DIAPASON: A FORMAL APPROACH FOR SUPPORTING AGILE AND EVOLVABLE INFORMATION SYSTEM SERVICE-BASED ARCHITECTURES

Hervé Verjus and Frédéric Pourraz
University of Savoie - Polytech'Savoie
LISTIC-LS - Language and Software Evolution Group
BP 80439, 74344 Annecy-le-Vieux Cedex, France

Keywords: Information system architecture, SOA, Services orchestration, BPM, Dynamic evolution, π-calculus.

Abstract: This paper presents a novel approach called Diapason, for expressing, verifying and deploying evolvable information system service-based architectures. Diapason promotes a π-calculus-based layered language and a corresponding virtual machine for expressing and executing dynamic and evolvable services orchestration that support agile business processes. Information system service-based architecture may dynamically evolve, whatever the changes may occur during the services orchestration lifecycle.

1 INTRODUCTION

Service-Oriented Architectures (SOA) is a recent paradigm for building large scale information system service-oriented architectures (IS-SOA) from distributed services. One of the main interest of SOA is basically the underlying ability of such architecture to inherently being evolvable; because the underlying idea of SOA is that the services are loosely coupled and the SOA could be adapted to its environment. Services are supposed to be autonomous, self-contained, and we do not control nor we have authority over them. When considering information system perspective, the openess, flexibility, agility of such information system service-oriented architecture is very challenging for addressing new functionalities, time to market, etc. SOAs introduce new engineering issues (Fitzerald and Olsson, 2006; Papazoglou et al., 2006) and SOA evolution is becoming very challenging (Papazoglou et al., 2006). Thus, important engineering questions are addressed to service-based information system architects: what about the quality of IS-SOA ? How can we ensure that IS-SOA fit expectations ? How are IS-SOAs able to be dynamically adapted ? How can we ensure that the executed IS-SOA is consistent with the design ?
We present in this paper our formal language called π-Diapason that lets the user/architect to formally express and control dynamic evolvable web services orchestrations. From service-oriented information system engineering perspective, service orchestration is a key issue as it supports (at least partially) business processes.
The section 2 of this paper will present works and challenges related to SOA engineering. Section 3 will present our formal language, called π-Diapason, for designing evolvable IS-SOAs and section 4 will then conclude.

2 RELATED WORK AND CHALLENGES

Services orchestration addresses business process through services invocations scheduling and organisation. Services orchestration aims at defining executable processes by providing orchestration languages (among the most well known BPEL, XLANG, WSFL, etc. (Peltz, 2003)) that are executable languages (by the way of workflow engines). BPEL allows to define abstract business processes and executable processes. But such languages lack in services orchestration reasoning, reuse, dynamic evolution (Papazoglou et al., 2006; Ravn et al., 2006): i.e. business processes expressed using these languages cannot be formally checked, nor they can evolve dynamically.

Verjus H. and Pourraz F. (2008).
DIAPASON: A FORMAL APPROACH FOR SUPPORTING AGILE AND EVOLVABLE INFORMATION SYSTEM SERVICE-BASED ARCHITECTURES.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 76-81
DOI: 10.5220/0001677000760081
Copyright © SciTePress

During the part three years, some works (Salaün et al., 2004; Fu et al., 2004; Foster et al., 2006; Solanki et al., 2006; Chirichiello and Salaün, 2005) have been devoted to employ formal approaches for services orchestration definition and analysis. These works mainly consider BPEL as the core language for expressing service orchestration. Thus, authors propose mappings approaches between formal languages and BPEL. As these languages are not expressive equivalent, mappings/transformation produce gap and discrepencies and analysis cannot be exhaustive enough. Our approch we will introduce hereafter, deals simultaneously with two important considerations that other approches do not: *(i) we propose a new formal and executable language (named π-Diapason) for orchestrating services avoiding mappings and transformations from or to BPEL; (ii) π-Diapason is based on the π-calculus that is the most expressive process algebra (complete in sense of the Turing machine).* As consequence, π-Diapason is more powerful than BPEL, π-Diapason is executable, π-Diapason supports evolvable services orchestration formalization that can be analyzed and executed.

