Reconfigurable Service-Based Architecture Based on Variability Description

Seza Adjoyan, Abdelhak-Djamel Seriai

Reconfigurable Service-Based Architecture Based on Variability Description

Seza Adjoyan
LIRMM, CNRS, University of Montpellier
Montpellier, France
adjoyan@lirmm.fr

Abdelhak-Djamel Seriai
LIRMM, CNRS, University of Montpellier
Montpellier, France
seriai@lirmm.fr

ABSTRACT

Self-adaptive systems evolve during system's execution against changes in operating environment. Such evolution and reconfiguration can be specified at architecture level using a syntactical expressive language such as Architecture Description Languages (ADLs). Variability modeling is an excellent instrument to model variations of software artifacts and their behavior within a self-adaptive system. However, existing ADLs that support dynamic reconfiguration do not explicitly model variation points on which the reconfiguration is based. This constitutes a barrier for a flexible management of reconfiguration at architecture level as well as traceability issues between a dynamic description given at architectural level and its counterpart at other abstraction levels. In this paper, we propose a variant-rich service-oriented ADL that enables system to reconfigure itself at runtime in response to a context change. To this end, our modular ADL, called Dynamic Service-Oriented Product Lines Architecture Description Language (DSOPL-ADL), specifies dynamic reconfigurations at architecture level besides specifying structural, variability and context information. Among several specified variable configurations at architecture level, one concrete configuration is generated at runtime triggered by a context value. Furthermore, an implementation code can be automatically generated from the architectural description.

CCS Concepts

•**Software and its engineering** → **Software architectures;** Software development process management;

Keywords

Architecture Description Language (ADL); variability; dynamic reconfiguration; architecture-centric reconfiguration; self-adaptive service-based systems; context-aware

1. INTRODUCTION

SAC 2017, April 03-07, 2017, Marrakech, Morocco

DOI: http://dx.doi.org/10.1145/3019612.3019767

It may happen that software architecture needs to evolve after its deployment [8]. Such an architecture is considered dynamic, since it can modify itself and adopt modifications during system's execution [5]. This is a required property in systems where stopping the execution to make modifications on it might cause dramatic effects. Relying on dynamic architecture can be useful in self-configurable systems. Self-configurable (or self-adaptive) systems aim to adapt their various artifacts autonomously as well as change their configuration dynamically in response to changes in the operating environment (i.e. context changes) without human interaction [7], [12], [10]. Dynamic adaptive systems are used in different domains of application such as natural catastrophe prevention (e.g. flood warning), traffic control system, e-commerce applications.

Reconfiguration can be specified at architecture level using an Architecture Description Language (ADL). ADL is a formalism that allows the specification of system's conceptual architecture [19]. It describes a high-level structure of the system rather than implementation details. Conventional ADLs support only static architecture description [18], [9]. Some ADLs, however, support dynamic architecture description, such as [17], [4], [18], [14], [13], [21].

Dynamic reconfiguration of software artifacts is often discussed only at implementation level. Such a late reconfiguration hinders the traceability of configuration and consistency between different levels of the software development life cycle. However, there are some approaches that discuss reconfiguration issues at early stages of development process, for example at architecture or design levels, such as [4], [14], [18]. In these approaches, reconfiguration is expressed in an ad-hoc manner; reconfiguration is described without any explicit identification and separation of concerns. This makes the configuration description hard to understand and the reconfiguration hard to implement. It also hinders a full management of dynamic reconfiguration at architecture level. Moreover, it constitutes a challenge to trace dynamic reconfiguration description/ management at different levels of abstraction.

In this paper, we propose to specify a dynamic reconfiguration at architecture level based on variability description. Variability modeling allows an explicit specification of configurable artifacts. More precisely, we extend our previous proposition of a modular ADL called Dynamic Service-Oriented Product Lines Architecture Description Language (DSOPL-ADL) [2] by enriching the static description with dynamic reconfiguration specifications. This enables a dynamic self-reconfiguration of system at runtime in response

to a context change. Moreover, we propose a process that generates, among several variable configurations described in reference architecture, a concrete configuration that satisfies context values. Furthermore, we automatically generate an executable implementation code from our concrete architectural description.

The rest of this paper is organized as follows: in section 2, we briefly present the modular Dynamic Service-Oriented Product Lines Architecture Description Language (DSOPL-ADL) through an illustrative example. In section 3, we present our approach of self-reconfigurable architecture description based on variability information. In section 4, we show how we can generate a concrete architecture and how to transform it to an executable language. Before concluding, in section 5, we present related work that propose ADLs that comprise dynamic description of reconfiguration.

