An Architecture Description Language for Dynamic Service-Oriented Product Lines

Seza Adjoyan  
UMR CNRS 5506 / LIRMM  
University of Montpellier  
Montpellier, FRANCE  
adjoyan@lirmm.fr

Abdelhak Seriai  
UMR CNRS 5506 / LIRMM  
University of Montpellier  
Montpellier, FRANCE  
seriai@lirmm.fr

Abstract—Reconciling Software Product Lines (SPL) and Service Oriented Architecture (SOA) allows modeling and implementing systems that systematically adapt their behavior in respond to surrounding context changes. Both approaches are complementary with regard to the variability and the dynamicity properties. Architecture Description Language (ADL), on the other hand, is recognized as an important element in the description and analysis of software properties. Different ADLs have been proposed in SOA or in SPL domains. Nevertheless, none of these ADLs allows describing variability and dynamicity features together in the context of service-oriented dynamic product lines. In this sense, our work attempts to describe the changing architecture of Dynamic Service-Oriented Product Lines (DSOPL). We propose an ADL that allows describing three types of information: architecture’s structural elements, variability elements and system’s configuration. Furthermore, we introduce context elements on which service reconfiguration is based.

Keywords—Architecture Description Language (ADL); Service-Oriented Architecture (SOA); Software Product Lines (SPL); dynamicity; variability; software architecture; Dynamic Service-Oriented Product Lines (DSOPL)

I. INTRODUCTION

Software Product Lines (SPL) and Service Oriented Architecture (SOA) have a common goal from a software development point of view, increase the reusability of existing assets rather than rebuilding new systems from scratch. SPL, on the one hand, allows the development of a family of products that share some common set of core assets [1], [2], [3]. Variability has always been a first concern in SPL studies [16]. According to [4], variability is the ability of a software artifact to quickly change and adapt for a specific context in a preplanned manner. SOA, on the other hand, is a special kind of software architecture, where the main architectural elements are coarse grained and loosely coupled services that are dynamically composable and inter-operable [5]. Being able to modify the architecture of a running system at such a high level of abstraction renders the system highly extensible, customizable and powerful [6].

Variability and dynamicity are core properties to develop complex adaptable software systems such as telecommunication, pervasive, crisis management, surveillance and security systems. In such systems, due to environment changes, a dynamic re-configuration should be carried out without having to re-deploy the whole system. Combining SOA and SPL constitutes the answer to this need [7]. SOA offers, through its encapsulation property and its explicit interfaces, a solution for achieving dynamic product lines. SPL offers, via variability modeling, analysis and design of changing points in service-oriented architectures.

Architecture Description Language (ADL) is a formalism that allows the specification of system’s conceptual architecture [8]. It enables architects to describe and validate systems against stakeholders’ requirements from one side, and ease the development and implementation process of complex systems, from another side. It often has a graphical representation or plain text syntax. Conventional ADLs support only static architecture description [6]. Some ADLs provide special formalism for SOA to describe service dynamicity or for SPL to describe variability. Unfortunately, no ADL supports the crosscutting SOA and SPL concepts.

To overcome this limitation, we propose an XML-based ADL that allows describing the architecture of a Dynamic Service-Oriented Product Line (DSOPL). It describes the four following elements: (i) the structural elements of a family of software products (i.e. services and connections), (ii) an architectural variability model (i.e. variability points and alternatives), (iii) context information, in addition to (iv) an architectural configuration model (i.e. reconfiguration rules based on context and variability). We choose to use XML as a description language to facilitate understandability and analysis of the described architecture. In addition, XML-based description facilitates tool-support design and interoperability.

The remainder of this paper is organized as follows: In section 2, we discuss related works regarding variability and dynamicity properties. In section 3, we characterize our proposed DSOPL-ADL’s elements and demonstrate their utility through a running example. Finally, in section 4, we summarize our contribution and provide directions for future research.

