π-*ADL for WS-Composition*: A Service-Oriented Architecture Description Language for the Formal Development of Dynamic Web Service Compositions

Flavio Oquendo

European University of Brittany
University of South Brittany – VALORIA – BP 573 – 56017 Vannes Cedex – France
`flavio.oquendo@univ-ubs.fr`

Abstract. *Enabling the specification of dynamic service-oriented architectures is a key challenge for an Architecture Description Language (ADL). This paper describes π-ADL for WS-Composition, a novel ADL that has its roots in the ArchWare European Project. It is a formal language specially designed for modeling dynamic architectures based on the typed π-calculus. While most ADLs focus on describing static architectures from a structural viewpoint, π-ADL focuses on formally describing dynamic architectures from both structural and behavioral viewpoints. How π-ADL for WS-Composition can be used for specifying dynamic Web Service compositions is introduced through a case study. Its design principles, concepts and notation are presented. The π-ADL for WS-Composition toolset is outlined.*

1. Introduction

Software architecture has emerged as an important subdiscipline of software engineering [12][23]. A key aspect of the design of any software system is its architecture, i.e. the fundamental organization of the system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [6].

A software architecture can be characterized according to its evolution at run-time [14]:

- static architectures: the architecture does not evolve during the execution of the system;
- dynamic architectures: the architecture may evolve during the execution of the system, e.g. components are created, discovered, deleted, or reconfigured at run-time.

An architecture description specifies an architecture. An architecture can be described according to different viewpoints. Two viewpoints are frequently used in software architecture [6][3]: the structural and the behavioral viewpoints.

The structural viewpoint may be specified in terms of:

- components (units of computation of a system),
- connectors (interconnections among components for supporting their interactions),
- configurations of components and connectors.

Thereby, from a structural viewpoint, an architecture description should provide a specification of the architecture in terms of components and connectors and how they are composed together. Further, in the case of a dynamic architecture, it must provide a specification of how its components and connectors can change at run-time.

The behavioral viewpoint may be specified in terms of:

- actions a system executes or participates in,
- relations among actions to specify behaviors,
- behaviors of components and connectors, and how they interact.

Architectures are described according to different architectural styles [33]. A mainstream architectural style is Service-Oriented Architecture (SOA) [16]. SOA is an architectural style for constructing complex software-intensive systems from a set of universally interconnected and interdependent building blocks, called services, where functionality is grouped around business processes.

Web services can be used to implement a service-oriented architecture. A major focus of Web services is to make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages. Each SOA building block can play one or more of the three following roles[1] [35]:

- the service provider provides the Web service and publishes its interface and access information to the service broker;
- the service broker makes the Web service interface and implementation access information available to potential service requesters;
- the service requester discovers a Web service by locating an entry in the broker registry, managed by the service broker, and then binds to the service provider in order to invoke the Web service.

Web services are supported by different languages. Interfaces and access information of Web services are described in WSDL (Web Services Description Language) [36]. In line with WSDL, WS-BPEL (Web Services Business Process Execution Language) and WS-CDL (Web Services Choreography Description Language) extend the service concept by providing support for defining and enforcing respectively orchestrations and choreographies of fine-grained services into more coarse-grained composite services.

Web service orchestration and choreography describe two ways of creating business processes by composition of Web services. Orchestration refers to an executable business process, including business logic and activity execution order. One Web service orchestrates the interactions with the other services. Choreography refers to the possible sequences of message exchanges for achieving a business goal. Orchestration can interact with both internal and external Web services. Choreography is associated with the public message exchanges that occur among external Web services.

Orchestration differs from choreography in that it describes a process flow between services, controlled by a single party. Choreography tracks the exchange of messages involving multiple parties, where no party owns the message flows [29].

Business processes, including orchestrations and choreographies, are defined using business process modeling languages. BPMN (Business Process Modeling Notation) [18] is a standardized visual notation for modeling business processes.

In architectural terms, in a SOA, services (which are provided, can be requested and whose interfaces descriptions can be published and discovered) are the architectural

[1] Service provider and service requester are mandatory roles in SOAs. Service broker and the registry of services are thereby optional; however they play key roles in dynamic web service compositions.

components, messaging passing connections are the architectural connectors, and orchestration and choreography the architectural configurations of components and connectors.

Designing a Web service-oriented architecture thereby involves different artifacts expressed in different languages, where these languages are semi-formal and possibly overlapping. This is a complex and error-prone task that can lead to poorly-designed architectures and failures in the implementation of systems with the required qualities.

Formal methods are increasingly used for modeling software architectures [24][9]. Their potential advantages have been widely recognized, in particular "to design the system right" by formally describing its structure and behavior enabling to check its correctness with respect to qualities.

Enabling the specification of dynamic Web service-oriented architectures is a key challenge for an Architecture Description Language (ADL). The research challenge is therefore threefold:

- To support the description of dynamic service-oriented architectures from structural and behavioral viewpoints. For describing dynamic architectures, the ADL must be able to describe changing structures and behaviors of services, their discovery, connections and compositions at run-time.
- To support the description of service-oriented architectures where business processes are modeled in visual notations such as BPMN, orchestrations in languages such as WS-BPEL, choreographies in languages such as WS-CDL and interfaces in languages such as WSDL. The ADL must be able to provide a unified foundation for describing the different SOA artifacts: business processes, orchestrations, choreographies, and interfaces.
- To support the description of service-oriented architectures enabling to rigorously reason about and verify their qualities, in particular related to conformance and correctness. The ADL must be formally defined.