2. BACKGROUND

2.1 Illustrative Example

We will use an illustrative example to demonstrate concepts related to our proposed approach. This example is about a simplified on-line sales scenario between four services; customer, retailer, warehouse and shipping, as illustrated in figure 1. The customer can access retailer's website, browse the catalog, select some items and command 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 in question.

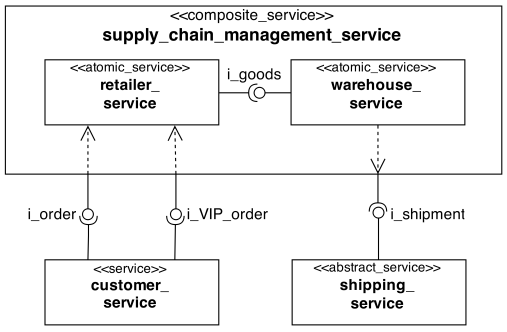

Figure 1: Illustrative example: On-line sales scenario architecture

2.2 Variability-Based Static DSOPL-ADL

DSOPL-ADL is a service-based and variant rich description language that specifies four main types of architectural information: (i) *structural description*, which specifies the architecture in terms of its structural artifacts, (ii) *variability description* defines variation points and alternatives on which system's configuration is based, (iii) *context information*, to which service configurations adapt and (iv) *configuration description* describes possible static configurations based on context and variability information.

As structural description, we represent architecture's main artifacts in terms of services, interfaces and their provided operations. A *service* is an encapsulated unit that interacts with other services through *interfaces*. Each service has a number of provided interfaces and may consume a number of required interfaces. Interfaces define a number of *operations* that are provided by the service. The system is hierarchically decomposed into finer-grained services. A composite service does not execute or implement any functionality by itself, but it delegates its task to one of its child services called atomic services. Listing 1 shows a part of the structural description related to our illustrative example. We can notice that "retailer" and "warehouse" services are atomic services of the composite service "supply chain management". "Retailer" service has two interfaces; one provider interface named `i_order`, which provides two operations and one consumer interface named `i_goods_request`.

Listing 1: Structural Description of Sales Scenario

```
<structural_description>
  <service name="supply_chain_management_service" is_atomic="N
       " ...>
    <interfaces>...</interfaces>
    <sub-architecture>
      <service name="retailer_service" ... is_atomic="Y">
        <interfaces>
          <interface name="i_order" role="provides">
            <operations>
              <operation name="submit_order_request" ...> </
                   operation>
              <operation name="get_catalog" ...> </operation>
            </operations>
          </interface>
          <interface name="i_goods_request" role="consumes">
               ... </interface>
        </interfaces>
      </service>
      <service name="warehouse_service" ... is_atomic="Y">...</
           service>
    </sub-architecture>
  </service>
  ...
</structural_description>
```

In addition to structural description, in DSOPL-ADL, we handle variability information as first-class elements. In variability description, we distinguish three types of variability: (i) *service variability*, where a variation point specifies different available alternative services (ii) *connection variability*, since two services can be differently connected to each other according to constraint's satisfaction and (iii) *composition variability*, where a set of interconnected services may vary, thus the importance to specify such a variability type.

As an example of variability description, we demonstrate in listing 2 a *service variability*. Here, the abstract "shipping" service is a variation point and has two alternative services; either a "relay point" shipping or "home delivery" shipping. During system execution, one of these alternatives and upon condition satisfaction, replaces the abstract "shipping" service.

Listing 2: Service variability description example

```
<variability_description>
  <variation_point name="shipping_variation_point"
       variation_type="service" ...>
    <alternatives>
      <alternative name="home delivery" ...>
        <constraints> ... </constraints>
      </alternative>
      <alternative name="relay point" ...>
        <constraints> ... </constraints>
```

```
    </alternative>
   </alternatives>
  </variation_point>
</variability_description>
```

Context elements need to be described in a self-adaptive architecture. Therefore, we include context description as first-class architectural elements to allow context-aware configurations (i.e. autonomous adaptation according to context changes). A context could either be primitive or composite (several primitive context information contribute to provide a complex context information).

3. DYNAMIC ARCHITECTURE DESCRIPTION BASED ON VARIABILITY DESCRIPTION

In systems where its environment remains unchanged during system execution, a static ADL that describes system's structural and behavioral specifications could be sufficient. In static ADL, architecture's configuration description allows specifying all available configuration rules. However, one valid configuration can be selected at design time and it can not be reconfigured during execution. Reconfiguration specification in DSPOL-ADL is here discussed where a dynamic architecture is specified based on variability description.