II. RELATED WORK

A. ADLs specifying dynamic properties

A software architecture can be classified in terms of its capability of evolution into two categories: static or dynamic. A static architecture reflects the static structure of software and is completely specified at design time [6], whereas in dynamic architecture, system may evolve after its compilation [1]. In this type of architecture, in addition to specifying the
system in terms of components, connectors and configurations, it should also specify how these components and connectors are evolved or reconfigured at runtime. This evolution of architecture at runtime may happen under several forms: adding/removing composing elements, reconfiguring architecture (modifying connections), or upgrading existing composing elements (substitution of composing elements).

ADLs are used to describe the properties of static or dynamic software architecture. Among existing ADLs in the literature, only few of them support dynamic reconfiguration such as C2 [9], Darwin [10], π-ADL [11], Rapide [12], ACME/Plastik [13] and Dynamic Wright [14]. In order to describe a specific configuration in [13], the expression “on [condition] do [operations]” is used to toggle between different choices at runtime. To replace an instance of component at runtime, **detach** and **attachments** statements are used in operations part to respectively unlink and link components. In π-ADL [11], the architecture is considered dynamic since third party services can be discovered and bound to service broker at runtime.

II. Dynamic Service Oriented Product Line (DSOPL) ADL

A. Illustrative example

We will use throughout the paper an illustrative example to exemplify concepts related to our proposed approach. This example is about a simplified online sales scenario between four actors: customer, retailer, warehouse and shipment services, as modeled in Fig. 1. The customer accesses retailer’s website, browses the catalog, selects some items and commands an order. The retailer fulfills customer’s order request and inquires the warehouse to prepare all items of the order. Once the order is prepared, the shipping service handles the delivery of items to the customer.

![Illustrative example: online sales scenario architecture](image)

B. The DSOPL-ADL structure

In order to describe the runtime variability of a Dynamic Service-Oriented Product Line (DSOPL) system at architecture level, we propose an XML-based ADL. This ADL is structured in four sections, as summarized in the schema of Fig. 2:

1. Structural element description: defines all types of the abstract structural entities of the system (services, interfaces, operations).
2. Variability description: here, variation points are defined and also all alternative services of each variation point with the constraints related to each alternative.
3. Context description: variability and configuration descriptions are based on information about context. Thus, information about context elements is described in a specific section of the ADL.
4. Configuration description: here, the rules used to create concrete services and connections are specified to describe how to configure (generate) concrete architectures based on structural, variability and context elements.

![Modular DSOPL-ADL](image)
Our approach implicitly separates the four aforementioned architectural concerns from each other. The modular description of architecture in four sections, each of them specifying one type of architectural description has the following advantages:

1) It facilitates the modification and re-utilization of each of the four sections of ADL.
2) It allows the description and analysis of the architecture by separating the four concerns (structure, variability, context and configuration).
3) It allows controlling the traceability links of each type of information among several abstraction levels. For example, the variability described in feature model at requirement level is translated at architecture level through its variability section.

C. Structural elements description

A service is an encapsulated and self-contained unit. It interacts with other services through interfaces. The system itself is a composite service and is hierarchically decomposed into finer-grained services. Leaf services are called atomic services. All other services in the hierarchical tree are considered composite. A composite service doesn’t execute or implement any functionality by itself, but it delegates this task to one of its composing services. Each composite service is described as sub-architecture.

Each service has a number of provided interfaces and may require a number of required interfaces. Interfaces define a collection of methods or operations that are supported by the service. Since services are developed independently from their future exploiting systems, they should have solid and well-defined interfaces that describe their functionalities and operations. Interfaces are two types, either provided interface or required interface. Provided interface of a service is an interface that the service realizes, whereas required interface is an interface that the service needs in order to operate. Services communicate to each other through provides/ consumes relationship via their provided/ required interfaces. An interface has a set of operations.