π-ADL for WS-Composition has been designed to meet this challenge.

The remainder of this paper is organized as follows. Section 2 introduces the approach and core concepts of *π-ADL for WS-Composition*. Section 3 presents through a case study how *π-ADL for WS-Composition* can be used for specifying a dynamic service-oriented architecture. Section 4 briefly outlines the *π-ADL for WS-Composition* toolset and its applications. In section 5 we compare *π-ADL for WS-Composition* with related work. To conclude we summarize, in section 6, the main contributions of this paper.

2. The Approach and Core Concepts

For enabling the specification of dynamic Web service-oriented architectures, meeting the threefold research challenge cited so far, we have designed a service-oriented architecture description language as a new member of the π-Architecture Description Language (π-ADL) [19][27][25] family. π-ADL provides a novel and customizable ADL that is general-purpose and Turing-complete[2]. It has been designed in the ArchWare[3] European Project, to be the root of an open family of ADLs, where members of the family are embedded domain-specific languages.

[2] This means that every possible computation can be expressed in π-ADL.

[3] The ArchWare European Project has been partially funded by the European Commission under contract No. IST-32360 in the IST Framework Program.

π-ADL for WS-Composition is, as a result, a service-oriented architecture description language for the formal development of dynamic Web service composition. It is:

- service-oriented: it provides constructs directly related to the service-oriented architecture style, including services, binding connections and their compositions by orchestration and choreography;
- formal: it is a formal language based on the typed *π*-calculus [13]; a verification toolset is provided for automated checking of service-oriented architectural properties [10];
- practical: it provides different user-friendly concrete syntaxes – textual and visual (via a BPMN profile) – hiding the complexity of its formal underpinnings to ease its use by architects and engineers;
- executable: an architecture description specifies how the system behaves in terms of service-oriented architecture concepts; its virtual machine runs SOA descriptions step by step, randomly or in an event-driven manner at design-time (as a simulation engine) and, if deployed, orchestrate and choreograph services at run-time (as an enactment engine).

π-ADL for WS-Composition takes its roots in previous work concerning the use of *π*-calculus as semantic foundation for architecture description languages [19][2][15] and business process management languages [30][31][5].

Indeed, a natural candidate for expressing (run-time) behavior would be the *π*-calculus as it is [13], which provides a general model of computation and is Turing-complete. However in *π*-calculus, even if every computation is possible to express, this expression becomes quite often too complex. In fact, the classical *π*-calculus is not suitable as an ADL since it does not provide architecture-centric constructs to friendly express architectures in particular with respect to architectural structures. Therefore, a language encompassing both structural and behavioral architecture-centric constructs is needed. *π*-ADL is this encompassing core language. It achieves Turing completeness and high architecture expressiveness with a simple formal notation. *π-ADL for WS-Composition* was defined as a style-specific extension of the typed *π*-calculus on top of *π*-ADL, of which it is an embedded domain-specific language.

But why do architects need a formal service-oriented ADL? Of course, ADLs that provide a semi-formal notation helps as the state of the practice today is mainly to use semi-formal notations for documenting software and service architectures [3]. A semi-formal ADL helps but it is not enough. Indeed, in order to check correctness, a formal ADL encompassing both structural and behavioral constructs is mandatory.

π-ADL for WS-Composition is this formal yet practical ADL. It supports automated verification of service-oriented structural and behavioral properties. In addition that, it is executable: it is very practical to "run" the architecture descriptions in order to validate its dynamic semantics.

For bridging the gap from business process to composition of services, it provides a BPMN profile that eases its use and fully automates its translation to executable specifications.

Precisely, *π-ADL for WS-Composition* supports description of service-oriented architectures from a run-time perspective. In *π-ADL for WS-Composition*, a service-oriented architecture is described in terms of services, connections and their compositions. Figure 1 depicts its main constituents.

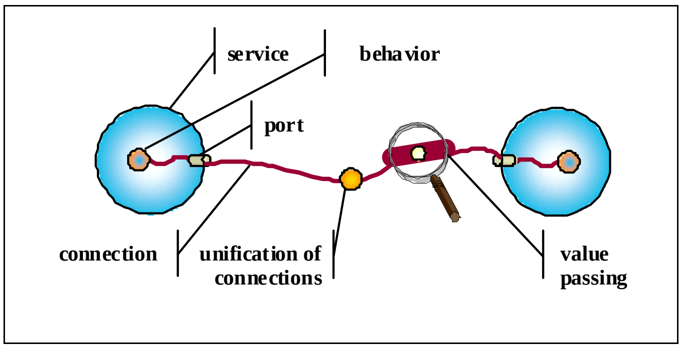


Figure 1. Architectural concepts in π-*ADL for WS-Composition*

A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service requester. A service is described in terms of external ports and is provided by a service provider specified in terms of its behavior[4]. Its architectural role is to specify computational elements of a service-oriented architecture.

A port is described in terms of connections between a service and its environment. Its architectural role is to put together connections providing an interface between the service provider and its environment. Protocols may be enforced by ports and among ports.

A connection is a basic interaction point. Its architectural role is to provide a communication channel between a service requester and a service provider. Connections may be passed or unified to enable communication.

A service can send or receive message values via connections. They can be declared as output connections (values can only be sent), input connections (values can only be received), or input-output connections (values can be sent or received).

