classification we have introduced for the reviewed works, namely: design and variability, runtime and context awareness, and unification.

Design and Variability

First of all a DSPL needs a design adaptation phase that allows developers to build products through automated processes. These processes need to take into account the variability of the product family as well as further analysis and management for different product configurations. We identify the following challenges for a design adaptation process:

- ***Automated Development Process*** An SPL exploits commonalities among a set of software products in order to identify and build reusable assets that can be used to derive new products reducing the effort and time invested when building several products. A DSPL needs to automate the development process of adaptable software.
- ***Variability and Correctness*** It is important that products are not only easier to develop, but also that their correctness remains guaranteed. When composing multiple parts to form a single software product, it is possible that two or more of those parts have conflicts regarding the elements where they are going to be composed and the requirements for the composition to take place. In other words, implicit dependencies may exist between different artifacts which may lead to composition problems. A design time adaptation has to exploit variability management in order to define a development process that analyses such dependencies and prevents incorrect products from being derived.
- ***Guarantee Platform Independence*** It is also desirable that business concepts about the products to be derived are separated from the details of the underlying platform. The derivation process in the DSPL has to postpone as much as possible the decisions about platform and implementation. This enables developers to define multiple targets and offer support for different platforms.

Runtime and Context awareness

Second, the DSPL has to deal with runtime reconfigurations. For this process, context information has to be used to decide about the adaptation. At the same time, the reconfiguration has to respect the constraints defined during the design with the variability. We identify the following challenges that have to be faced to realize a process of adaptation at runtime:

- ***Define and adaptation cycle with well-assigned responsibilities*** An equivalent process of product derivation as the one defined for the design adaptations has to be defined. It has to take as input the running product and its configuration, and has to return a new adapted version of the product. A complete adaptation loop has to be established, by differentiating different sub-process in charge of: monitoring the context information, analyzing and deciding about possible adaptations, and finally, executing the adaptation on the software product.
- ***Use context information for the decision making*** A fundamental issue in adaptive software development is the management of context events, and its manipulation in order to modify products dynamically. The DSPL has to take context information into account to decide the appropriate configuration at the right moment when adaptations take place in order to offer a better experience to the final users.
- ***Extend the concept of feature at runtime*** Since products in the SPL are described as a set of selected features, an important challenge to achieve dynamic product derivation is to define a way to maintain, and update, the state of a product in terms of the features it is supporting at a given moment of its execution. Furthermore, this information has to be used, in the same way as in the design time adaptations, to guarantee that the product will respect the constraints of the product family after the adaptation has taken place.

A unified representation and management of design time and runtime adaptations

Finally, to provide the unification of adaptations at design time and at runtime, the DSPL has to define a language and use an underlying platform that allows definition of adaptations independently from the moment when they take place. This would allow developers to define only once any adaptation, and use it independently at design for building a product, or at runtime for adapting an existing product.

Additional Properties for a DSPL

In addition to the challenges for the design phase, the runtime phase, and the unification of adaptations, we consider that any framework for developing DSPLs has to consider the following properties.

- ***Extensibility*** Extensibility is a property of highly importance in any SPL. Since requirements are evolving constantly, it is desirable that SPLs can be extended or adapted to support the derivation of new products, different execution platforms, or new functionalities required by different stakeholders. This fact is reinforced by [32] when authors define domain and application engineering processes. These two processes are usually implemented in iterative developments cycles. This practice intends to exploit the complementary nature of each process. For example, during the application engineering, it is possible to identify new requirements. Those requirements can be supported by the existing DSPL in a new iteration, by creating their corresponding assets. This allows the SPL to evolve and extend its scope over time. DSPLs are no different than traditional SPLs regarding the need for extensibility. It is important to provide the mechanisms to extend the scope of the product family and support new functionalities regardless of the derivation time.
- ***Scalability*** In any SPL, one of the biggest challenges refers to the management of the combinatorial explosion of product configurations. The size of a product family can grow exponentially when features are added. Larger product families represent a challenge in terms of scalability and performance. In an approach for DSPLs, it is necessary to consider this issue because part of the management of the product family is postponed at the execution of the different products. Calculations over larger product families performed at runtime may have an impact on the adaptation and the overall performance of the products.
- ***Runtime History*** When a product is adapted at runtime, it changes its configuration. If such changes include the deletion of several parts of the product, then such modifications have to remain available. Like these, products can be able to go back to a previous state before one or several adaptations have taken place. A DSPL has to take into account this kind of changes.
- ***Usability*** Finally, another property for an approach in DSPLs is usability. By usability, we mean the difficulty encountered by newcomers when starting to use a new framework for DSPLs. This can be related with the changes in the development process, especially when there are automated parts that are mixed with manual parts; and also, it can be related with the use of new languages for modeling the different assets that are combined to produce the software products. We consider that a framework for DSPL has to remain usable, for the automation to have a positive impact on the effort and time invested when building individual software products, regardless of the changes on the development process introduced by the framework.

4 CONCLUSION

In this paper we have surveyed several approaches in literature that are close related to the main contributions of our work. We have reviewed an important number of research works that use a variety of technologies (i.e. MDE, SPL, AOSD, CBSE) in order to build software and/or adapt it at runtime. We made a classification of the approaches surveyed. This classification has been used to concretize the main objectives of our approach which are variability management, automated development and correctness, platform independence, and derivation at runtime. We conclude then this part dedicated to the study and analysis of the background and state of the art.

