A Formal Framework For Building, Checking And Evolving Service Oriented Architectures

Hervé Verjus and Frédéric Pourraz
University of Savoie - Polytech'Savoie
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. Services-Oriented Architectures promote loose coupling, services distribution, dynamicity and agility and introduce new engineering issues. 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, analized 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 maintenance and evolution. This paper presents Diapason, a formal framework that allows us to formally support SOA design, ckecking, execution and evolution.

1 Introduction

Service-Oriented Architectures (SOA) is a recent paradigm for building large scale software applications from distributed services. One of the main interest of SOA [20, 14, 27] is basically the underlying ability of such architecture 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. As services are supposed to be autonomous, self-contained, one have no control nor authority over them. P2P architectures illustrate such idea when, for example, a service is no more available and could be replaced dynamically by another. Thus, SOAs introduce new engineering issues [20, 12, 8, 21] and SOA evolution is becoming very challenging [20, 21]. In that perspective, recent works focus on designing evolvable and quality aware systems from a high level of abstraction in order to reason about it and to control it: software architecture field copes with such objectives [11, 6, 17, 16]. Thus, important engineering questions are addressed to SOAs architects: what about the quality of SOAs ? How can we ensure that SOA fit expectations ? How are SOAs able to be dynamically adapted ? 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 [20]. SOA paradigm has *a substantial impact on the way software systems are developed* [14]. Thus, SOAs suggest new requirements and new desires. We present in this paper our formal framework called Diapason that lets the user to formally express and check web services orchestrations that are able to be dynamically adapted in a formal and controled manner.

The section 2 of this paper will present works and challenges related to SOA engineering. Section 3 will introduce a virtual print shop scenario that will illustrate our approach. Section 4 will present our formal language, called π-Diapason, for designing evolvable Web-service-based architectures. Section 5 will address services orchestration property definition and analysis; Section 6 will show how our toolkit supports services orchestration deployment and execution while section 7 will conclude.

2 Related work and challenges

In [14] the authors present challenges in SOA engineering domain that encompass requirements, architecture, design, implementation, testing, deployment and reengineering. The authors mentionned, amongst other, the following issues:

- thanks to the architecture: services-oriented frameworks, platform-independent architectural styles, non-functional-attribute-driven design;
- thanks to the design: design pattern, platform-specific models, personalization and adaptation, services choreography and orchestration;
- thanks to the implementation: model-driven approaches, template-based code generation, language extensions to support service-oriented development, transformation frameworks;
- thanks to the testing: architecture-level: proof-of-concept, transaction management, quality of service, load/stress testing, global-level dynamic: composition, orchestration, versioning, monitoring, and regression testing;
- thanks to maintenance and reengineering: evolution patterns, dependency and impact analysis, infrastructures for change control and management, tools, techniques and environments to support maintenance activities, multilanguage system analysis and maintenance, reengineering processes, tools for the verification and validation of compliance with constraints, round-trip engineering.

Related to this, there are many additional opportunities that our approach will deal with:

- Languages for services orchestration and composition
- Reasoning about services compositions
- Integration by non-experts
- Orchestrations (fragments) reuse
- Services orchestration maintenance and evolution support

Works arround services composition are manifold. They range from services choreography, to services orchestration [22]. 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 [3], XLANG [29], WSFL [15], BPML, etc. [22]) 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 [21, 24]: 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. [27] 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 [10], 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 [1]. Some other approaches are similar in translating BPEL to a more formal language in order to perform some verification tasks [9, 28, 4]. Such translation steps (forward and/or backward) introduce information loss as the languages are neither semantically nor powerfull equivalent. Thus, 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. Thanks to SOA dynamic evolution purpose, the required translation steps from a semi-formal language to a formal one has no sense. Our approach is novel and improves the existing proposals in many points we will present in this paper. We now introduce a virtual print shop scenario that will illustrate our approach; more precisely, this scenario will be formalized using our services orchestration formal language π-Diapason and our property definition language called logic-Diapason.

