Diapason: an Engineering Approach for Designing, Executing and Evolving Service-Oriented Architectures

Frédéric Pourraz and Hervé Verjus
University of Savoie - Polytech'Savoie - FRANCE
LISTIC - Language and Software Evolution Group (LS-LSE)
BP 80439, 74944 Annecy-le-Vieux Cedex
{herve.verjus;frederic.pourraz@univ-savoie.fr}

Abstract

Web services are often employed to create wide distributed evolvable applications from existing components that constitute a service-based software system. Service-Oriented Architectures promote loose coupling, services distribution, dynamicity and agility. As services involved in a SOA are remote and autonomous services, the SOA designer does not control them and unpredictable behaviour can occur. Services orchestration is a key issue in order to fit expectations and reach objectives. Thus, service-oriented architectures have to be designed and deployed with rigor in order to be plainly useful and quality aware. Orchestration languages (BPEL4WS, BPML, etc.) fail in some points due to the lack of formalization and expressiveness, particularly when addressing service-based architecture evolution. This paper presents Diapason, an engineering approach for fully and formally designing service-based architectures, deploying them on the Internet, executing them according to the design and dynamically evolving them taking into account business changes and modifications.

1 Introduction

Building a software-intensive system from existing software blocks of computation is not a novel idea: these blocks are sometimes called objects, sometimes they are called components, modules, etc. As the blocks are widely distributed accross the Internet, designing a software-intensive system from these blocks is not so easy. In the last ten years, huge amount of work has been dedicated to design and deploy software-intensive distributed systems [3]. Such systems are supported by software blocks that are often strongly integrated by using technologies that cannot easily support changes. But now, we speak about time to market, enterprise agility and software-intensive system adaptability, i.e. software intensive systems being able to react to business changes and modifications. Software intensive systems evolution is becoming a key issue [7, 2] in such context and SOA propose new perspectives [14]. Recent works focus on one hand on designing a system from a high level of abstraction in order to reason about it and to control it: software architecture field copes with such objectives [21]; on another hand, providing approaches and technologies for supporting software-systems evolution is also very challenging [10, 2].

One of the main interest of Service-Oriented Architectures [6, 14, 20] is basically the underlying ability of such architectures to inherently being evolvable; because the underlying idea of SOA is that the services (that can be defined as software functionality packages accessible through a networked infrastructure) are loosely coupled and the SOA could be adapted to its environment. P2P architectures illustrate such idea when, for example, a service is no more available and could be replaced dynamically by another (we will discuss in section 3 about the way of dynamically replacing/changing such services either by another service, either by the same service being modified). As P2P becomes more popular, B2B, B2C architectures are appearing and SOA is becoming a new way of building software-intensive systems, supporting automated activities that were traditionnaly only supported by software applications. Then, the same needs and questions we address to software applications are now addressed also to SOAs, taking into account SOA's characteristics: what about the quality of SOAs ? How can we ensure that SOA fit requirements, satisfy user's needs ? How are SOAs able to evolve according to changes ? How SOAs can be maintained over time ? How can we ensure that the executed SOA is consistent with the design ? But SOAs are not traditionnal software applications: services may be heterogeneous, widely distributed and are loosely coupled. Loose coupling is achieved through encapsulation and communication through message passing;

technology neutrality results from adopting standardized mechanisms; and rich interface languages permit the service to export sufficient information so that eventual clients can discover and connect to it [14]. SOA paradigm has *a substantial impact on the way software systems are developed* [6]. Thus, SOAs suggest new requirements and new desires. We investigate in proposing an approach for formally designing, checking, deploying and executing SOAs (service-based software applications) that deal with such questions and mentionned issues.
The section 2 of this paper will introduce a SOA-based scenario that will illustrate evolution requirements and will present some claims. Section 3 will present our approach, called Diapason, for designing and deploying Web-service-based architectures and will illustrate it through the scenario introduced in section 2. Section 4 will propose some related works, while section 5 will conclude.