Therefore, services provide the locus of computation, while connections enable the interaction between service requesters and service providers. In order to have actual communication, there must be a connection connecting them.

Architecturally, from a black-box perspective, only service ports (with their connections) and values passing through connections are observable. From a white-box perspective, service behaviors are also observable.

Services can be composed together to construct composite services, which may themselves be services. Composite services can be decomposed and recomposed in different ways or with different services in order to construct different compositions.

Composite services comprise external ports (i.e. observable from the outside) and a behavior defined as a service orchestration. These external ports receive values coming from either side, incoming or outgoing, and simply relay it to the other side keeping the mode of the connection. Ports can also be declared to be restricted. In that case, constituents of composite services can use connections of restricted ports to interact with one another but not with external elements.

[4] As defined by the W3C's WSA [35], a service is an abstract notion that must be provided by a concrete behavior. The behavior (that is specified in the architecture description) is implemented by a concrete piece of software or hardware that sends and receives message values on behalf of the service.

Architectures are described as composite services comprising a behavior defined as a service choreography. An architecture can itself be a composite service in another architecture, i.e. a sub-architecture.

π-ADL for WS-Composition provides primitive constructs for supporting the description of all these service architectural concepts.

3. The Language and a Case Study

We present hereafter *π-ADL for WS-Composition* in more detail. Instead of providing a formal description [26], we will illustrate its main concepts through the formal description of a typical dynamic service-oriented architecture, including both orchestration and choreography. We will also illustrate how BPMN can be used as a visual notation to describe business processes that are formally translated as service-oriented architectures using *π-ADL for WS-Composition*.

Figure 2 shows the description of a purchasing business process in BPMN [32]. There are three roles – the *Customer*, the *Broker*, and the *Bank* – each one modeled by an orchestration. A choreography models their interactions.

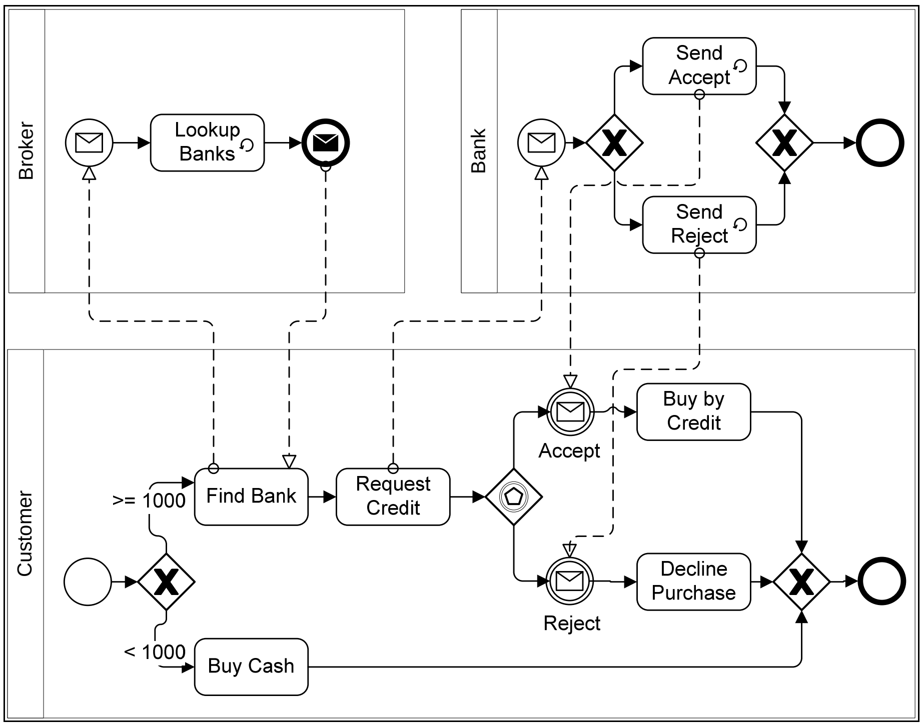


Figure 2. Purchase process description in BPMN

When a customer launches the purchase process, as described in the *Customer*'s pool, first there is a decision based on the price of the article. If it is lower than a threshold (in that case € 1000), the customer pays cash. However, if it is equal or higher than € 1000, the customer will request a broker to find a bank with the lowest interest rate at that time. Once the customer gets the reference of a bank from the broker, s/he will request a credit to that bank. The bank will make a decision: to accept or reject the requested credit. Using a deferred choice, the customer will handle the reply from the bank. In case the bank accepts, the customer will buy the article using the credit. In case it rejects, s/he declines the purchase.

The architecture of the system supporting the purchase process is dynamic: new banks can be registered with the broker during the execution of the system. Banks are dynamically found by the broker, and are thereby not known at design-time. The customer knows the broker at design-time and only knows the bank at run-time according to what was discovered by the broker. The binding between the customer and the broker is performed at design-time and between the customer and the bank at run-time, possibly with different banks for different purchases.

In terms of the service-oriented architecture style, the architecture is described by orchestrations and choreographies. In each role's pool, an orchestration describes how different services are consumed to provide a more complex service and a choreography describes how the different roles dynamically interact to provide the overall business service.

In terms of control flows in the orchestration, the following BPMN constructs are used: *sequence flow*, *exclusive-or gateway*, and *event-based gateway*. In terms of message flows in the choreography, the *send* and *receive* constructs are used.