3 An illustrating scenario

Let us consider a virtual print shop that proposes print functionalities 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 in determining the best policy in order to respond to the client's request. Such policy could be defined taking account some criterias like print servers availability, capability, networking time, etc. At a first glance, the print servers are able to provide the printing quantity in their print queues. When a client's request occurs, we are defining a simple orchestration policy that consists in selecting the print server that has the smallest print quantity. The client's print resquest it sent to the selected print server. The system implements such policy. Let us now imagine that policy is changing (whatever the reasons are) in order to take into account the case of a print quantity equals to "-1". In this case the print services do not only provide the printing quantity their are dealing with, but also they are now providing the print server status: "suspended" if the printing quantity equals to "-1". We now consider in our scenarios, that one on the two print servers's status is becoming "suspended", or when a server is no more available (network failure, etc.), the choosen print server is the sole that is currently available.

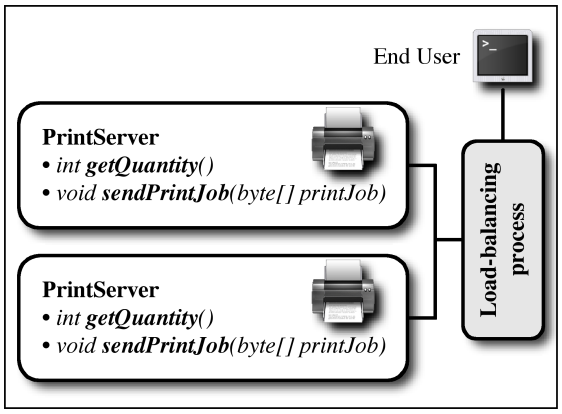

Figure 1. The virtual print shop architecture

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 illustrating an SOA that will change over time, depending on the print servers's availability and status.

Part of the WSDL print services definition is given in the following:

```
<wsdl:definitions>
   ...
   <wsdl:message name="getQuantityRequest"/>
   <wsdl:message name="getQuantityResponse">
      <wsdl:part name="quantity" type="soapenc:int"/>
   </wsdl:message>
   <wsdl:message name="sendPrintJobRequest">
      <wsdl:part name="printJob" type="soapenc:byte[]"/>
   </wsdl:message>
   <wsdl:message name="sendPrintJobResponse"/>
   <wsdl:portType name="PrintServer_PortType">
     <wsdl:operation name="getQuantity">
       <wsdl:input message="impl:getQuantityRequest"/>
       <wsdl:output message="impl:getQuantityResponse"/>
     </wsdl:operation>
     <wsdl:operation name="sendPrintJob">
       <wsdl:input message="impl:sendPrintJobRequest"/>
       <wsdl:output message="impl:sendPrintJobResponse
            "/>
     </wsdl:operation>
   </wsdl:portType>
   ...
</wsdl:definitions>
```

Such WSDL services definition extract does not contain $< wsdl : service >$ and $< wsdl : binding >$ tags for simplication and code consiness purpose. Remember that these services are supposed to be widely distributed and would be available on the Internet (their URLs could be known). As we consider services as black boxes (we only have their API describing the operations each service provides), we do not control how services are implemented, when and how they can be changed, maintained over time. Maintainability of a SOA is a key issue.

4 π-Diapason: a π-calculus-based language for expressing evolvable Web services orchestrations

4.1 π-Diapason formal foundations

Diapason is a formal framework allowing formal services based systems modeling, verification, deployment and execution. The aim of using a process algebra (which formally models interactions between processes [27]) as a fundament is to provide a mathematical model in order to guarantee the software conformance with the end-user's requirements. Diapason combines strengths of formal and enactable process algebra-based languages that support dynamic and evolvable software architectures [5, 6] with services orchestration purposes, concepts and abstractions [22]. The relevance of using process algebras in services compositions has been already claimed and justified [7, 27]. 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 [13], CCS [18], π-calculus [19], etc. In our case, we have adopted the π-calculus due to its main feature: the

process mobility. This concept allows us to dynamically evolve application's topology 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 in never way a behaviour. 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, the processes B and C which are 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, π-Diapason is based on the high order π-calculus which is more powerful. In addition to the first kind of mobility, hight 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.
π-Diapason aims at proposing well defined formal abstractions for expressing services orchestration that can be then executed (because π-Diapason is an executable formal language); π-Diapason:

- allows the SOA architect to design and specify SOAs (focusing on services orchestration);
- provides Domain Specific Layer (see below) in order to simplify SOA design;
- is formally defined, based on π-calculus;
- supports dynamic SOA evolution (focusing on services orchestration dynamic evolution);
- it is executable: it is powerful and expressive enough that a virtual machine can interprete 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 mappings rules, no need to consistency management. The SOA specified will be the one that will be interpreted. SOA's execution is precisely carried out by the π-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 5).