3.1 Behavioral Activities

To capture the reconfiguration aspect of a dynamic architecture, we first represent the different inter-service communications (i.e. message flows) which are realized through message passing. Architectural behavior is based on three activities:

1. **receive:** In this type of uni-directional communication, a consumer service (i.e. client) sends a message to the service provider without expecting any instant response. Consequently it only specifies an *input message* and requires no output message. The `receive` element specifies the consumer service using `consumer_instance` attribute and its interface using `consumer_interface` attribute. It also specifies the service provider through `provider_instance` attribute and its interface `provider_interface`. The consumer service triggers the execution of a specific *operation* at the provider's side which is specified in the `operation` attribute. Finally, the arguments passed to the operation are carried on `input_message` attribute.

2. **respond** communication is used to reply to a message that was previously received through a `receive` communication. In that case, values of `consumer_instance`, `consumer_interface`, `provider_instance` and `provider _interface` attributes match the same attributes' values of the corresponding `receive` communication. The response message is passed on the `output_message` attribute.

3. **invoke:** In this type of communication, the service provider receives a message from the service consumer and should in his turn respond by a message. The receiving message is carried on the *input message*, whereas the response is returned on the *output message*. In order to invoke a provider service, the bi-directional `invoke` element is used. Here, the consumer service is identified by the `consumer_instance` and its `consumer_interface`, whereas the provider service is identified by the `provider_instance` and its `provider _interface`. The concrete operation which is called at the provider's side is specified by the `operation` attribute.

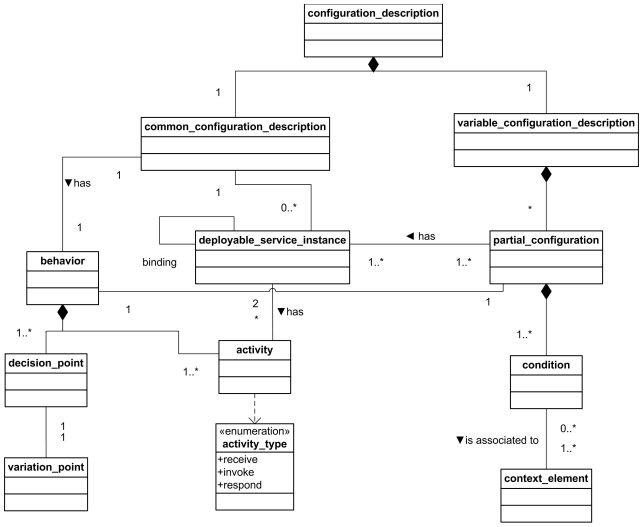

Figure 2: Configuration description meta-model of DSOPL-ADL

In order to specify architecture's common configuration and its variable (re-)configurations, we distinguish two sections in DSOPL-ADL configuration description: (i) a *common configuration description* part, where common services of the architecture are instantiated and bound with a certain behavior and a (ii) *variable configuration description* part, where partial configurations describe the behavior of variation points. Next, each of these parts are detailed. Figure 2, presents the meta-model of configuration description.

3.2 Common Configuration Description

Common configuration description section contains two subsections: *initialization* and *behavior*. The *initialization* sub-section describes the entire structural information about the common services of the architecture. Here, all common services (i.e. those that are not subject to variability) are instantiated and bound. `Deployable service instance` is used to create an instance from a particular service. The `binding` part has two references to two different service interfaces, the one that calls an operation `consumer_interface` and the one that provides the operation `provider_interface`.

In the *behavior* sub-section, we describe the workflow of the entire architecture and mask the parts of the architecture that are subject to variability. The workflow is described in form of a sequence of communication activities (receive, respond and invoke) between services. In each communication activity, we identify the direction of flow by identifying consumer and provider instances as well as specifying the interfaces that will communicate from each side of consumer and provider instances in order to execute a particular operation.

3.3 Variable Configuration Description

The second part of the configuration description is the *variable configuration description* which contains several *partial configurations*. Each partial configuration refers to an alternative point that is defined in the variability description module of DSOPL. A variable partial configuration is triggered by conditions. Any `partial_configuration` has two parts, as also demonstrated in listing 3:

1. *condition* part: where we specify the condition that is driven by context elements. Once the condition satisfied, a given behavior is selected to integrate the existing architecture. In case several conditions of different alternatives of the same variation point are satisfied, the alternative with the higher priority is privileged.