2 An illustrating scenario

Let us consider a virtual print shop that proposes print functionality to clients. There are several and remote print servers that may respond to a client's request. Our purpose is to hide the print servers distribution to the client by providing a sole print service. Such service will orchestrate all print shop services in order to best satisfy the client. That consists of determining the best policy in order to respond to the client. Such policy could be defined taking account some criterias like print server availability, capability, networking time, etc. At a first glance, the print servers are able to provide the printing quantity in their print queues. We are defining a simple orchestration policy when a client's request occurs; such policy consists of selecting the print server that has the smallest printing quantity. The client's printing resquest it sent to the selected print server. The system implements such policy. Let us now imagine that the policy is changing (whatever the reasons are) in order to take into account print servers unaivalability. When a print server is becoming suspended or unavailable, its associated print service does not more provide the printing quantity in the print server queue but provides instead a negative printing quantity (with the -1 as value). The orchestration has now to deal with a -1 printing quantity's value. We are now considering in our scenario that one of the two print servers is becoming suspended or unavailable; then, when a client's request occurs, the choosen print server is the sole that is currently available.

We only will concentrate on the load-balancing process (see Figure 1) among print servers in order to define the best-suited policy. This virtual print shop will be implemented as a SOA. The architecture basically comprises two print services (that are print servers's proxies providing operations like getQuantity, sendPrintJob). The scenario is

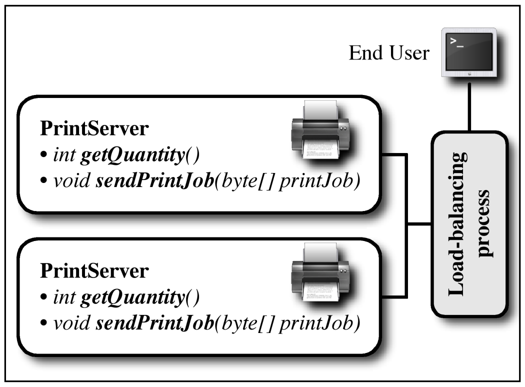

Figure 1. The virtual print shop architecture

illustrating an SOA that will change over time: the print servers will be modified by adding a new operation that provides print server's status.
As we consider services as black boxes (we only have their API describing the operations each service provides, i.e. the WSDL files), we do not control how services are implemented, when and how they can be changed, maintained over time.

In the following, we propose a formal-based engineering approach named Diapason, for formalizing, deploying, executing and evolving services-orchestrations.

3 Diapason: a SOA-based systems formal engineering approach

In [6] the authors present challenges in SOA engineering domain that encompass requirements, architecture, design, implementation, testing, deployment and reengineering. They identified evolution of SOA as important issue; thus, novel approaches have to be proposed.

Our approach aims at addressing important issues enumerated in [14, 6, 20]:

- a services orchestration formal language allowing to define and reason on evolvable services orchestrations. Such language has to be dedicated to SOA domain experts and has to be as simple as possible in order to be useful for SOA designer;
- services orchestration checking against some properties expressed by using π-Diapason and logic-Diapason;
- services orchestration deployment and execution environment conform to the services orchestration definition;
- services orchestration dynamic runtime evolution with formal built-in checking mechanisms.