REFERENCES

[1] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. IEEE Transactions on Software Engineering (TSE), 34(2):162–180, 2008.

[2] Hugo Arboleda, Andrés Romero, Rubby Casallas, and Jean-Claude Royer. Product derivation in a model-driven software product line using decision models. In Antonio Brogi, João Araújo, and Raquel Anaya, editors, CIbSE, pages 59–72, 2009.

[3] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In ECSA, pages 1–17. LNCS, 2005.

[4] Leire Bastida, Francisco Javier Nieto, and Roberto Tola. Context-aware service composition: a methodology and a case study. In SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments, pages 19–24, New York, NY, USA, 2008. ACM.

[5] Peter J. Brown. The stick-e document: a framework for creating context-aware applications. In A. Brown, A. Brüggemenn-Klein, and A. Feng, editors, Special Issue: Proceedings of the Sixth International Conference on Electronic Publishing, Document Manipulation and Typography, Palo Alto, volume 8, pages 259–272, John Wiley and Sons, June 1996.

[6] Nelly Bencomo, Pete Sawyer, Gordon Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In 2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland,, 2008.

[7] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, GPCE, volume 3676 of Lecture Notes in Computer Science, pages 422–437. Springer, 2005.

[8] Siobhán Clarke. Extending standard uml with model composition semantics. Sci. Comput. Program., 44(1):71–100, 2002.

[9] Pierre-Charles David and Thomas Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In Welf Löwe and Mario Südholt, editors, Software Composition, volume 4089 of Lecture Notes in Computer Science, pages 82–97. Springer, 2006.

[10] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A Dynamic Software Product-Line Approach using Aspect Models at Runtime. In Fifth Domain-Specific Aspect Languages Workshop, 2010.

[11] Tijs Van der Storm. Generic feature-based software composition. In Markus Lumpe and Wim Vanderperren, editors, Proc. of the 6th International Symposium on Software Composition (SC'2007) - Revised Selected Papers, volume 4829 of Lecture Notes in Computer Science, pages 66–80. Springer, 2007.

[12] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. Computer, 41(4):93–95, 2008.

[13] IBM. An architectural blueprint for autonomic computing. white paper, June 2006. White Paper.

[14] Jörg Kienzle,Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In AOSD '09: Proceedings of the 8th ACM international conference on Aspectoriented software development, pages 87–98, New York, NY, USA, 2009. ACM.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, Proc. ECOOP 2001, LNCS 2072, pages 327–353, Berlin, June 2001. Springer-Verlag.

[16] Kwanwoo Lee, Goetz Botterweck, and Steffen Thiel. Aspectual separation of feature dependencies for flexible feature composition. In Proc. of the 33rd Annual IEEE International Computer Software and Applications Conference, pages 45–52. IEEE CS, 2009.

[17] Sten A. Lundesgaard, Arnor Solberg, Jon Oldevik, Robert B. France, Jan Øyvind Aagedal, and Frank Eliassen. Construction and execution of adaptable applications using an aspect-oriented and model driven approach. In Jadwiga Indulska and Kerry Raymond, editors, DAIS, volume 4531 of Lecture Notes in Computer Science, pages 76–89. Springer, 2007.

[18] Brice Morin, Olivier Barais, and Jean-Marc Jezequel. K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France, oct 2008.

[19] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, May 2009.

[20] Philip K. McKinley, Seyed M. Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report, Department of Computer Science and Engineering, Michigan State University, 2004.

[21] Angel Núñez, Jacques Noyé, and Vaidas Gasi ¯unas. Declarative definition of contexts with polymorphic events. In International Workshop on Context-Oriented Programming, COP '09, pages 2:1–2:6, New York, NY, USA, 2009. ACM.

[22] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 14(3):54–62, 1999.

[23] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In ICSE Companion, pages 899–910, 2008.

[24] An Phung-Khac. A Model-driven Feature-based Approach to Runtime Adaptation of Distributed Software Architectures. PhD thesis, Université Européenne de Bretagne., November 2010.

[25] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In 12th International Software Product Line Conference (SPLC 2008), pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[26] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Thierry Coupaye. A component-based and aspect-oriented model for software evolution. IJCAT, 31(1/2):94–105, 2008.

[27] Raghu Y. Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspectoriented design class models. T. Aspect-Oriented Software Development I, 3880:75–105, 2006.

[28] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. Engineering languages for specifying product-derivation processes in software product lines. In Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers, pages 188–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Pablo Trinidad, Antonio Ruiz Cortés, and Joaqu´in Pe na. Mapping Feature Models onto

Component Models to Build Dynamic Software Product Lines. In International Workshop on Dynamic Software Product Line, 2007.

[30] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In SPLC '07: Proceedings of the 11th International Software Product Line Conference, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society

[31] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations, pages 152–167, Berlin, Heidelberg, 2008. Springer- Verlag.

[32] K. Pohl, G. B¨ockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, New York, NY, USA, 2005.

[33] Ji Zhang and Betty H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In ICSE '06: Proceedings of the 28th International Conference on Software Engineering, pages 371–380, New York, NY, USA, 2006. ACM Press.

[34] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum.-Comput. Interact., 16(2):97–166, 2001.

[35] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, pages 1–7, New York, NY, USA, 2005. ACM.

[36] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon S. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, volume 5301 of Lecture Notes in Computer Science, pages 782–796. Springer, 2008.

[37] David L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053–1058, December 1972.

[38] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I, pages 135–173, 2006.