The structural description of a service reflects this service meta-model. A service is described based on the following architectural attributes, as shown in Fig. 3: (1) Every service has a name specified by service_name attribute. (2) It has a textual_description that explains in plain text the main functionalities of the service, its inputs and expected outputs. (3) is_atomic has a Boolean value to indicate whether the service is atomic or composite. Fig. 4 shows the structural section description of the architecture related to our illustrative example.

```
<structural_description>
  <service name="supply_chain_management_service" ... is_atomic="true">
    <interfaces>
      <interface name="i_order" role="provides">
        <operation name="submit_order_request" ...
      </interface>
    </interfaces>
  </service>
  ...
</structural_description>
```

D. Variability description

Variability in SPL architecture refers to the ability of making changes to system’s architecture. We distinct three types of variability:

1) Service variability

It represents binding an alternative service that satisfies pre-conditioned constraints on runtime. Back to our sales example, there are two alternatives of shipment; either a relay point shipment or home delivery shipment, as shown in Fig. 5.

The decision of which alternative to choose is taken automatically at runtime depending on customer’s selection in addition to other environmental conditions such as the existence of a relay point service in customer’s city, as depicted in Fig. 9.

2) Variability of connection

It may exist several alternative connections between services. The selection of the appropriate connection is done
automatically at runtime according to constraints’ satisfaction. For example, the customer service in Fig. 6 can access the retailer service and thus command an order via two different connections; either a connection for a regular customer or a connection for a VIP customer which normally has some extra privileges. The variation_point “customer_variation_point” in Fig. 9 is an example of variability of connection.

3) Variability of composition

This type of variability concerns replacing not only a service or a connection, but replacing a set of interconnected services by another set of interconnected services within a composite architecture. Fig. 7 illustrates another alternative composition of supply_chain_management_service than the one in Fig. 1. Here, in addition to the roles of retailer and warehouse services, the manufacturer service realizes requested items and returns them to the warehouse service.

Figure 7. Example of composition variability in sales scenario

The meta-model of variability description is given in Fig. 8. We specify in this section the different variation points that exist in the system at architectural level. A variation point specifies the part of the architecture that can be variable. Each variation point has the following attributes: (1) variation_name indicating its unique name, (2) variation_type that specifies the type of this variation. Possible values of variation_type are either service, connection or composition. (3) variation_time specifies whether this variation may occur at compile-time (i.e. before runtime) or at runtime. Contrary to traditional SPL approaches where variability is clearly and completely specified at design time [21], variation_time attribute is important in SOA systems, where selection of an alternative during runtime is totally possible. However, some variation points could be specified at compile-time. This reduces the overhead of loading the entire configuration at runtime. Each variation point has several alternatives, which are possible elements to fill the selected variation point. Each alternative has a unique name alternative_name and an order of priority. This attribute helps the system automatically determine which architectural element is chosen in case there is more than one valid configuration at a given time. The alternative with the highest priority priority=“1” is the preferred one in a variation point. Each alternative has a set of constraints, in forms of pre-conditions and post-conditions, to operate properly. Pre-conditions specify a group of conditions that should be satisfied before executing the selected alternative (i.e. alternative can be selected, only if all constraints of pre-conditions are satisfied). Post-condition represents desirable outcomes when process is completed successfully. Pre and Post-conditions are the equivalent of crosscutting “requires”, “excludes” and “and” constraints in Feature Model FM in SPL paradigm. For example, the pre-condition that states that in order to choose the alternative “relay_point_delivery_alternative”, “relaying_point_service_in_city” should be available (see Fig. 9), this statement is equivalent in FM to a “requires” constraint from “relay_point_delivery” feature to “relaying_point_in_city” feature. On the contrary, condition “unavailable” is equivalent to “excludes” constraint in FM.