4.2 π-calculus basis for the π-Diapason language

Thanks to the π-calculus [18], some operators are required (we defined naming conventions: upper case letters stand for process while lower case letters stand for variables):

- 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) $\stackrel{def}{=}$ P: the process definition which allows to express the recursion.

Starting for the 0 process (i.e. the inactive process), the definition of a P process can be expressed as follows:
P $\stackrel{def}{=}$ 0 — x(y).P — $\bar{x}$y.P — τ.P — P1—P2 — P1+P2 — [x = y]P — A (x1, ..., xn)

π-Diapason has been designed as a layered language which provides three abstraction levels.

4.3 The first layer

The π-Diapason first layer is the expression of the high order, typed, asynchronous and polyadic π-calculus [19]:

- polyadic for sending simultaneously several values on a same channel (i.e. in order to invoke a service with some parameters),
- asynchonous; thus a process that sends a value on a channel is not blocked event if the receiver is not ready to proceed the receiving action,
- typed for allowing typed value declaration. Thus, type checking is then possible,
- high order for allowing to transmit connections and processes on channels that stands for mobility (we will employ π-calculus mobility as a conceptual means for evolving services orchestration).

The π-Diapason virtual machine supports this first layer. The π-Diapason first layer non-symbolic syntax is a XSB [26] Prolog-based syntax. Given the naming conventions: a process's name begins with a lower case letter while variable's name begins either with an underscore character "_", either with a upper case letter, this first layer syntax definition is as follows:

$0 \equiv$ terminate
P $\equiv$ apply(p)
P.Q $\equiv$ sequence(apply(p), apply(q))
x(y) $\equiv$ receive(X, Y)
$\bar{x}$y $\equiv$ send(X, Y)
$\tau \equiv$ unobservable
P — Q $\equiv$ parallel_split([apply(p), apply(q)])
P + Q $\equiv$ deferedChoice([apply(p), apply(q)])
$[x = y]$ P $\equiv$ if_then(C, apply(p))
or if_then_else(C, apply(p), apply(q))

Process definition and application: a process can be defined and applied in two different ways.
The first one consists in creating an anonymous process that is applied only once (and cannot be reused). Such process is called as *behaviour* and is declared and applied as follows:

```
apply(behaviour(...)).
```

The second way consists in first defining a named process that can be possibly applied several times in a given services orchestration. Such process is called as *process* and its definition and application are as follows:

```
process(process_name(_parameter1, _parameter2, ...),
     behaviour(...))
...
apply(process_name(_value1, _value2, ...))
```

Inter-process communication channels can be defined as connections. A connection is named. We may send a *_value* on a connection *connection_name*.

```
send(connection('connection_name'), _value)
```

Values and variables. Values are literals (integers, floats, booleans, strings). Variables are named (with a first character that is either the underscore character, either a upper case letter). A variable's value can be either a literal, either a connection or a process.

Collections. A collection is either a list, either an array. A list is sorted collection that contains values that may of different types while an array only contains same type values.

```
list([_value1, _value2, ...])
array([_value1,_value2, ...])
```

We introduced an iterate operator for iterating over a collection. Iteration consists in executing a behaviour at each collection's element.

```
iterate(_collection, _iterator, _behaviour)
```

New types definition can be added in the language. We do not detail this feature in this paper.

4.4 The second layer

The π-Diapason 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 [30] are currently described in this layer; the recent twenty new ones introduced in [25] will be expressed soon. This second layer:

- lets us to describe any complex process in an easiest way and at a higher level of abstraction, than only using the first layer (π-calculus definition layer that is less intuitive);
- allows the user to define recurrent structures that will serve as language extensions and will be reused in other process pattern definitions. We have currently express some patterns in order to provide a first library but, as we mentioned, any other structure can be described using this layer;
- contains the formal definition of the services orchestration paterns. Thus, future verifications could be performed;
- is generic enough to be domain independant and can be served as basis for domain-specific languages defined upon it.