3.1 Diapason foundations

Diapason is based on our works in software architecture domain [17, 16] particularly under the scope of architecture evolution from a high level of abstraction [24, 2]. Software architecture encompasses software elements and their relationships at different level of abstraction (very abstract level, also called conceptual, and concrete level that is very close to the code).
The enthusiasm around the development of formal languages for architecture description comes from the fact that such formalisms are suitable for automated handling and models formal reasoning, properties checking [20, 17]. These languages are used to formalize the architecture description as well as its refinement. The benefits of using such an approach are manifold. They rank from the increment of architecture comprehension among the persons involved in a project (due to the use of an unambiguous language), to the reuse at the design phase (design elements are reused) and to the property description and analysis (properties of the future system can be specified and the architecture analyzed for validation purpose). Once the information system architecture has been identified and formalized, the architect may reason on it [20].
Several ADLs were proposed [9] that mainly focused on architecture design, at a high level of abstraction. In such context, managing the gap between abstract level and implementation level remains an issue. Our approach does not distinguish both levels (at the opposite of [20]) but proposes instead to unify design (abstract) and implementation (concrete) by considering relevant services orchestration abstractions and by providing behaviour expression and execution mechanisms. Thus, using Diapason, the SOA designed is also the one that will be executed. Diapason combines strengths of formal and enactable process algebra-based languages that support dynamic and evolvable software architectures [24, 2] with services orchestration purposes, concepts and abstractions [25, 15]. The relevance of using process algebras in services compositions has been already claimed and justified [20]. As proposed in [23], several workflow patterns have been identified in order to define complete and executable worklows. The Diapason approach addresses some challenges identified in [6].

We will focus on runtime dynamic maintenance and evolution of SOA in the following that remain an important issue [14].

3.2 Services orchestration using π-Diapason

Diapason is a π-calculus based approach allowing formal services based systems modeling. The aim of using a process algebra (which formally models interactions between processes [20]) as a fundament is to provide a mathematical model in order to guarantee the software conformance with the end-user's requirements. In other words, thanks to a mathematical description, a services based system description can be proven. Different process algebras have been provided, for example CSP [5], CCS [11], π-calculus [12], etc.. In our case, we have adopted the π-calculus due to its main particularity: the process mobility. This concept allows us to dynamically evolve SOAs by the way of processes exchanges. In the case of services orchestration, processes (i.e. orchestrations) is formally defined in π-calculus terms of behaviours and channels. A channel aims at connecting two behaviours and lets them interacting together. The first order π-calculus has a restricted policy according to the type of informations which can be transited over a channel. Only simple data or channel can be transmitted but a behaviour cannot. Transiting a channel reference over another channel provides a way, for a process A, which has got a channel with a process B and another channel with a process C, to send, for example to B, its channel with C. Finally, both processes B and C which were not able to communicate as far for now, can now communicate with a common channel. This his the first kind of mobility. In our case, Diapason is based on the high order π-calculus which is more powerful. In addition to the first kind of mobility, high order π-calculus let channels to exchange channels as well as behaviours. This brings a more powerful mobility. In this way, a behaviour can send (via a channel) a behaviour to another behaviour. The transmitted behaviour could be executed by the behaviour's receiver. Thus, this latter may be dynamically inherently modified by the behaviour it just has received.

Diapason provides two different languages and a virtual machine. The first language called π-Diapason lets us formally describe an SOA. Such SOA will be then deployed as a Web Service Oriented Architecture (WSOA). The second language called Logic-Diapason lets us express some SOA properties. π-Diapason aims at avoiding refinement steps in architecture-centric approaches [13, 24, 2]. This can be done by proposing well defined abstractions for the services orchestration as well as a runtime environment that supports the π-Diapason language. π-Diapason is a powerful language:

- it allows the SOA architect to design and specify SOAs (focusing on services orchestration);
- it provides Domain Specific Layer (see below) in order to simplfy SOA design;
- it is formally defined, based on π-calculus;
- it supports dynamic SOA evolution (focusing on ser-

vices orchestration dynamic evolution);

- it is enactable: it is powerful and complete enough, supporting behaviour expression that a virtual machine interpretes it.

Thus, there is no gap between design (abstract level) and implementation (concrete level) as it is the same language that covers both levels. There is no need for mappings rules, no need for consistency management (at the opposite of what is proposed in [20, 4]). The SOA specified will be the one that will be interpreted. SOA's execution is precisely carried out by the Diapason virtual machine; this latter can be used for SOA simulation and validation purpose and/or for runtime engine that interpretes services orchestration expressed in π-Diapason (see section 3.4).