These BPMN constructs can be translated to *π-ADL for WS-Composition* by the recurrent application of Workflow Patterns [34]. This mapping from BPMN to *π-ADL for WS-Composition* is defined in terms of a formal transformation language, namely π-ARL [20]. It provides a formal semantics for BPMN[5] in terms of the π-calculus through π-ADL (in a similar way to the one defined in [21] for UML2).

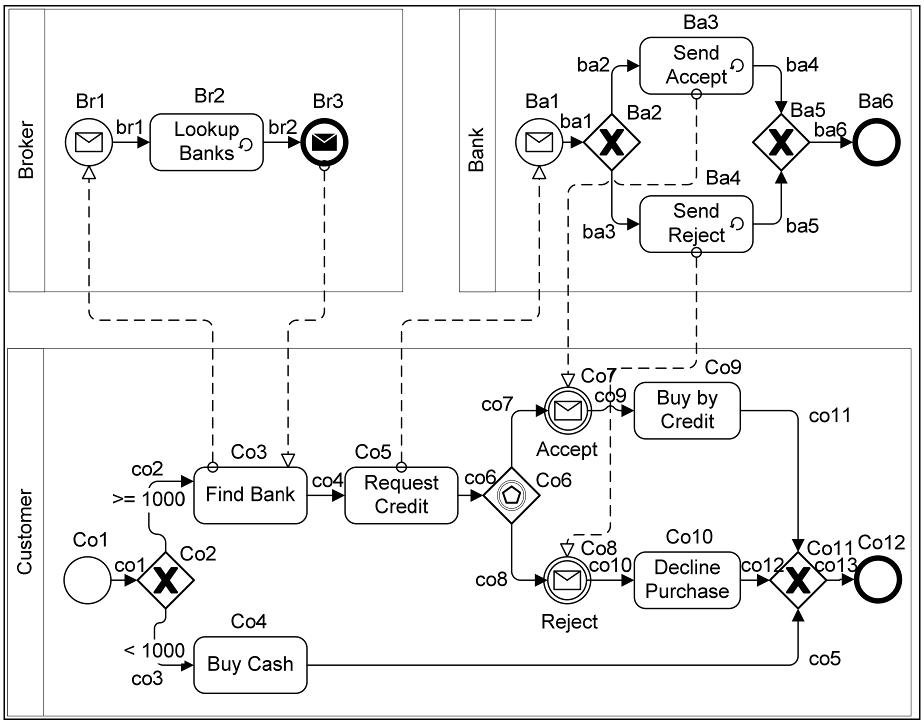


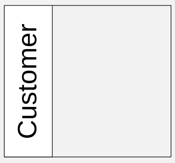
Figure 3. Annotated purchase process description in BPMN

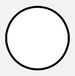
Using the method proposed in [31], the business process described in BPMN in Figure 2 is annotated as shown in Figure 3**Erreur ! Source du renvoi introuvable.**. All nodes and flows are annotated with unique labels (partially shown). In addition, basic type and service information is provided.

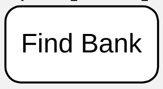
[5] BPMN lacks a formal semantics. The explicit mapping from BPMN to *π-ADL for WS-Composition* explicitly assigns the semantics of a BPMN business process model in terms of the π-calculus through π-ADL.

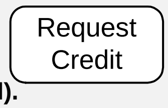
The *Customer*'s orchestration described in the *Customer*'s pool is translated to *π-ADL for WS-Composition* as follows.

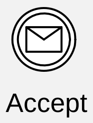
```
service Customer is abstraction(
broker : connection[connection[connection[Request]]],
shop : connection[view[buycash : connection[], buybycredit : connection[],
declinepurchase : connection[]]], amount : Real ) { type Amount is Real.
  type Accept is connection[]. type Reject is connection [].
  type Request is view[amount : Amount, accept : Accept, reject : Reject].
  port is { connection broker is connection(connectionToBank : connection[connection[Request]]).
    connection broker::connectionToBank is connection(request : connection[Request]).
    connection broker::connectionToBank::request is connection(Request).
    connection broker::connectionToBank::request::accept is connection().
    connection broker::connectionToBank::request::reject is connection().
    connection bank is connection(Request) }.
  activity Co1 is abstraction( co1 : connection[Real], amount : Real )
    behavior is { unobservable. via co1 send amount. done }.
  activity Co2 is abstraction( co1 : connection[Real], co2 : connection[Real],
  co3 : connection[Real] )
    behavior is { replicate via co1 receive amount : Real.
      choose {  if (amount >= 1000.00) do via co2 send amount. done
      or        if (amount < 1000.00) do via co3 send amount. done} }.
  activity Co3 is abstraction( co2 : connection[Real], co4 : connection[connection[Request], Real],
  broker : connection[connection[connection[Request]]] )
    behavior is { connection connectionToBank is connection(connection(Request)).
      replicate via co2 receive amount : Real. via broker send connectionToBank.
      via connectionToBank receive bank : connection[Request].
      via co4 send (bank, amount). done }.
  activity Co4 is abstraction( co3 : connection[Real], co5 : connection[] )
    behavior is { replicate via co3 receive amount : Real. via shop::buycash send.
      via co5 send. done }.
  activity Co5 is abstraction( co4 : connection[connection[Request], Real],
  co6 : connection[connection[],connection[]] )
    behavior is { replicate via co4 receive (bank : connection[Request], amount : Real).
      value request is view(amount = amount, accept = connection(), reject = connection()).
      via bank send request. via co6 send (request::accept, request::reject). done }.
  activity Co6 is abstraction( co6 : connection[connection[],connection[]], co7 : connection[],
  co8 : connection[] )
    behavior is { replicate via co6 receive (accept : connection[], reject : connection[]).
      choose {  via accept receive. via co7 send. done
      or        via reject receive. via co8 send. done
      } }.
  activity Co7 is abstraction( co7 : connection[], co9 : connection[] )
    behavior is { replicate via co7 receive. unobservable.
      via co9 send. done }.
  activity Co8 is abstraction( co8 : connection[], co10 : connection[] )
    behavior is { replicate via co8 receive. unobservable. via co10 send. done }.
  activity Co9 is abstraction( co9 : connection[], co11 : connection[] )
    behavior is { replicate via co9 receive. via shop::buybycredit send.
      via co11 send. done }.
```