Let us take the example of the synchronization pattern, called *synchronize*. This pattern allows to merge different parallelized processes. Expressed using the first layer, its 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 paralleled processes. Once applied, this pattern will use the *iterate* behaviour 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.
Thus, this layer contributes significantly to the services orchestration by using formal orchestration patterns. π-Diapason second layer constitutes a novel and extensible services orchestration formal language and is a serious alternative to well known but less expressive and less extensible orchestration languages (BPEL4WS, WSFL, etc.).

4.5 The third layer: a formal language for expressing evolvable web services orchestrations

This layer is a domain specific layer. In our case it provides the end user language for the expression of web services orchestration. This third abstraction level is defined and is expressed by using the two previous layers (i.e. in terms of π-calculus); thus, a Web Service Oriented Architecture expressed in this third level language is a π-calculus process (without translation). Our approach is clearly different from translation-based approaches, like for example, the translation of a BPEL orchestration to a formal language where it is up to the user to give an interpretation of what a BPEL orchestration is, by defining its own translation mechanism. As consequence a same BPEL orchestration can have several interpretations; thus, properties checking and reasoning on services orchestration are less interesting. Our approach does not deal with translation but promotes a semantical π-calculus expression of each services orchestration concepts. In such a way, an orchestration described in π-Diapason is formally defined and will always have the same interpretation. Moreover, if the π-Diapason orchestration satisfies properties, our approach guarantees that the orchestration will still satisfy them at runtime because the verifications are directly be done on the π-calculus interpretation. The third layer provides a high level language that allows the designer to formalize services orchestration without to be in touch with π-calculus. This layer lets us to describe (a) the behaviour of a services orchestration, (b) the orchestration inputs and outputs, (c) the complex types manipulated and required in such services orchestration, (d) operations of all of the services involved in the orchestration.

In order to define web services orchestrations we have to express web services operations involved in the orchestration. We do not care how services are implemented but we just need operations provided. Thus web service operations are formalized using WSDL files that contain all required information (i.e. operation's name, operation's parent web service, operation's invocation URL, operation's parameters, optionally, operation's return value).

Operation concept is defined in terms of types of the π-Diapason's second layer.

```
type(operation, list([operation_name, service, url,
     requests, response])).
...
type(operation_name, string).
type(service, string).
type(url, string).
...
```

We do not explain deeper such operation complete definition (request and response are not detailed in this paper).

Thanks to the communication protocol (SOAP) employed in WSOA, complex types have to be formalized. Each complex type formalization includes the complex type name, its namespace and its constituents (other types). This formalization is not presented here.

Invoking operation is formalized as a π-calculus process called *invoke*. Such invocation process takes some parameters: the operation name, a collection containing the operation arguments and a return value. In term of π-Diapason first layer concepts, such definition consists in sending a message on a connection named *request* and then, waiting for a message on a connection named *response*. The definition of the invocation process is given as follow:

```
process(invoke(operation(_operation), requests_values
     (_requests), response_value(_response)),
sequence(send(connection('request'), operation_value(
     list([operation(_operation), requests_values(
     _requests)]))),
  receive(connection('response'), response_value(
       _response)))).
```

Web services orchestration is then defined as a π-Diapason second layer process. Such process takes four parameters: the name of the orchestration (remember that a named process can be reused as necessary), the orchestration parameters (i.e. a collection), a return value and a behaviour that orchestrates some invocation processes. The orchestration process behaviour consists in applying the behaviour passed as parameter. Such definition implies new types definition (i.e. parameters, return, etc.) that are not introduced in the paper.

```
type(orchestration_name, string).
process(orchestration(orchestration_name(_name),
     parameters(_parameters), return(_return),
     behaviour(_behaviour)),
   apply(behaviour(_behaviour))).
```

4.6 Services orchestration dynamic evolution

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.