π-Diapason is a layered language which provides three abstraction levels.

The first layer is the expression of the high order, typed, asynchronous and polyadic π-calculus [12]. This layer lets us to express any process (i.e. π-calculus behaviours) in terms of:

- μ.P: the prefix of a process by an action where μ can be :
  - $x(y)$: a positive prefix, which means the receiving event of the variable y on the channel x,
  - $\bar{x}$y: a negative prefix, which means the sending event of the variable y on the channel x,
  - τ: a silent prefix, which means an unobservable action,
- $P|Q$: the parallelisation of two processes,
- $P + Q$: the indeterministic choice between two processes,
- $[x = y]P$: the matching expression,
- A(x1, ..., xn) $\overset{def}{=}$ P: the process definition which allows to express the recursion.

The second layer is defined on top of the first layer, using the first layer language. This second abstraction level is the expression of the previously mentioned workflow patterns: it is itself a formal process pattern definition language. The twenty first patterns proposed in [23] are currently described in this layer; the recent twenty new ones introduced in [18] will be expressed soon. This second layer lets us to describe any complex process in an easiest way, than only using the first layer (π-calculus definition layer that is less intuitive). Using this second layer language, the user is now able to define recurrent structures that will serve as language extensions and will be reused in other process pattern definitions. We have currently expressed some patterns in order to provide a first library but, as we mentioned, any other structure can be described using this layer. Let us take the example of the synchronization pattern, called *synchronize*. As we will see in some following examples, a *synchronize* pattern allows to merge different parallelized processes. Expressed using the first layer, the *synchronize* pattern description is the following:

```
pattern(synchronize(connections(_connections)),
  iterate( _connections,
           iterator(_connection),
           behaviour(receive(_connection, _values)))).
```

The *synchronize* pattern takes a list of connections (i.e channels in π-calculus) as parameters. The length of the list corresponds to the number of parallelized processes. Once applied, this pattern will use the *iterate* behaviour (not detailed in this paper) provided by the first layer. The *iterate* behaviour takes three parameters: a list (on which one will iterate), the iteration variable and a behaviour which will be applied for each iteration. Thanks to the *synchronize* pattern, the *iterate* pattern is used as follows: the list passed as parameter is a list of connections; thus, the iteration variable is a connection (of the list); the behaviour is defined as a receiving action attempt on the current connection (the iteration variable value). When the *iterate* pattern is terminated (i.e. all of the connections involved have received any value), the orchestration process goes on to the next steps.

The third layer is a domain specific layer. Thanks to the SOA domain, this layer provides the end user language for the expression of Web Services Oriented Architectures. This third abstraction level is defined and expressed by using the two previous layer; thus, a WSOA expressed in this third level language is directly expressed as a π-calculus process. This layer lets us to describe:

- the behaviour of a services orchestration,
- the orchestration inputs and outputs,
- the complex types manipulated and required in such services orchestration,
- operations of all of the services involved in the orchestration.

We are now illustrating the π-Diapason language by expressing the virtual print shop scenario described in section 2:

- We are creating a new services orchestration called ”VirtualPrintShop” with a sole parameter (as input) of type ”arrayOfByte” and with a variable named _*printJob* (all variables are prefixed with an underscore).

```
orchestration(
   name('VirtualPrintShop'),
   parameters([_printJob], [arrayOfByte]),
   ...
```