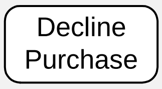




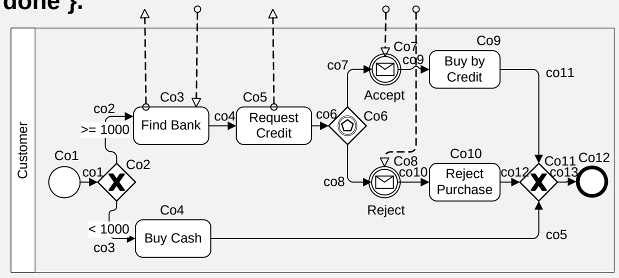






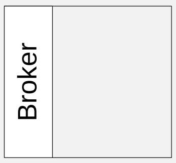


```
activity Co10 is abstraction( co10 : connection[], co12 : connection[] )
  behavior is { replicate via co10 receive. via shop::declinepurchase send.
    via co12 send. done }.
activity Co11 is abstraction( co5 : connection[], co11 : connection[], co12 : connection[],
co13 : connection[] )
  behavior is { replicate
    choose {  via co5 receive. via co13 send. done
    or        via co11 receive. via co13 send. done
    or        via co12 receive. via co13 send. done
    } }.
activity Co12 is abstraction( co13 : connection[] )
  behavior is { replicate via co13 receive. done }.
behavior is {
  -- Orchestration of the Activities
  connection co1 is connection(Real).
  connection co2 is connection(Real).
  connection co3 is connection(Real).
  connection co4 is connection(
    connection(Request), Real).
```



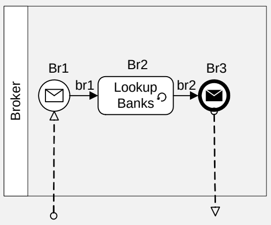
```
  connection co5 is connection(). connection co6 is connection(connection(),connection()).
  connection co7 is connection(). connection co8 is connection().
  connection co9 is connection(). connection co10 is connection().
  connection co11 is connection(). connection co12 is connection().
  connection co13 is connection().
  compose { Co1(co1, amount) and Co2(co1, co2, co3) and Co3(co2, co4, broker)
  and Co4(co3, co5) and Co5(co4, co6) and Co6(co6, co7, co8) and Co7(co7, co9)
  and Co8(co8, co10) and Co9(co9, co11) and Co10(co10, co12)
  and Co11(co5, co11, co12, co13) and Co12(co13) } } }
```

The *Broker*'s orchestration described in the *Broker*'s pool is translated to *π-ADL for WS-Composition* as follows.

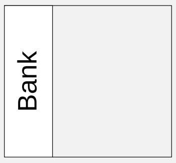


```
service Broker is abstraction(
broker : connection[connection[connection[Request]]],
banklist : connection[connection[Request]]) { type Amount is Real.
  type Accept is connection[]. type Reject is connection [].
  type Request is view[amount : Amount, accept : Accept, reject : Reject].
  port is { connection broker is connection(connectionToBank : connection[connection[Request]]).
    connection broker::connectionToBank is connection(connection(Request)).
    connection banklist is connection(connection(Request)) }.
  activity Br1 is abstraction( broker : connection[connection[connection[Request]]],
  br1 : connection[connection[connection[Request]]] )
    behavior is { replicate via broker receive connectionToBank : connection[connection[Request]].
      via br1 send connectionToBank. done }.
  activity Br2 is abstraction( br1 : connection[connection[connection[Request]]],
  banklist : connection[connection[Request]],
  br2 : connection[connection[connection[Request]], connection[Request]] )
    behavior is { via br1 receive connectionToBank : connection[connection[Request]].
      via banklist receive bank : connection[Request].
      via br2 send (connectionToBank, bank). behavior() }.
```

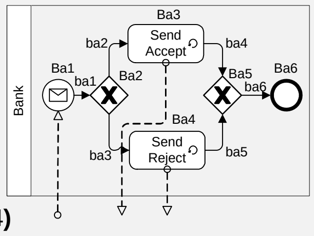
```
activity Br3 is abstraction(
br2 : connection[connection[connection[Request]], connection[Request]] )
  behavior is { replicate via br2 receive (connectionToBank, bank).
    via connectionToBank send bank. done }.
behavior is {
  -- Orchestration of the Activities
  connection br1 is connection(connection(connection(Request))).
  connection br2 is connection(connection(connection(Request)),
    connection(Request)).
  compose { Br1(broker, br1) and Br2(br1, banklist, br2) and Br3(br2) } } }
```



The *Bank*'s orchestration described in the *Bank*'s pool is translated to *π-ADL for WS-Composition* as follows.