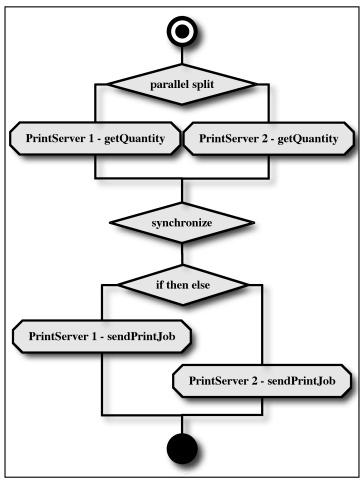

Figure 2. The virtual print shop services orchestration behaviour

```
...
 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 3 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)))))))
```

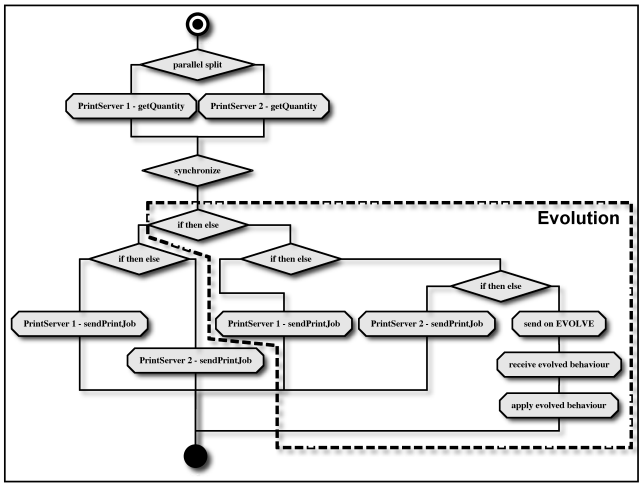

Figure 3. The virtual print shop services orchestration modified behaviour

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 duration before processing the request, etc. (it is up to the services orchestration designer).

5 logic-Diapason: a services orchestration properties definition language

5.1 Analysis principles

When services-orchestration has been defined using π-Diapason, the π-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.

This computation is made by extracting all the possible conditionnal sequences. By conditionnal sequences, we mean all the values that can be assigned to the different conditionnal structures, like an *if_then_else* for example, that might occur in a given orchestration (thanks to sequences and paralellizations). Once this computation is terminated, the π-Diapason virtual machine simulates the differents execution paths and extract a list of states (that what we call a trace) for each paths. Thanks to the virtual print shop example (before evolution), this computation leads to extract two traces. In this example, we only consider a single conditionnal structure with two possible ways (*true* or *false*). When the condition is evaluated as *true*, the first trace output results are the following :

```
parallel_split
PrintServer_1.getQuantity / PrintServer_2.getQuantity
synchronize
PrintServer_1.sendPrintJob
terminate
```

When the condition is evaluated as *false*, the second trace output results are as follows:

```
parallel_split
PrintServer_1.getQuantity / PrintServer_2.getQuantity
synchronize
PrintServer_2.sendPrintJob
terminate
```

5.2 Property definition

When obtained, traces are then analyzed against defined properties expressed using the logic-Diapason language. Generics properties can be proved, like deadlock free, liveness properties and safety properties [2, 27]. In the same way, the logic-Diapason language lets us define and check

well suited properties to prove that a behaviour can or cannot occur during the execution of a specific services orchestration. The logic-Diapason language proposes different operators. The two first one allow to define the scope of a property:

```
forall(_property)    // the property must be true for
     all possible execution traces
exists(_property)    // the property must be true for
     at least one execution trace
```

Some other operators allow to define boolean structures :

```
and(_property_1, _property_2)    // both properties
     must be verified
xor(_property_1, _property_2)    // at least one
     property must be verified
not(_property)                   // the musn't be
     verified
```

Finaly, four operators are provided in order to reason against the scheduling of the states inside a trace:

```
state(_state, _operator, _number_of_occurence)
               // tests the occurency number of a state
unordered_states(_states)
               // tests if a set of states exists
                    whatever the order is
ordered_states(_states)
               // tests if a ordered list of states
                    exists in the same order than
                    listed (other states can be
                    interleaved)
strictly_ordered_states(_states, _operator,
     _number_of_occurence)
               // tests if a ordered list of state
                    exists, and its occurency number,
                    in the strictly same order than
                    listed (no other state can be
                    interleaved)
```

By using these operators, we can now define a property in order to check whether or not the virtual print shop services orchestration is valid against different constraints. For example, we might ensure that a sole print service is invoked at a time; this leads to define a property that consists in invoking one and only one print job operation. Moreover, we can constrain a pre-condition to this print action, i.e. by invoking the print quantity before requesting a print. These constraints are described in the logic-Diapason language as follows:

```
property(
  forall(
    xor(
      and(
        state('PrintServer_1.sendPrintJob', '=', 1),
        and(
          state('PrintServer_2.sendPrintJob', '=', 0),
          ordered_states(['PrintServer_1.getQuantity',
               'PrintServer_1.sendPrintJob']))),
      and(
        state('PrintServer_2.sendPrintJob', '=', 1),
        and(
          state('PrintServer_1.sendPrintJob', '=', 0),
          ordered_states(['PrintServer_2.getQuantity',
               'PrintServer_2.sendPrintJob'])))))).
```