- Then, we are defining (i) the complex types needed by all the Web services operations involved in the orchestration, none in this scenario, and (ii) the operations definition. The operation definition includes the operation's name, its Web service parent, the URL, input and output parameters.

```
...
complex_types([]),
operations([
  operation( name('getQuantity'),
    service('PrintServer_1'),
    url('http://print-server-1/'),
    requests([]),
    response(name('quantity'), type('int'))),
operation( name('sendPrintJob'),
    service('PrintServer_1'),
    url('http://print-server-1/'),
    requests([request(name('printJob'), type('
         arrayOfByte'))]),
    response(_)),
operation( name('getQuantity'),
    service('PrintServer_2'),
    url('http://print-server-2/'),
    requests([]),
    response(name('quantity'), type('int'))),
operation( name('sendPrintJob'),
    service('PrintServer_2'),
    url('http://print-server-2/'),
    requests([request(name('printJob'), type('
         arrayOfByte'))]),
    response(_))]),
...
```

- The orchestration behaviour can thirdly be described. It consists of scheduling the Web services operations invocations by the way of process patterns (sequence, parallel, conditional expressions).

```
...
behaviour(
 parallel_split([
  sequence(
    apply(
      invoke( operation('getQuantity'),
              service('PrintServer_1'),
              requests([]),
              response(_quantity_1))),
      send(connection('print server 1'), values([]))),
  sequence(
    apply(
      invoke( operation('getQuantity'),
              service('PrintServer_2'),
              requests([]),
              response(_quantity_2))),
      send(connection('print server 2'), values([]))),
  sequence(
    apply(
       synchronize(connections([connection('print
            server 1'), connection('print server 2')]))
            ),
```

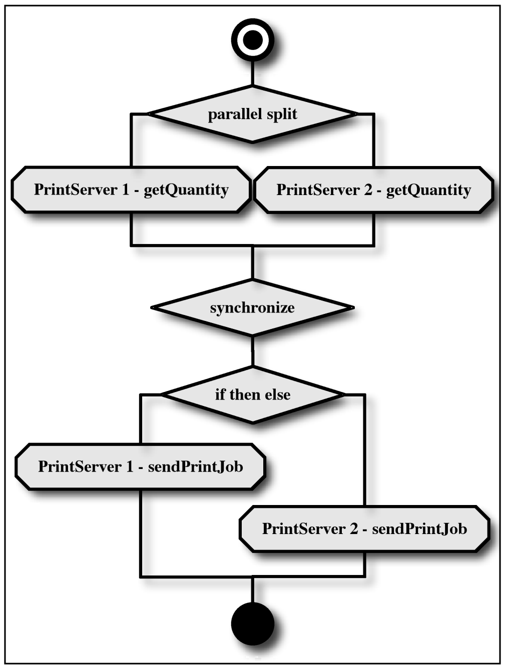
parallel split
PrintServer 1 - getQuantity
PrintServer 2 - getQuantity
synchronize
if then else
PrintServer 1 - sendPrintJob
PrintServer 2 - sendPrintJob

Figure 2. The virtual print shop services orchestration

```
   sequence(if_then_else(_quantity_1 < _quantity_2,
     apply(
       invoke( operation('sendPrintJob'),
               service('PrintServer_1'),
               requests([value(_printJob)]),
               response(_)),
       apply(
         invoke( operation('sendPrintJob'),
                 service('PrintServer_2'),
                 requests([value(_printJob)]),
                 response(_))),
    terminate))])),
...
```

- Finally, we are adding the returned parameter in terms of type and variable name; none in this example.

```
...
return(_) ).
```

3.3 Services orchestration dynamic evolution: orchestration changes principles and mechanisms

Thanks to the π-calculus mobility (first order but extended to behaviour mobility support in the high order), we may modify the services orchestration dynamically, at runtime, without to stop this orchestration being executed. By construction and due to the layered languages we propose, a services orchestration expressed using the third layer language is semantically and formally defined as a π-calculus process (in term of the first layer language). Evolving a services orchestration is quite as the same as evolving a π-calculus process. We offer two different ways of performing services orchestration dynamic evolution:

