# $\pi$-Diapason: A $\pi$-calculus Based Formal Language For Expressing Evolvable Web Services Orchestrations

Frédéric Pourraz and Hervé Verjus

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

**Abstract.** This paper presents $\pi$-Diapason, a $\pi$-calculus-based formal and layered language for expressing complex Web Services orchestrations that are able to dynamically evolve. Once formalized, one can reason on the Web Services orchestration. Then, it is deployed, executed and can be dynamically changed at runtime. Services orchestration verification, execution and evolution are supported by some tools ($\pi$-Diapason virtual machine, $\pi$-Diapason services orchestration execution traces checker).

## 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 [1–3] 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 [1, 4–6] and SOA evolution is becoming very challenging [1, 6]. 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 [7–12]. 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 ?
We present in this paper our formal language called $\pi$-Diapason that lets the user to formally express web services orchestrations that are able to be dynamically adapted in a formal controled manner.

The section 2 of this paper will present works and challenges related to SOA engineering. Section 3 will present our formal language, called $\pi$-Diapason, for

designing evolvable Web-service-based architectures. Section 4 will address services orchestration checking and deployment using our approach and section 5 will then conclude.

## 2 Related work and challenges

In [2] 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 [13]. 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 [14], XLANG [15], WSFL [16], BPML, etc. [13]) 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 [6, 17]: 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. [3] 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. As consequence, 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. We will now present the $\pi$-Diapason language we offer for expressing services orchestration that are inherently able to dynamically evolve.

## 3 $\pi$-Diapason: a formal language for expressing evolvable web services orchestrations

### 3.1 $\pi$-Diapason formal foundations

Diapason [18] is a $\pi$-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 [3]) 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 [19], CCS [20], $\pi$-calculus [21], etc. In our case, we have adopted the $\pi$-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 $\pi$-calculus terms of behaviours and channels. A channel aims at connecting two behaviours and lets them interacting together. The first order $\pi$-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, $\pi$-Diapason is based on

the high order $\pi$-calculus which is more powerful. In addition to the first kind of mobility, hight order $\pi$-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.

$\pi$-Diapason aims at proposing well defined formal abstractions for expressing services orchestration that can be then executed (because $\pi$-Diapason is an executable formal language); $\pi$-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 $\pi$-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 $\pi$-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 $\pi$-Diapason (see section 4).

### 3.2 $\pi$-calculus basis for the $\pi$-Diapason language

Thanks to the $\pi$-calculus [20], some operators are required (we defined naming conventions: upper case letters stand for process while lower case letters stand for variables):

- $\mu$.P: the prefix of a process by an action where $\mu$ 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,
  - $\tau$: 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:

$$P \stackrel{def}{=} 0 \mid x(y).P \mid \bar{x}y.P \mid \tau.P \mid P1|P2 \mid P1 + P2 \mid [x = y]P \mid A\ (x1,\ ...,\ xn)$$

$\pi$-Diapason has been designed as a layered language which provides three abstraction levels. The $\pi$-Diapason layers syntax and semantics (transition rules and type definition rules are given in the appendix).

### 3.3 The first layer

The $\pi$-Diapason first layer is the expression of the high order, typed, asynchronous and polyadic $\pi$-calculus [21]:

- 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 $\pi$-calculus mobility as a conceptual means for evolving services orchestration).

The $\pi$-Diapason virtual machine supports this first layer. The $\pi$-Diapason first layer non-symbolic syntax is a XSB [22] 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:

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

See the appendix for the $\pi$-Diapason layers' definition and implementation.