For all possible executions (see the previous property definition), either we have one and only one occurence of the *PrintServer_1* service's *sendPrintJob* operation, and no occurence of the *PrintServer_2* service's *sendPrintJob* operation, either we have the opposite (one and only one occurence of the *PrintServer_2* service's *sendPrintJob* operation and no occurence of the *PrintServer_1* service's *sendPrintJob* operation). In both cases, the property definition implies that a *getQuantity* operation has to be invoked before invoking a *sendPrintJob* operation on the same service.

6 Services orchestration deployment and execution

According to verifications the user made, 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 (i.e. orchestration compositions). Finally, the new Web service deployed is executed thanks to our π-Diapason virtual machine. This Web service embeddeds the π-Diapason orchestration description and the π-Diapason virtual machine. The π-Diapason virtual machine (π-Diapason interpreter) has been implemented using XSB [26]. 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.

7 Conclusion

We have introduced our approach for formally define, check, deploy, execute and evolve Web services orchestrations; π-Diapason and logic-Diapason are complementary languages for that purpose. Diapason is our engineering envdironment that supports and enacts the Diapason framework and accompanied tools (graphical modeller for services orchestration modelling, analyser for checking traces against properties, graphical animator for simulating services orchestration execution). In addition, π-Diapason supports on the fly services orchestration evolution 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. A more sophisticated scenario has been presented in [23] that has also be enacted using Diapason.

References

[1] *The SPIN Model Checker: Primer and Reference Manual.* Number ISBN 0-321-22862-6. Addison-Wesley, 2004.

[2] I. Alloui. Property verification and change impact analysis for model evolution. In *1ères journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, pages 169–174, 2005.

[3] 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.

[4] A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards a formal development negotiation among web services using lotos/cadp. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)*, 2005.

[5] S. Cimpan and H. Verjus. Challenges in architecture centred software evolution. In *CHASE: Challenges in Software Evolution*, pages 1–4, Bern, Switzerland, April 2005.

[6] S. Cîmpan, H. Verjus, and I. Alloui. Dynamic architecture based evolution of enterprise information systems. In *International Conference on Enterprise Information Systems (ICEIS'07)*, pages 221–229, Madeira, Portugal, June 2007.

[7] A. Ferrara. Web services: A process algebra approach. Technical report, June 2004.

[8] B. Fitzerald and C. Olsson, editors. *The Software and Services Challenge*. EY 7th Framework Programme, Contribution to the preparation of the Technology Pillar on "Software, Grids, Security and Dependability", 2006.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ltsa-ws: A tool for model-based verification of web service compositions and choreography. In *IEEE International Conference on Software Engineering (ICSE 2006)*, 2006.

[10] 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.

[11] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[12] M. Hepner, R. Gamble, M. Kelkar, L. Davis, and D. Flagg. Patterns of conflict among software components. *Journal of Systems and Software*, 79(4):537–551, 2006.

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

[14] 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.

[15] F. Leymann. Web services flow language (wsfl 1.0), 2001.

[16] 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.

[17] T. Mens, R. Wuyts, K. D. Volder, and K. Mens. Workshop proceedings — declarative meta programming to support software development. *ACM SIGSOFT Software Engineering Notes*, 28(2), Jan. 2003.

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

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

[20] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In I. CS, editor, *WISE'03*, pages 3–12, 2003.

[21] 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].

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

[23] 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.

[24] A. P. Ravn, O. Owe, P. Giambiagi, and G. Schneider. Language-based support for service oriented architectures: Future directions. In *Proceedings of 1st International Conference on Software and Data Technologies (ICSOFT 2006)*, page 6, Setúbal, Portugal, 2006.

[25] 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.

[26] 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.

[27] 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.

[28] M. Solanki, A. Cau, and H. Zedan. Asdl: A wide spectrum language for designing web services. In *15th International World Wide Web Conference (WWW2006)*, 2006.

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

[30] 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.