2. *behavior* part: where we specify all dynamic activities that will be executed in order to let the concerning alternative integrate the existing architecture.

The specification of DSOPL-ADL reconfiguration description is given in listing 3. We choose to specify it in REgular LAnguage for XML, New Generation (RELAX NG) [6], an OASIS standard schema language for XML. Relax NG comes in two versions; an XML syntax version and a compact non-XML syntax version. We choose to specify the DSOPL-ADL configuration description using RELAX NG compact version due to its simplicity and expressiveness; unlike other schema languages (e.g. XML Schema) it has a clean formal model.

Listing 3: Configuration description Specification

```
start =
  element configuration_description {
    element common_configuration_description {
      element initialization { services, bindings },
      element behavior {(invoke | receive | respond | element
            decision_point { attribute variability_reference})+ }
    },
    element variable_configuration_description {
      element partial_configuration {
        element condition {
          element context { element name, element value }
        },
        element behavior { services, bindings, (invoke | receive
              | respond)+ }
      }+
    }
 }
services =
  element services {
    element deployable_service_instance {
      attribute service,
      attribute service_instance
    }+
  }
bindings =
  element bindings {
    element binding {
      attribute consumer_instance,
      attribute consumer_interface,
      attribute provider_instance,
      attribute provider_interface
    }+
  }
receive =
  element receive {
    attribute consumer_instance,
    attribute consumer_interface,
    attribute input_message,
    attribute name,
    attribute operation,
    attribute provider_instance,
    attribute provider_interface
  }
respond =
  element respond {
    attribute consumer_instance,
    attribute consumer_interface,
    attribute name,
    attribute operation,
    attribute output_message,
    attribute provider_instance,
    attribute provider_interface
  }
invoke =
  element invoke {
    attribute consumer_instance,
    attribute consumer_interface,
    attribute input_message,
    attribute name,
    attribute operation,
    attribute output_message,
    attribute provider_instance,
    attribute provider_interface
  }
```

Back to our example, both "customer" and composite "supply chain management" services make part of the common architecture, since they are not subject to any variation. This implies that their instantiations and bindings can be specified at design time, as depicted in the `common _configuration_description` part of listing 4. Whereas shipping services "relay point" and "home delivery", which are alternatives of abstract "shipping" service, are dynamically instantiated at runtime to replace the abstract "shipping" service. This is why their instantiation and behavior are specified in the `variable_configuration_description` part.

Listing 4: Configuration description of sales scenario

```
<configuration_description>
 <common_configuration_description>
  <initialization>
   <services>
        <deployable_service_instance service="customer_service"
             service_instance="customer_service_instance"/>
        <deployable_service_instance service="
             supply_chain_management_service" .../>
   </services>
   <bindings>
    <binding consumer_instance="customer_service_instance"
         consumer_interface="i_customer_order" provider_instance
         ="supply_chain_management_service_instance"
         provider_interface="i_order_delegation" />
   </bindings>
  </initialization>
  <behavior>
   <receive name="receive_customer_request" consumer_instance="
        customer_service_instance" consumer_interface="
        i_customer_order" provider_instance="
        supply_chain_management_service_instance"
        provider_interface="i_order_delegation" operation="
        prepare_order" input_message="receive_order_items">
   </receive>

   <decision_point variability_reference="
        shipping_variation_point"/>

   <respond name="send_order_details" consumer_instance="
        customer_service_instance" consumer_interface="
        i_customer_order" provider_instance="
        supply_chain_management_service_instance"
        provider_interface="i_order_delegation" operation="
        prepare_order" output_message="out_order_details">
   </respond>
  </behavior>
 </common_configuration_description>

 <variable_configuration_description>
  <partial_configuration>
   <condition>
```

```
      <context>
       <name> shipping </name>
       <value> home </value>
      </context>
  </condition>
  <behavior>
   <services>
      <deployable_service_instance service="
           home_delivery_shipping_service" service_instance="
           home_delivery_shipping_service_instance" />
      </services>
      <bindings>
       <binding consumer_instance="
           supply_chain_management_service_instance
           consumer_interface="i_shipment_ready_delegation"
           provider_instance="
           home_delivery_shipping_service_instance"
           provider_interface="i_home_delivery" />
      </bindings>
      <invoke name="invoke_home_delivery_service"
          consumer_instance="
          supply_chain_management_service_instance
          consumer_interface="i_shipment_ready_delegation"
          provider_instance="
          home_delivery_shipping_service_instance"
          provider_interface="i_home_delivery" operation="
          order_delivery_to_home" input_message="in_ship-
          order" output_message="out_ship_order">
      </invoke>
  </behavior>
 </partial_configuration>
 <partial_configuration>
  <condition>
      <context>
       <name> shipping </name>
       <value> relay_point </value>
      </context>
  </condition>
  <behavior>
      ...
       <deployable_service_instance service="
           relay_point_delivery_shipping_service" ... />
      <bindings .../>
      <invoke .../>
  </behavior>
 </partial_configuration>
</variable_configuration_description>
</configuration_description>
```