### 3.4 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, ...))
```

### 3.5 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)
```

### 3.6 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.

### 3.7 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)
```

### 3.8 New types definition

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

### 3.9 The second layer

The $\pi$-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 [23] are currently described in this layer; the recent twenty new ones introduced in [24] 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 ($\pi$-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 $\pi$-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. $\pi$-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.).

### 3.10 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 expressed by using the two previous layer; thus, a Web Service Oriented Architecture expressed in this third level language is directly expressed as a $\pi$-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.

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, 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 $\pi$-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 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.

### 3.11 Invoking operation

is formalized as a $\pi$-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 $\pi$-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)))).
```

### 3.12 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))).
```

### 3.13 Services orchestration dynamic evolution: orchestration changes expression 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 [18].

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 and the services orchestration can be considered at some extends, as a self-adaptable services orchestration.

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 1) 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 [21]). Once received (the *_evolved_behaviour* variable is becoming not null), this changed

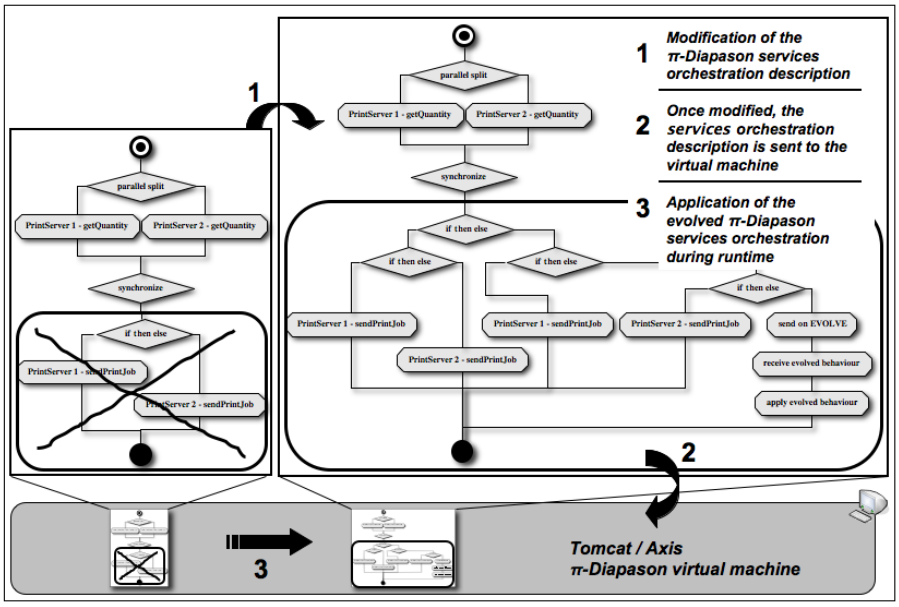


**Fig. 1.** The services orchestration evolution process

or new behaviour (see the orchestration in the right part of the Figure 1) can be applied within the orchestration (the receiver). Such behaviour application modifies dynamically the orchestration according to the behaviour's $\pi$-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 containing an "EVOLVE" connection and for applying this new (modified) 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 $\pi$-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 1). It is up to the user to express and on the fly provide to the $\pi$-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 Services orchestration checking, deployment and execution

When services-orchestration has been formalized using π-Diapason, the π-Diapason virtual machine is used in order to achieve two goals. The first one is the services orchestration simulation before its execution (the validation) and the second one is the execution itself. Simulation provides 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 Diapason* language (this properties definition language is not presented in this paper). Generic properties can be proved, like deadlock free, liveness properties and safety properties [3, 25, 18]. According to these verifications, it is up to the architect to validate and to decide whether or not the π-Diapason formalized orchestration can be deployed or not. 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. orchestrations compositions). Finally, the new web service deployed is executed thanks to the π-Diapason code. The deployed web service integrates the π-Diapason virtual machine that has been implemented using XSB Prolog [22]. When services orchestration has to dynamically evolve, the virtual machine computes again execution traces taking into account changes; traces are then analyzed against properties definition and the changes may (or not) be applied on the fly, at runtime, without to stop the current services orchestration execution.

## 5 Conclusion

π-Diapason is a formal and layered language for expressing processes at the second layer level by the way of patterns; the third layer offers a high level language that allows the user to formalize evolvable web services orchestration without being in touch with π-calculus. A services orchestration expressed using π-Diapason can be then validated, executed, deployed and evolved dynamically. Diapason is an environment that supports π-Diapason and provides some tools (virtual machine, checker, graphical animator). π-Diapason supports business services orchestration evolution that is required and motivated in plethora of papers [18, 17, 3, 25]: all or part of an orchestration definition (called a services orchestration fragment) can be dynamically 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 and consistently modified.