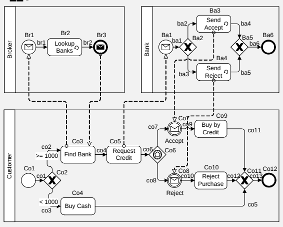


```
service Bank is abstraction( bank : connection[Request] ) { type Amount is Real.
  type Accept is connection[]. type Reject is connection [].
  type Request is view[amount : Amount, accept : Accept, reject : Reject].
  port is { connection bank is connection(request : Request).
    connection request::accept is connection(). connection request::reject is connection() }.
  activity Ba1 is abstraction( bank : connection[Request], ba1 : connection[Request] )
    behavior is { replicate via bank receive request : Request. via ba1 send request.
      behavior() }.
  activity Ba2 is abstraction( ba1 : connection[Request],
  ba2 : connection[Request], ba3 : connection[Request] )
    behavior is { replicate via ba1 receive request : Request. unobservable. -- Decision by the bank
      choose {  via ba2 send request. done
      or        via ba3 send request. done } }.
  activity Ba3 is abstraction(ba2 : connection[Request], ba4 : connection[] )
    behavior is { via ba2 receive request : Request.
      unobservable. via request::accept send. via ba4 send. behavior() }.
  activity Ba4 is abstraction( ba3 : connection[Request], ba5 : connection[] )
    behavior is { via ba3 receive request : Request.
      unobservable. via request::reject send. via ba5 send. behavior() }.
  activity Ba5 is abstraction( ba4 : connection[], ba5 : connection[], ba6 : connection[] )
    behavior is { replicate
      choose {  via ba4 receive. via ba6 send. done
      or        via ba5 receive. via ba6 send. done } }.
  activity Ba6 is abstraction( ba6 : connection[] )
    behavior is { replicate via ba6 receive. done }.
  behavior is { -- Orchestration of the Activities
    connection ba1 is connection(Request).
    connection ba2 is connection(Request).
    connection ba3 is connection(Request).
    connection ba4 is connection(). connection ba5 is connection().
    connection ba6 is connection().
    compose { Ba1(bank, ba1) and Ba2(ba1, ba2, ba3) and Ba3(ba2, ba4)
    and Ba4(ba3, ba5) and Ba5(ba4, ba5, ba6) and Ba6(ba6) } } }
```



The *overall* choreography shown in the message interactions among the three roles is translated to *π-ADL for WS-Composition* as follows. The customer dynamically binds to a *Bank* found by a *Broker* at the *banklist* connection if the purchase is above a threshold value, as detailed in the orchestrations.

```
architecture Purchasing is abstraction( banklist : connection[connection[Request]] )
  type Amount is Real. type Accept is connection[]. type Reject is connection [].
  type Request is view[amount : Amount, accept : Accept, reject : Reject].
  port is { connection banklist is connection(connection(Request)).
    connection purchasing is connection(shop : connection[view[buycash : connection[],
    buybycredit : connection[],declinepurchase : connection[]]], amount : Real).
    connection purchasing::shop is connection(view[buycash : connection[],
    buybycredit : connection[],declinepurchase : connection[]]) }.
  behavior is { -- Choreography of the Services
    connection broker is connection(connection(connection(Request))).
    compose { Broker(broker, banklist)
    and replicate connection bank is connection(Request). Bank(bank)
    and
      replicate via purchasing receive (shop : connection[view[buycash : connection[],
      buybycredit : connection[], declinepurchase : connection[]]], amount : Real).
      Customer(broker, shop, amount)
    } } }
```



4. Implementation and Experimentation

A major impetus behind developing formal languages for architectural description is that their formality renders them suitable to be manipulated by software tools. The usefulness of an ADL is thereby directly related to the kinds of tools it provides to support architectural description, but also analysis, refinement, code generation, and evolution [28]. Indeed, *π-ADL for WS-Composition* is supported by a comprehensive toolset for supporting service-oriented architecture formal development.

By defining *π-ADL for WS-Composition* as an embedded domain-specific language in π-ADL, it inherits from its core toolset and its customizable tools [28]. The resulting toolset is composed of:

- a visual modeling tool implemented as an extension of the Objecteering UML Modeler that itself supports a BPMN profile;
- a callable compiler and a persistent virtual machine for “running” architecture descriptions;
- a verification tool based on CADP, and XSB for checking architectural properties;
- a refinement tool providing a transformation framework based on the Maude rewriting logic system;
- a code synthesizer, including π-ADL-to-Web service code generation tools.

All tools supporting *π-ADL for WS-Composition* are integrated using an XML-based interchange language and web services. This comprehensive toolset is partially available as Open Source Software.

π-ADL has been applied in practice in several software projects in France, Italy, UK, Switzerland, and China. In particular, different features of *π-ADL for WS-Composition* and its supporting toolset have been applied at Thésame Inc. (France) for architecting and developing an agile business process management system for wide-area supply-chain management. It is based on a service-oriented architecture and the web service technology stack, comprising BPMN and WSDL. The π-ADL virtual machine is directly deployed as the business process execution engine.

This experimentation has shown that *π-ADL for WS-Composition* and its toolset are suitable for formally developing dynamic Web service compositions.

5. Related Work

Different ADLs have been proposed in the literature [11][8], including: AADL, ACME/Dynamic-ACME, AESOP, AML, ARMANI, CHAM-ADL, DARWIN, KOALA, META-H/AADL, PADL, PLASTIK, RAPIDE, SADL, σπ-SPACE, UNICON-2, WEAVES, WRIGHT/Dynamic-WRIGHT, and ZETA.