In the `<initialization>` part of `<common_configuration_description>`, "customer" and "supply chain management" services are bound together through their interfaces in the `<binding>` part, as depicted in listing 4. In the `<behavior>` part, the workflow starts by a trigger activity from the "customer" service, which makes a sales order. The "supply chain management" service receives the ordered items through the `input_message = "receive_order_items"` of `<receive>` activity and executes the operation `operation = "prepare_order"`.

Next, a `<decision_point>` indicates the existence of a variation point with a reference to the `"shipping_variation_point"` which is explicitly defined in the *variability description* part of DSOPL-ADL. The `<decision_point>` part of the behavior will not be executed at design time but at runtime. The different partial configurations related to each variation point are described in the `<variable_configuration_description>` part of the *configuration description* of DSOPL-ADL. All related instructions of instantiating and binding either "home delivery shipping" service or "relay point shipping" service to the "supply chain management" service will be performed only at run-time according to the context value of "shipping" in the `<condition>` part of each `<partial_configuration>`. Once the corresponding shipping service is bound, "supply chain management" service will invoke one of these shipping services and execute either "order_delivery_to_home" or "order_delivery_to_relay _point" operations. Obviously, information about ordered items are passed from the "supply chain management" service to selected shipping service through the message `input_message = "in_ship_order"`. Likewise, shipping information (such as shipping delays and costs) are returned through the message `output_message = "out_ship_order"`.

Finally, as a respond to customer's initial order request, the "supply chain management" service executes the "prepare_order" operation and sends the order details to the "customer" service through an `output_message = "out_order_details"` within a `<respond>` activity. Here, both service instances (`consumer_instance` and `provider_instance`) as well as their interfaces (`consumer_interface` and `provider_interface`) are the same of the `<receive>` activity, since it is a respond to that request.

4. ARCHITECTURE CENTRIC RECONFIGURATION OF SERVICE-BASED SYSTEM

4.1 Concrete Architecture Generation

A concrete configuration of architecture is generated from a given reference architecture's *configuration description* through the following consecutive steps:

1. The `<common_configuration_description>` part of the configuration description is copied to the concrete configuration description of architecture without any modifications, except copying `<decision_point>` part which is treated differently in step 2.

2. Each variation point called `<decision_point>` in the `<common_configuration_description>` part of DSOPL-ADL is replaced by an appropriate `<partial _configuration>` according to context value satisfaction of that partial configuration.

3. Consequently, all service instances related to that partial configuration in addition to their bindings both described in `<services>` and `<binding>` sub-sections of `<partial_configuration>` are integrated to the concrete architecture to `<services>` and `<binding>` sections, respectively.

Figure 3 demonstrates how a concrete architecture is generated in on-line sales example after integrating a partial configuration (here, relay point delivery shipping service) to the common configuration part.

4.2 Executable Code Generation

In this section, we demonstrate how an executable business process can be obtained based on the generated DSOPL concrete architecture. Among existing business process specification languages, we choose to generate DSOPL's concrete architectural description in Business Process Execution Language (BPEL), since it is the most dominant language [11] and has become a de-facto standard for specifying workflow in a service-oriented environment and hence executing business processes for web services composition [3]. BPEL defines in principle two main types of activities: basic activities to interact with external services (invoke, receive, reply) and structural activities to control the internal business workflow by conditional choices, parallel activities and looping.