- the first one (external evolution) is decided on the services orchestration provider in order to maintain it (i.e. adding, removing, changing functionalities);
- the second one (internal evolution) is fired by the services orchestration itself in order to announce a bug or to request modification(s) when orchestration fails. In this case, the orchestration π-Diapason definition integrates the evolution code.

```
...
 behaviour(
  parallel_split([
    // An external evolution may be requested
    receive(connection('EVOLVE'), values([
         _evolved_behaviour]))
      parallel_split([
       sequence(
         apply(
           invoke( operation('getQuantity'),
                   service('PrintServer_1'),
                   requests([]),
                   response(_quantity_1))),
           send(connection('print server 1'), values([])
                )),
       sequence(
         apply(
           invoke( operation('getQuantity'),
                   service('PrintServer_2'),
                   requests([]),
                   response(_quantity_2))),
           send(connection('print server 2'), values([])
                )),
       sequence(
         apply(
           synchronize(connections([connection('print
                server 1'), connection('print server 2')
                ]))),
       sequence(if_then_else(_evolved_behaviour != NULL,
                  // Evolution Required
                  apply(_evolved_behaviour)
                  // NO Evolution Required
                  if_then_else(_quantity_1 < _quantity_2
                     ,
                     apply(invoke( operation('
                          sendPrintJob'),
                                   service('
                                        PrintServer_1'),
                                   requests([value(
                                        _printJob))]),
                                   response(_)),
                     apply(invoke( operation('
                          sendPrintJob'),
                                   service('
                                        PrintServer_2'),
                                   requests([value(
                                        _printJob))]),
                                   response(_)))),
        terminate))]])])),
  ...
```

To perform the external evolution, some changes are required in the orchestration π-Diapason description. These changes are supported by some specific π-Diapason code structures inside the behaviour (see the code previously shown). Thus, an "evolution point" has been added. A connection called "EVOLVE" in the code, is always available during the entire services orchestration lifecycle. This connection allows us to dynamically pass a behaviour to the orchestration. Once received (the *_evolved_behaviour* variable is becoming not null), this behaviour can be applied within the orchestration. Such behaviour application modifies dynamically the orchestration according to the behaviour's π-Diapason definition that integrates changes. Otherwise, when no behaviour is received, the orchestration process goes on without modification. Thanks to our illustrating scenario, the behaviour integrating changes that correspond to the evolution scenario presented in section 2 is the *_evolved_behaviour* variable's value:

```
behaviour(
  if_then_else( (_quantity_1 != -1 , _quantity_2 != -1)
     // Case 1
     if_then_else(_quantity_1 < _quantity_2,
        apply(invoke( operation('sendPrintJob'),
                      service('PrintServer_1'),
                      requests([value(_printJob))]),
                      response(_)),
        apply(invoke( operation('sendPrintJob'),
                      service('PrintServer_2'),
                      requests([value(_printJob))]),
                      response(_))),
     if_then_else( (_quantity_1 != -1 , _quantity_2 ==
          -1)
        // Case 2
        apply(invoke( operation('sendPrintJob'),
                      service('PrintServer_1'),
                      requests([value(_printJob))]),
                      response(_)),
        if_then_else( (_quantity_1 == -1 , _quantity_2
             != -1)
           // Case 3
           apply(invoke( operation('sendPrintJob'),
                         service('PrintServer_2'),
                         requests([value(_printJob))]),
                         response(_)),
           // Case 4
           // Evolution request by the process itself
           sequence( send(connection('EVOLVE'), values
                ([])),
           sequence( receive(connection('EVOLVE'),
                values([_evolved_behaviour])),
                     apply(_evolved_behaviour)))))))
```