3 EVOLVABLE IS-SOAS FORMALIZATION

3.1 π-Diapason Formal Foundations

Diapason (Pourraz and Verjus, 2007) is a π-calculus based approach allowing formal services based systems modeling, deployment and execution. The aim of using a process algebra (which formally models interactions between processes (Salaün et al., 2004)) 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 (Hoare, 1985), CCS (Milner, 1989), π-calculus (Milner, 1999), 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 IS-SOA architect to design and specify IS-SOAs (focusing on services orchestration);
- provides Domain Specific Layer (see below) in order to simplify IS-SOA design;
- is formally defined, based on π-calculus;
- supports dynamic IS-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 the design (abstract level) and the corresponding implementation (concrete level) as it is the same language that covers both levels. There is no mapping rules, and no need for consistency management. The IS-SOA specified will be the one that will be interpreted. IS-SOA's execution is precisely carried out by the π-Diapason virtual machine; this latter can be used for IS-SOA simulation and validation purpose and is the runtime engine that interpretes services orchestration expressed in π-Diapason.

3.2 π-calculus Basis

Thanks to the π-calculus (Milner, 1989), some operators are required (we defined naming conventions: upper case letters stand for processes 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. The π-Diapason formal definition and implementation are given in (Pourraz and Verjus, 2007).

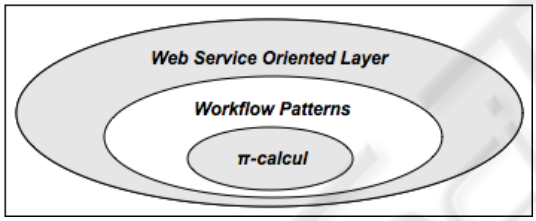

Figure 1: π-Diapason as a layered language.

3.3 The First Layer

The π-Diapason first layer is the expression of the high order, typed, asynchronous and polyadic π-calculus (Milner, 1999):

- polyadic for simultaneously sending 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 pass connections and processes on channels that stands for mobility (we will employ π-calculus mobility as conceptual means for evolvable services orchestration).

The π-Diapason virtual machine supports this first layer. The π-Diapason first layer non-symbolic syntax is a XSB (Sagonas et al., 2006) 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 ≡ terminate
P ≡ apply(p)
P.Q ≡ sequence(apply(p), apply(q))
x(y) ≡ receive(X, Y)
$\bar{x}$y ≡ send(X, Y)
τ ≡ unobservable
P — Q ≡ parallel_split([apply(p), apply(q)])
P + Q ≡ deferedChoice([apply(p), apply(q)])
$[x = y]$ P ≡ 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.

3.4 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 (van der Aalst et al., 2003) are currently described in this layer and we will define some others 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 verification tasks 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 (BPEL, WSFL, etc.).

3.5 The Third Layer

This layer is a domain specific layer. In our case it provides to the end user, language for formalizing web services orchestration (it consists in a Domain Specific Language). This third abstraction level is defined by using the two previous layers; thus, an information system Web service oriented architecture 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's inputs and outputs,
- the complex types manipulated and required in such services orchestration,
- 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. We obtain operations' specifications by analyzing WSDL files that contain all required information, among them:

- the operation's name;
- the operation's parent web service;
- the operation's invocation URL;
- the operation's parameters;
- the operation's return value (optional).