Figure 8. Variability description meta-model of DSOPL-ADL

Figure 9. Variability description of sales scenario
E. Context description

Architecture reconfiguration is based on context changes. The context consists of any element that influences the behavior and/or the structure of the architecture. It can be related either to system’s environment (e.g. escalator state in the case of crisis management software), evaluated quality of service (e.g. time to response to a query), hardware architecture changes (e.g. server failure), etc. Thus, context element needs to be described in a dynamic ADL. We include these context elements as part of the architecture description to allow context-aware configurations (i.e. autonomous run-time adaptation according to context changes). A context element could capture raw data from a single information source such as a GPS locator that locates customer’s current location to search for a nearby relay point for the shipping service in our sales example. In this case, context element is considered as a primitive context. In some other cases, a single information source could not be sufficient to take decisions; in that case, different atomic information sources’ values are collected, combined and analyzed in order to give sufficient and more accurate information about the context value. We call this context as composite context. We can consider the weather forecast example, where the weather is considered hot when both temperature and humidity sensors exceed a certain threshold.

A simplified meta-model of context is illustrated in Fig. 10. Any context element has a unique name and a context_type to indicate to which family of contexts it belongs (e.g. contexts related to environment, user preferences, etc.). Context element also has values_type that indicates the type of its values, either primitive types such as integer, double, etc. or user-defined types. In Fig. 11, we show two primitive context descriptions from our sales scenario.

F. Configuration description

In traditional architectures, where environment is considered stable, services are selected and composed at design time. In contrast, in dynamic environment, parts of the software can be instantiated or evolved at runtime. Therefore, we need to maintain, in addition to structural information, architectural information of the running system. The configuration section of DSOPL-ADL allows describing all the configuration rules to generate valid architectures. A valid architecture is a concrete architecture whose services and connections comply with configuration rules.

The configuration description section of DSOPL-ADL has an initialization sub-section, where all static elements (services and connections) in addition to alternatives, whose variation_time="compile_time", are instantiated. The connection part has two references to two different service interfaces, the one that calls the information consumer_interface and the one that provides the information provider_interface. The configuration description also has a dynamic_configuration sub-section where architectural configurations are triggered based on runtime context conditions. In other words, a concrete architecture is selected through two consecutive execution levels: (1) static bind where core services are selected and bound then (2) late-binding where remaining services and variation points are bound.

In initialization sub-section, we first bind static services to the configuration in addition to their connections. In dynamic_configuration sub-section, we integrate selected instances of services by observing context changes that are specified in the condition part of the configuration rule. Fig. 12 illustrates the architectural configuration meta-model. Any partial_configuration has a name and an attribute called priority of type integer, which determines which configuration to choose in case more than one partial_configuration satisfies current conditions. At that time, the one with the higher priority is privileged. Each partial_configuration is composed of two parts; condition part and dynamic_action part. In the condition part, we specify conditions that are driven by context elements. In the dynamic_action part, we specify all dynamic activities that will be realized. Every action concerns an architectural element which can either be a service or a connection. Action_type defines the type of change that will apply on the selected element. Its values are limited to bind, unbind, activate or deactivate concerned elements.
In our illustrative example, customer and supply chain management services are instantiated at design time, as depicted in Fig. 13, whereas the relay point shipping service or home delivery shipping service are instantiated dynamically depending on environment’s conditions.

<configuration_description>
<initialization>
<services>
<deployable_service_instance service_instance_name="home_delivery_shipping_service_instance" ...>
</deployable_service_instance>
<deployable_service_instance service_instance_name="supply_chain_management_service_instance" ...>
</deployable_service_instance>
</services>
<context_element name="shipping" />
<condition>
<priority="2">
<expression operator="equals" value="home">
<expression_operator>
<expression>
<architecture_element element_type="service" name="home_delivery_shipping_service_instance" action_type="activate"/>
</expression>
</expression_operator>
</expression>
</condition>
<dynamic_configuration>
<initialization>
<services>
<connect provider_interface="i_goods_response">
<connection consumer_interface="i_goods_request" connected -->
</connection>
</services>
<services>
<initialize>
<configuration>
<context_element name="shipping" />
<condition>
<priority="2">
<expression operator="equals" value="home">
<expression_operator>
<expression>
<architecture_element element_type="service" name="home_delivery_shipping_service_instance" action_type="bind"/>
</expression>
</expression_operator>
</expression>
</condition>