This behaviour definition expressed in π-Diapason takes into account the status of both print servers by checking if the return value equals to "-1" (in this case the print server is in a "suspended" or "unavailable" state). If none of them is suspended (see the "Case 1" comment in the code above), the default orchestration policy remains unchanged: the print job is sent to the print server that is the less loaded. Otherwise, if one of both print servers are suspended (see the "Case 2" and "Case 3" comments in the code above), the selected print server will be the only one available, even if its loading threshold has been already raised. When all of the print servers are unavailable (see the "Case 4" comment in the code above), we are illustrating the internal evolution strategy. This latter is fired by the orchestration itself (see the code following the "Case 4" comment): when all print servers are unavailable, the orchestration definition expressed in π-Diapason does not contain the policy to apply (i.e. it is an unpredictable situation). In such situation, the new policy (containing the changes) has to be on the fly defined in a π-Diapason behaviour and such definition is dynamically sent to the connection named "EVOLVE". Once the behaviour has been received, it is dynamically applied at runtime (as already explained); we can imagine to add a new print server, to send a delay before processing the request, etc.

3.4 Orchestration checking, deployment and execution

When services orchestration has been defined using π-Diapason, the Diapason virtual machine is used in order to achieve two goals. The first one is the simulation before execution (the validation) and the second one is the execution itself. Simulation provide a way to compute all possible execution traces of an orchestration expressed in π-Diapason. Such traces are then analyzed against defined properties using the logic-Diapason language (this properties definition language is not detailed in this paper). Generics properties can be proved, like deadlock free, liveness properties and safety properties [20]. In the same way, logic-Diapason lets us define and check well suited properties to prove that a behaviour can or cannot appear during the execution of a specific orchestration. According to these verifications, it is up to the architect to validate and to decide whether or not the π-Diapason expressed orchestration can be deployed or not yet. In a positive case, the entire orchestration is deployed as a new Web service in order to easily be invoked and, for example, to be reused in another orchestration (we can also define orchestrations compositions - i.e. orchestrations that compose other orchestrations). Finally, the new web service deployed is executed thanks to our Diapason virtual machine. This web services embeddeds the π-Diapason orchestration description and the Diapason virtual machine. The Diapason virtual machine (π-Diapason interpreter) has been implemented using XSB [19]. When services orchestration has to dynamically evolve, vitrual machine computes again execution traces taking into account changes; traces are then analyzed against properties definition. Using properties analysis, it is up to the architect to validate and to decide whether or not changes have to be really applied on the current architecture. Changes may be applied on the fly, at runtime, without to stop the current services orchestration execution. The evolution mechanisms are explained in section 3.3.

4 Related work

Works arround services composition are manifold. They rank from services choreography, to services orchestration [15]. Basically, services choreography focuses on messages between actors (even they are not really identified) involved in business processes. Services choreography brings an abstract view of process interactions but does not aim at focusing on process execution. Services orchestration addresses business process through services invocations scheduling and organisation. Services orchestration aims at defining executable processes by providing orchestration languages (amongst the most well known BPEL4WS [1, 25], XLANG [22], WSFL [8], BPML, etc. [15]) that are executable languages (by the way of workflow engines). BPEL4WS allows to define abstract business processes and executable processes. But such languages lack in services orchestration reasoning, reuse, dynamic maintenance and evolution [14]: i.e. business processes expressed using these languages cannot be formally checked, nor they can evolve dynamically. When services are modified, the orchestration has to be manually modified accordingly and process execution cannot be dynamically changed. [20] presents a framework for the use of process algebra in web services compositions. The authors distinguish two layers: an abstract layer for which process algebras can be used and a concrete layer using classical services description, orchestration and choreography languages (WSDL, BPEL4WS, WS-CDL). Services are implemented with programming languages (Java, C#,...). The abstract layer allows the designer to reson on services compositions before translating such formal compositions to semi-formal ones. The formal mapping between the two layers deals with the semantic consistency between the layers as the executable layer is less powerful than the abstract layer. In [4], the authors present a framework where BPEL specifications of web services are translated to an intermediate representation, followed by the translation of the intermediate representation to a verification language. As an intermediate representation the authors use guarded automata augmented with unbounded queues for incoming messages, where the guards are expressed as XPath expressions. As the target verification language the authors use Promela, input language of the model checker SPIN. As consequence of both approaches ([20, 4]), we cannot guarantee that the implemented services orchestration will be totally compliant with the designed one. As the authors promote services orchestration languages such as BPEL4WS, there is no novel approach for services orchestration deployment and enactment and Diapason brings a significant contribution.