Operation concept is defined in terms of types of the π-Diapason 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 Web Service Oriented Architectures, 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))).
```

3.6 IS-SOA Dynamic Evolution

Thanks to the π-calculus mobility, we may modify the services orchestration dynamically, at runtime, without to stop this runtimed orchestration. Inherently, 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 (Pourraz and Verjus, 2007).

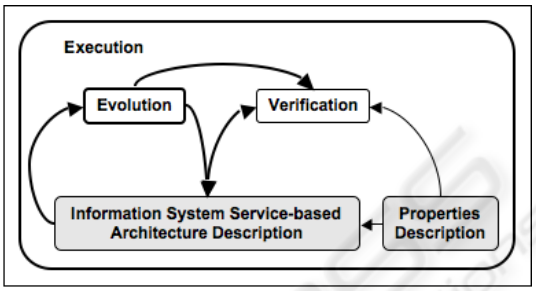

Figure 2: The services orchestration evolution process.

We are now explaining how a services orchestration formally defined using π-Diapason third layer is able to evolve dynamically (at runtime). The orchestration behaviour (see the orchestration in the left part of the Figure 2) consists in scheduling the Web services operations invocations by the way of process patterns (sequence, parallel, conditional expressions) defined in the second layer of the π-Diapason language. In order to perform the external evolution, an "evolution point" has been defined. A connection called "EVOLVE", defined in the services orchestration π-Diapason code, is a connection for firing and receiving services orchestration changes. This connection allows us to dynamically pass a behaviour to the orchestration (i.e. conform to process mobility (Milner, 1999)). Once received (the *_evolved_behaviour* variable is becoming not null), this changed or new behaviour (see the orchestration in the right part of the Figure 2) can be applied within the orchestration (the receiver). 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 as it was planned, without modification. The following piece of code is a services orchestration example that may receive a new behaviour through the "EVOLVE" connection and then apply dynamically this new behaviour.

```
...
 behaviour(
  parallel_split([
    // An external evolution may be requested
    receive(connection('EVOLVE'), values([
         _evolved_behaviour]))
    // some invocations of second layer patterns
    ...
    sequence(if_then_else(_evolved_behaviour != NULL,
                 // Evolution Required
```

```
                apply(_evolved_behaviour)
                // NO Evolution Required
                if_then_else( // a test,
                   // the orchestration goes on without
                        modification
                   ...
      terminate))])])),
...
```

The π-Diapason expression of the behaviour that is containing modifications (for example new orchestration process, adding, removing services and/or operations) is dynamically received and applied (see the bottom part of the figure 2). It is up to the user to express and on the fly provide to the π-Diapason virtual machine, the definition of such behaviour. This evolution mechanism at π-Diapason code level deals with unpredicatble situations that may occur at runtime, without having to suspend or to stop the execution.

4 CONCLUSIONS

π-Diapason is a formal and layered language for expressing processes at the second layer level by the way of patterns (strongly related to business processes/workflows); the third layer offers a high level language that allows the user to formalize evolvable information system service-based architectures (i.e. service orchestration) without being in touch with π-calculus. Such approach aims at providing means for supporting agile business processes and flexible and open information systems service-based architectures through service orchestration: services orchestration may be internally modified by receiving a new behaviour and the current execution may be deeply and consistently modified. Our approach is non-intrusive as the service orchestration is encapsulated in the deployed web service that is the services choreographer without to modify any other part of the information system architecture.

ACKNOWLEDGEMENTS

This work is partially funded by the french ANR JC05 42872 COOK Project.

REFERENCES

Chirichiello, A. and Salaün, G. (2005). 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)*.

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

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

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

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Milner, R. (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press.

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

Peltz, C. (2003). Web services orchestration: A review of emerging technologies, tools, and standards. Technical report, HP.

Pourraz, F. and Verjus, H. (2007). π-diapason: A π-calculus based formal language for expressing evolvable web services orchestrations. Research Report LISTIC 07/06, University of Savoie - LISTIC.

Ravn, A. P., Owe, O., Giambiagi, P., and Schneider, G. (2006). 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.

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

Salaün, G., Ferrara, A., and Chirichiello, A. (2004). Negotiation among web services using lotos/cadp. In Springer, editor, *Proceedings of the European Conference On Web Services (ECOWS'04)*, volume 3250, pages 198–212, Erfurt, Germany.

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

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