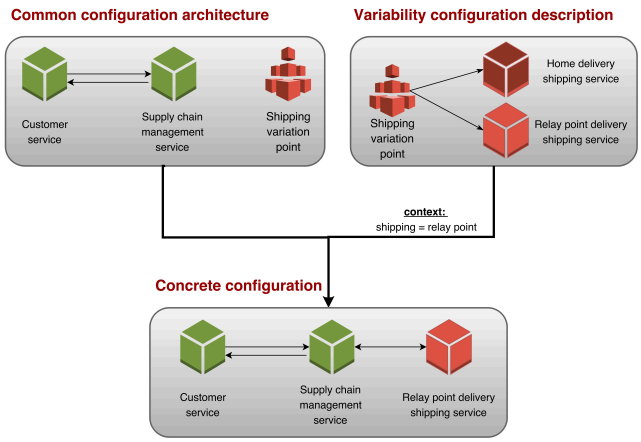

Figure 3: Concrete architecture generation

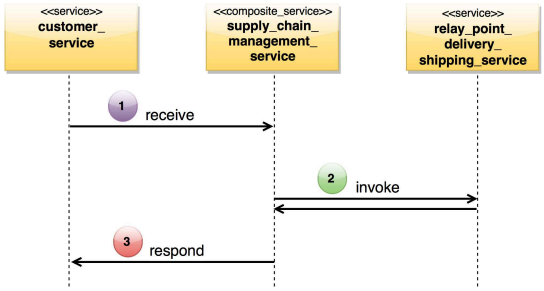

Figure 4: Sales order activities' sequence

The most challenging part of BPEL's code generation is the transformation of DSOPL-ADL's behavior into BPEL's activities due to the difference of workflow realization between both paradigms. Conceptually, there are two different perspectives of realizing a workflow: choreography and orchestration. In choreography, the configuration is realized between autonomous peer-to-peer collaborations, whereas in orchestration a single central workflow engine coordinates the execution flow between all involved Web services. In DSOPL-ADL, configuration is specified following the choreography perspective; i.e. configuration is realized through message passing from one service to another one without intermediaries. Figure 4 demonstrates the sequence of activities for the sales order concrete architecture example. BPEL, in contrast, supports both choreography and orchestration perspectives. However, in order to obtain an executable process, orchestration perspective should be followed [15]. That is why we propose a mapping between DSOPL-ADL concepts at architecture level and BPEL concepts at implementation level. The purpose of this mapping is to generate a BPEL skeleton from DSOPL-ADL description. Table 1 displays the concepts of our DSOPL-ADL and their mapping to BPEL paradigm. We can notice the existence of one-to-one mapping between both paradigms (such as the structural element "deployable_service_instance" in DSOPL-ADL is mapped to "partnerLink" in BPEL), one-to-many mapping (such as the activity "receive" in DSOPL-ADL is mapped to four consecutive activities in BPEL) and unfortunately there are concepts that do not have any direct correspondence in one of the two paradigms (such as the "binding" which does not have any correspondence in BPEL or "variables" in BPEL which do not have any correspondence in DSOPL-ADL).

According to the mapping rules presented in table 1, the sequence of activities in the concrete architecture of on-line sales order example (represented graphically in figure 4) is transformed to BPEL activities as demonstrated in figure 5. We can notice that a new actor is added in BPEL called "Sales Order Process" which is a central process that coordinates the message passing and order of flow between services called "Partner Links". This central process must be either a consumer or provider part in all BPEL activities, this explains why each DSOPL-ADL activity is mapped to one or more activities in BPEL. The "receive" activity in DSOPL-ADL (designated by number 1), which receives customer's order and prepares it, is represented by four activities in BPEL: receive, assign_customerRequest, invoke_ prepareOrder and assign_orderInformation (all designated by number 1 in figure 5). The bi-directional activity "invoke" in DSOPL-ADL (designated by number 2 in figure 4) is transformed to two activities in BPEL: invoke_ OrderDeliveryToRelayPoint and assign_CustomerOutput. This latter copies the content of returned message from relay_point _delivery_shipping_service and assigns it to a local temporary variable. Finally, the "respond" activity (designated by number 3 in figure 4), which is in charge to return order's confirmation and information about shipping to the customer, is transformed to the "reply" activity in BPEL (designated by 3 in figure 5).

Due to space limitation, we do not demonstrate the entire BPEL code, but only one behavior transformation. In the sequence section, the transformation result of `<receive>` activity is demonstrated in listing 5. It is composed of four sequential activities: `<receive>`, `<assign>`, `<invoke>` and `<assign>`.