5 Conclusion

Diapason is a novel approach for formally define, deploy, execute and maintain services orchestrations. We are insisting on the evolution mechanisms in this paper. Formal foundations of the π-Diapason language can be found in [16]. π-Diapason supports on the fly services orchestration changes by employing high order π-calculus mobility concept: all or part of an orchestration definition (called a fragment) can be provided to the current executing orchestration on one of its channels (in terms of π-calculus). This fragment definition (a behaviour) is then applied within the evolvable orchestration. Thus, the services orchestration is internally modified according to the fragment and the current execution may be deeply modified. We are now focusing on SOA quality attributes expressions (using the logic-Diapason language) and we are investigating changes impacts analysis in order to improve checking toolkit.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. Specifications, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.

[2] S. Cîmpan, H. Verjus, and I. Alloui. Dynamic architecture based evolution of enterprise information systems. In *International Conference on Enterprise Information Systems (ICEIS)*, 2007.

[3] L. Davis, R. Gamble, M. Hepner, and M. Kelkar. Toward formalizing service integration glue code. In *IEEE International Conference on Services Computing*, 2005.

[4] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In A. Press, editor, *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, USA, 2004.

[5] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[6] K. Kontogiannis, G. A. Lewis, and D. B. Smith. The landscape of service-oriented systems: A research perspective. In *Proceedings of International Workshop on Systems Development in SOA Environments*, 2006.

[7] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[8] F. Leymann. Web services flow language (wsfl 1.0).

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

[10] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 123–131. IEEE Computer Society, 2005.

[11] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[12] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[13] F. Oquendo, I. Alloui, S. Cimpan, and H. Verjus. The archware adl: Definition of the abstract syntax and formal semantics. Deliverable D1.1b, ArchWare Consortium, ArchWare European RTD Project IST-2001-32360, 2002.

[14] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing: A research roadmap. In F. Cubera, B. J. Krämer, and M. P. Papazoglou, editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. http://drops.dagstuhl.de/opus/volltexte/2006/524 [date of citation: 2006-01-01].

[15] C. Peltz. Web services orchestration: A review of emerging technologies, tools, and standards.

[16] F. Pourraz and H. Verjus. π-diapason: un langage pour la formalisation des architectures orientées services web. In *1ère Conférence francophone sur les Architectures Logicielles (CAL 2006)*, pages 119–127, Nantes, September 2006.

[17] F. Pourraz, H. Verjus, and F. Oquendo. An architecture-centric approach for managing the evolution of eai services-oriented architecture. In *Eighth International Conference on Enterprise Information Systems (ICEIS 2006)*, pages 234–241, Paphos, Cyprus, May 2006.

[18] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPM Center Report BPM-06-22 , BPMcenter.org, 2006.

[19] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, S. Dawson, and M. Kifer. The xsb system version 3.0 volume 1: Programmer's manual. Technical report, XSB consortium, 2006.

[20] G. Salaün, L. Bordeaux, M. S. L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS*, pages 43–50. IEEE Computer Society, 2004.

[21] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994.

[22] S. Thatte. Xlang - web services for business process design.

[23] W. H. M. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases, 14(3)*, 2003.

[24] H. Verjus, S. Cimpan, I. Alloui, and F. Oquendo. Gestion des architectures évolutives dans archware. In *1ère Conférence francophone sur les Architectures Logicielles (CAL 2006)*, pages 41–57, Nantes, September 2006.

[25] S. Weerawarana and C. Francisco. Business processes: Understanding bpel4ws, part 1. IBM developerWorks, 2002.