Most of these ADLs assume that architectures are static. Some however supports the description of dynamic features of architectures. They are Dynamic-ACME, DARWIN, Dynamic-WRIGHT, KOALA, PLASTIK, WEAVES, σπ-SPACE, and RAPIDE. But this support is rather limited, and in the case of Dynamic-ACME, PLASTIK and WEAVES, they are not formally defined. Overall, none of these ADLs have the expressive power to describe dynamic service-oriented architectures where connection mobility is needed in order to model the dynamic discovery and invocation of new discovered services.

Unlike these other ADLs, the support provided by *π-ADL for WS-Composition* is complete with respect to expressiveness of dynamic service-oriented architectures based on its underlying foundation, i.e. the typed π-calculus.

As a computer language, an ADL is defined by both a syntax and a formal, or semi-formal, semantics. Typically, ADLs embody a conceptual framework reflecting characteristics of the style or domain for which the ADL is intended.

The focus of *π-ADL for WS-Composition* is on the formal modeling of dynamic service-oriented architectures, and for supporting the computer-aided formal verification and refinement of these models.

π-ADL for WS-Composition introduces the notion of Web service and service composition as first-class citizens in architecture descriptions, which is not the case of other ADLs. Service compositions in terms of service orchestration and service choreography are formally specified in terms of typed abstractions over behaviors.

Furthermore, *π-ADL for WS-Composition* provides a notation to express assertions as properties of service-oriented architectures (not presented in this paper due to lack of space) based on π-AAL [10]. It combines predicate logic with temporal logic in order to allow the specification of both structural properties and behavioral properties. Regarding structural properties, the property notation is based on predicate logic. Regarding behavioral properties, it is based on modal μ-calculus Moreover, having a unified notation for expressing both structural and behavioral properties facilitates the specification task of architects, by allowing a more natural and concise description.

6. Concluding Remarks

In this paper, we have presented *π-ADL for WS-Composition*, a service-oriented architecture description language for the formal development of dynamic Web service compositions, including orchestration and choreography.

Instead of presenting its formal underpinnings and semantics, this paper introduced *π-ADL for WS-Composition* through the description of a dynamic service-oriented architecture of a purchase management system.

π-ADL for WS-Composition is used for modeling the structure and behavior of dynamic service-oriented architectures. It is defined as an embedded domain-specific language in π-ADL. It complements the π-ADL family of ADLs with a formal ADL for formally describing service-oriented architectures. It is itself user-defined and formally specified in terms of π-ADL. It is worth noting that the definition of the language itself is open. An architect can thereby tune *π-ADL for WS-Composition* for his/her specific purposes.

The use of BPMN as a visual notation for business process modeling assists in filling the gap between semi-formal process diagrams and formal service-oriented architecture descriptions, which can be analyzed and refined, by providing the ability to elaborate the architecture specification.

As stated in the introduction, enabling the specification of dynamic Web service-oriented architectures is a key challenge for an ADL. *π-ADL for WS-Composition* meets this threefold research challenge:

- It supports the description of dynamic service-oriented architectures from structural and behavioral viewpoints, in particular supporting the dynamic nature of web service discovery and invocation.
- It provides a unified foundation for describing the different service-oriented architecture artifacts: business processes, orchestrations, choreographies, and interfaces. Transformation models support the translation from BPMN to *π-ADL for WS-Composition* and from it to WS-BPEL and WS-CDL[6].
- It supports the formal description of service-oriented architectures enabling to rigorously reason about and verify their qualities.

Last but not least, *π-ADL for WS-Composition* bridges the gap between semi-formal languages such as BPMN and formal calculus of services such as COWS [7], SCC [1], or SOCK [4]. The former provides computer support without formal semantics, the later formal semantics without any computer support.

Future work is mainly related with the development of a service-oriented architecture formal method founded on [22]. This formal method, called the *π-Method for SOA*, like formal methods such as B, FOCUS, VDM, and Z, aims to provide full support for formal description and development. Unlike these formal methods that do not provide any architectural support, the *π-Method for SOA* has been built to support service-oriented architecture-centric formal software engineering.

References

[1] Boreale M. et al.: SCC: a Service Centered Calculus. Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM 2006), LNCS 4184, Springer, Vienna, Austria, September 2006.

[2] Chaudet C., Greenwood M., Oquendo F., Warboys B.: Architecture-Driven Software Engineering: Specifying, Generating, and Evolving Component-Based Software Systems. IEE Journal: Software Engineering, Vol. 147, No. 6, UK, December 2000.

[6] The definition and implementation of automated transformation models from *π-ADL for WS-Composition* to WS-BPEL and WS-CDL are in progress.