Listing 5: Transformation result of receive activity

```
<sequence>
  <receive name="start" partnerLink="customer_service" ...
       operation="asyncOperation" variable="inputVar"
       createInstance="yes">
  </receive>
  <assign name="assign1">
    <copy>
      <from variable="inputVar" ... />
      <to variable="Process_orderIn" ...></to>
```

DSOPL-ADL behavioral concepts	BPEL concepts
structural elements	
deployable_service_instance	partnerLink
operation	role's name & invoke's name
behavior	sequence
interactive activities	
receive	4 consecutive activities: receive, assign, invoke and assign
invoke	2 consecutive activities: invoke and assign
respond	reply
examples of missing correspondence	
binding	-
-	variables

Table 1: DSOPL-ADL to BPEL mapping

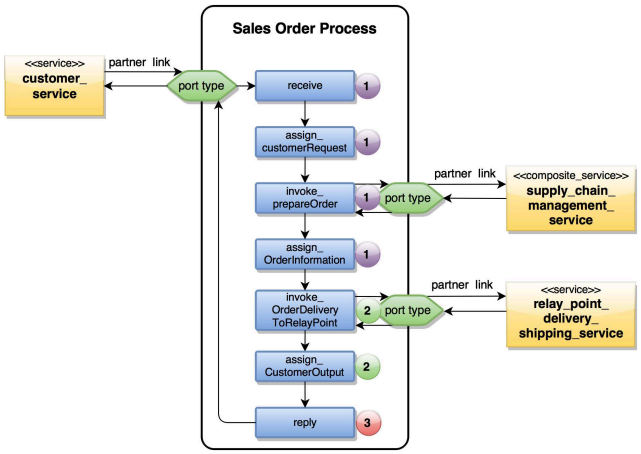

Figure 5: Transformation to BPEL - sales order example

```
    </copy>
  </assign>
  <invoke name="prepare_order" partnerLink="
       supply_chain_management_service" operation="
       prepare_order" ... inputVariable="Process_orderIn"
       outputVariable="Process_orderOut"/>
  <assign name="assign2">
    <copy>
      <from variable="Process_orderOut" ... />
      <to variable="Order_delivery_to_relay_pointIn" ... />
    </copy>
  </assign>
</sequence>
```

5. RELATED WORK

While common ADLs have more or less agreed on what elements to represent regarding structural specifications, there is not yet a common agreement of what the ADLs shall specify from behavioral point of view. In global, the evolution of architecture at runtime may happen under several forms:

- creating (instantiating) / removing composing elements
- binding / unbinding composing elements to the architecture
- reconfiguring architecture (modifying connections)
- upgrading existing composing elements (substitution of composing elements)

It is evident that dynamicity is differently considered and perceived in different research communities, hence the importance to classify dynamic architecture descriptions (whether described in ADLs or other formalisms) into two types: (i) *centralized dynamicity management*, where all instructions of modifying system's behavior at architecture level are defined in a central configuror. Hence the behavioral description is independent from architectural elements' functionality definition. Various approaches have emerged to explicitly describe interaction between architecture's structural elements in form of a sequence of activities such as in [21], [13] and [1]. (ii) *event-driven dynamicity*, where constraints in form of triggers or events are defined inside each architectural element. Here, an internal observer listens to environment's changes and modifies elements' behavior (e.g. its connection with other elements) only if a pre-defined constraint or condition is satisfied. Darwin [17], Plastik [14] and Dynamic Wright [4] are examples that use this technique.

Among existing ADLs in the literature, only few of them support dynamic reconfiguration such as C2 SAD(E)L[18], Darwin [17], π-ADL [20], Rapide [16], ACME/Plastik [14] and Dynamic Wright [4].

Among service-based ADL that handle dynamic architecture, π-ADL for WS-Composition [21] is a service-oriented ADL for Web Service (WS) composition that has the same roots as π-ADL [20] and highly relies on BPMN's visual notation. It formally describes service-oriented dynamic architectures from both structural and behavioral viewpoints. π-ADL for WS-Composition is considered a dynamic ADL since third-party services can be discovered and bound at runtime. The definition of the architecture is divided in two parts: (i) *structure* definition, where each component is defined and (ii) *behavior*, where the instances of components, and connectors are defined abstractly and also the connection between each component and connector.

Another example of service-oriented ADL supporting dynamicity is Service-Oriented Architecture Description Language (SOADL) [13]. In this approach, behavioral specification is represented by a sequence of actions that describe the temporal constraints between operations in one or more ports. It specifies the architecture in terms of services, interfaces, behavior, semantics and quality properties. It also supports architecture-based service composition. By observing the pseudo-schema syntax of SOADL, we can distinguish four main parts: (1) *Port* is the interaction point of service. It plays a provider or requester role. (2) *Behavior* consists of a sequence of actions, either a basic action or a composite one. (3) *SubArchitecture* part describes the structure of the (sub)system of a composite service. More precisely this part includes three parts: (i) *Dependency* part declares local or external service types that the (sub)system may use, (ii) *Configuror* part specifies all possible configurations for the given system. Each configuration is triggered by an event. (iii) *Constraint* part defines a set of constraints that determine how an architectural design is permitted to evolve over time. (4) Finally, *Properties* part describes properties of security, transaction, load balance, version, or information related to implementation. SOADL discusses the dynamic reconfiguration of services by treating only the substitution of service instances in case of unavailability of a main service. Substituting services are statically defined at design-time in the *configuror* part.

We classify in table 2 existing dynamic ADLs according to their support of the aforementioned dynamic actions. Existing software architecture configuration description languages either do not allow any dynamic reconfiguration at run-time and thus are considered as static ADLs, or they provide certain dynamicity according to planned and predefined changes in a given architecture. However, the reconfiguration decisions within these approaches are expressed in an ad-hoc manner and not in an explicit specification (e.g. using a variability description) as in our proposed approach DSOPL-ADL.

6. CONCLUSION AND PERSPECTIVES

We have presented an architecture description language that allows the reconfiguration at runtime of a software architecture based on variability description. To manage the

dynamic ADL/ dynamic action	Darwin [17]	Dynamic Wright [4]	SOADL [13]	π-ADL [20]	C2 SAD(E)L [18]	π-ADL for WS-Composition [21]	Plastik [14]
create architectural element	Y	Y	N	Y	Y	N	Y
remove architectural element	N	Y	N	N	Y	N	Y
bind architectural element to architecture	Y	Y	Y	Y	Y	Y	Y
unbind architectural element to architecture	N	Y	Y	N	Y	N	Y
reconfigure architecture (modify connections)	N	Y	Y	N	Y	N	Y
substitute architectural element (upgrade)	N	Y	Y	N	Y	N	Y

Table 2: Supported dynamic actions in existing dynamic ADLs

runtime reconfiguration at architecture level, we have extended a previous work called DSOPL-ADL which described static structural, variability and context elements and extended it with dynamic reconfiguration aspect. For that purpose, we presented a reconfiguration description meta-model as well as the language schema. During system's execution, architecture can adapt its behavior to environment changes that are specified as context elements and consequently generate a concrete architecture. Furthermore, we transformed generated concrete architecture to another level of abstraction, such as to business process level (i.e. to BPEL).

7. REFERENCES

[1] Web services business process execution language version 2.0, 2007.

[2] S. Adjoyan and A. Seriai. An architecture description language for dynamic service-oriented product lines. In *27th International Conference on Software Engineering and Knowledge Engineering*, July 2015.

[3] A. Albreshne, P. Fuhrer, and J. Pasquier. Web services orchestration and composition, 2009.

[4] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, pages 21–37. Springer, 1998.

[5] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 28–33, 2004.

[6] J. Clark and M. Murata. Relax ng specification. http://relaxng.org/spec-20011203.html, 2001.

[7] A. Classen, A. Hubaux, F. Sanen, E. Truyen, J. Vallejos, P. Costanza, W. De Meuter, P. Heymans, and W. Joosen. Modelling variability in self-adaptive systems: Towards a research agenda. In *Proceedings of international workshop on modularization, composition and generative techniques for product-line engineering*, pages 19–26, 2008.

[8] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond (2nd Edition)*. Addison-Wesley Professional, 2 edition, 2010.

[9] C. Deiters and A. Rausch. A constructive approach to compositional architecture design. In *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 75–82. Springer, 2011.

[10] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software & Systems Modeling*, 12(2):349–367, 2013.

[11] N. Griffiths and K.-M. Chao. *Agent-Based Service-Oriented Computing*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[12] C. Jaggernauth, B. Kaminska, and D. Gubbe. Context-aware model for dynamic adaptability of software for embedded systems. *International Journal of Computer (IJC)*, 19(1):91–113, 2015.

[13] X. Jia, S. Ying, H. Cao, and D. Xie. A new architecture description language for service-oriented architecture. In *6th International Conference on Grid and Cooperative Computing GCC*, pages 96–103, 2007.

[14] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL specifications to an efficient and reconfigurable runtime component platform. In *5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005*, pages 131–140.

[15] M. B. Juric. A hands-on introduction to bpel. http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html, 2007.

[16] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21:336–355, 1995.

[17] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153. Springer-Verlag, 1995.

[18] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 24–27. ACM.

[19] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, Jan 2000.

[20] F. Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 2004.

[21] F. Oquendo. π-ADL for WS-Composition: A Service-Oriented Architecture Description Language for the Formal Development of Dynamic Web Service Compositions. In *Second Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2008)*, pages 1–14, Aug. 2008.