[3] Clements P. et al.: Documenting Software Architectures: Views and Beyond. Addison Wesley, 2003.
[4] Guidi C., Lucchi R., Gorrieri R., Busi N., Zavattaro G.: SOCK: A Calculus for Service Oriented Computing. Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC'06), LNCS 4294, Springer, Chicago, December 2006.
[5] Greenwood M., Balasubramaniam D., Cimpan S., Kirby N.C., Mickan K., Morrison R., Oquendo F., Robertson I., Seet W., Snowdon R., Warboys B., Zirintsis E.: Process Support for Evolving Active Architectures. LNCS 2786, Springer, September 2003.
[6] IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, October 2000.
[7] Lapadula A., Pugliese R., Tiezzi F.: A Calculus for Orchestration of Web Services. Proceedings of the 16th European Symposium on Programming (ESOP'07), LNCS 4421, Springer, Braga, Portugal, March 2007.
[8] Leymonerie F., Cimpan S., Oquendo F.: State of the Art on Architectural Styles: Classification and Comparison of Architecture Description Languages, Revue Génie Logiciel, No. 62, September 2002 (In French).
[9] Marcos E., Cuesta C.E., Oquendo F. (Eds.): Special Issue: Software Architecture. International Journal of Cooperative Information Systems (IJCIS), Vol. 16, No. 3/4, September/December 2007.
[10]Mateescu R., Oquendo F.: π-AAL: An Architecture Analysis Language for Formally Specifying and Verifying Structural and Behavioral Properties of Software Architectures. ACM SIGSOFT Software Engineering Notes, Vol. 31, No. 2, March 2006.
[11]Medvidovic N., Taylor R.: A Framework for Classifying and Comparing Architecture Description Languages. IEEE Transactions on Software Engineering, 2000.
[12]Medvidovic N., Dashofy E., Taylor R.: Moving Architectural Description from Under the Technology Lamppost. Information and Software Technology, Vol. 49, No. 1, 2007.
[13]Milner R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, 1999.
[14]Morrison R., Balasubramaniam D., Oquendo F., Warboys B., Greenwood M.: An Active Architecture Approach to Dynamic Systems Co-evolution. Proceedings of the 1st European Conference on Software Architecture (ECSA'07), Madrid, Spain, September 2007.
[15]Morrison R., Graham K., Balasubramaniam D., Mickan K., Oquendo F., Cimpan S., Warboys B., Snowdon B., Greenwood M.: Support for Evolving Software Architectures in the ArchWare ADL. Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), Oslo, Norway, June 2004.
[16]OASIS: Reference Model for Service Oriented Architecture, V. 1.0, OASIS Standard, October 2006, URL: http://docs.oasis-open.org/soa-rm/v1.0/.
[17]OASIS: Reference Architecture for Service Oriented Architecture, V. 1.0, OASIS Standard, 23 April 2008, URL: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.html.
[18]OMG: Business Process Modeling Notation Specification, OMG Final Adopted Specification, Ref. dtc/06-02-01, February 2006.
[19]Oquendo F.: π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software

Architectures. ACM SIGSOFT Software Engineering Notes, Vol. 28, No. 8, USA, May 2004.

[20]Oquendo F.: π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures. ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, September 2004.

[21]Oquendo F.: Formally Modelling Software Architectures with the UML 2.0 Profile for π-ADL. ACM SIGSOFT Software Engineering Notes, Vol. 31, No. 1, January 2006.

[22]Oquendo F.: π-Method: A Model-Driven Formal Method for Architecture-Centric Software Engineering. ACM SIGSOFT Software Engineering Notes, Vol. 31, No. 3, May 2006.

[23]Oquendo F.: Software Architectures. Encyclopédie de l'informatique et des systèmes d'information, Editions Vuibert, November 2006 (In French).

[24]Oquendo F. (Ed.): Proceedings of the European Conference on Software Architecture (ECSA'07). LNCS 4758, Springer, September 2007.

[25]Oquendo F.: Tutorial of the ArchWare Architecture Description Language, Deliverable D1.9, ArchWare European RTD Project, IST-2001-32360, June 2005.

[26]Oquendo F.: The π-Service Oriented Architecture Description Language: Abstract Syntax and Formal Semantics. VALORIA Technical Report, University of South Brittany, June 2008.

[27]Oquendo F., Alloui I., Cimpan S., Verjus H.: The ArchWare Architecture Description Language: Abstract Syntax and Formal Semantics. Deliverable D1.1b, ArchWare European RTD Project, IST-2001-32360, December 2002.

[28]Oquendo F., Warboys B., Morrison R., Dindeleux R., Gallo F., Garavel H., Occhipinti C.: ArchWare: Architecting Evolvable Software. LNCS 3047, Springer, May 2004.

[29]Peltz C.: Web Services Orchestration and Choreography. IEEE Computer, Vol. 36, No. 10, October 2003.

[30]Pourraz F.: Diapason: An Architecture-Centric Formal Approach for the Evolutionary Composition of Web Services, PhD Thesis, University of Savoie, France, December 2007 (In French).

[31]Puhlmann F.: A Unified Formal Foundation for Service Oriented Architectures. Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA'06), Hamburg, October 2006.

[32]Puhlmann F.: On the Application of a Theory for Mobile Systems to Business Process Management. PhD Thesis, University of Potsdam, Germany, 2007.

[33]Ratcliffe O., Scibile L., Cimpan S., Oquendo F.: Towards an Inductive Definition and Evolution of Architectural Styles, Proceedings of the 17th International Conference on Software and Systems Engineering and their Applications (ICSSEA'04), Paris, France, December 2004.

[34]van der Aalst W.M.P., ter Hofstede A.H.M., Dumas M.: Patterns of Process Modeling. In Process-Aware Information Systems: Bridging People and Software through Process Technology, Wiley & Sons, 2005.

[35]W3C: Web Services Architecture, W3C Working Group Note, February 2004, URL: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[36]W3C: Web Services Description Language (WSDL), V. 2.0, W3C Recommendation, 26 June 2007, URL: http://www.w3.org/TR/2007/REC-wsdl20-20070626.