## References


1. Papazoglou, M.P.: Service-oriented computing: Concepts, characteristics and directions. In CS, I., ed.: WISE'03. (2003) 3–12
2. Kontogiannis, K., Lewis, G.A., Smith, D.B.: The landscape of service-oriented systems: A research perspective. In: Proceedings of International Workshop on Systems Development in SOA Environments. (2006)
3. Salaün, G., Bordeaux, L., Bordeaux, M.S.L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: ICWS, IEEE Computer Society (2004) 43–50
4. Hepner, M., Gamble, R., Kelkar, M., Davis, L., Flagg, D.: Patterns of conflict among software components. Journal of Systems and Software **79**(4) (2006) 537–551
5. Fitzerald, B., Olsson, C., eds.: The Software and Services Challenge, EY 7th Framework Programme, Contribution to the preparation of the Technology Pillar on "Software, Grids, Security and Dependability" (2006)
6. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-oriented computing: A research roadmap. In Cubera, F., Krämer, B.J., Papazoglou, M.P., eds.: 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].
7. Garlan, D., Shaw, M.: An introduction to software architecture. In Ambriola, V., Tortora, G., eds.: Advances in Software Engineering and Knowledge Engineering, Singapore, World Scientific Publishing Company (1993) 1–39
8. Cîmpan, S., Verjus, H., Alloui, I.: Dynamic architecture based evolution of enterprise information systems. In: International Conference on Enterprise Information Systems (ICEIS'07), Madeira, Portugal (2007) 221–229
9. Mens, T., Wuyts, R., Volder, K.D., Mens, K.: Workshop proceedings — declarative meta programming to support software development. ACM SIGSOFT Software Engineering Notes **28**(2) (2003)
10. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2005), IEEE Computer Society (2005) 123–131
11. Cimpan, S., Verjus, H.: Challenges in architecture centred software evolution. In: CHASE: Challenges in Software Evolution, Bern, Switzerland (2005) 1–4
12. Verjus, H., Cimpan, S., Alloui, I., Oquendo, F.: Gestion des architectures évolutives dans archware. In: 1ère Conférence francophone sur les Architectures Logicielles (CAL 2006), Nantes (2006) 41–57
13. Peltz, C.: (Web services orchestration: A review of emerging technologies, tools, and standards)
14. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1. Specifications, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems (2003)
15. Thatte, S.: (Xlang - web services for business process design)
16. Leymann, F.: Web services flow language (wsfl 1.0) (2001)

17. Ravn, A.P., Owe, O., Giambiagi, P., Schneider, G.: Language-based support for service oriented architectures: Future directions. In: Proceedings of 1st International Conference on Software and Data Technologies (ICSOFT 2006), Setúbal, Portugal (2006) 6
18. Pourraz, F., Verjus, H.: Diapason: an engineering environment for designing, enacting and evolving service-oriented architectures. In: International Conference on Software Engineering Advances (ICSEA 2007), France, IEEE Computer Society (2007) 23–30
19. Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
20. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
21. Milner, R.: Communicating and Mobile Systems: The $\pi$-calculus. Cambridge University Press (1999)
22. Sagonas, K., Swift, T., Warren, D.S., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R.F., Dawson, S., Kifer, M.: The xsb system version 3.0 volume 1: Programmer's manual. Technical report, XSB consortium (2006)
23. van der Aalst, W.H.M., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases, 14(3) (2003)
24. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPM Center Report BPM-06-22 , BPMcenter.org (2006)
25. Pourraz, F., Verjus, H., Oquendo, F.: An architecture-centric approach for managing the evolution of eai services-oriented architecture. In: Eighth International Conference on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus (2006) 234–241

# A Appendix A. BNF (*Backus-Naur Form*) of the π-Diapason first layer

description ::= < declaration >+
declaration ::= type_declaration |
behaviour_declaration

type_declaration ::= **type(** type_name **,** type **).**

type ::= **ANY** |
**string** |
**integer** |
**boolean** |
array |
list
array ::= **array**(collection_name)
list ::= **list**([collection_name < **,** collection_name >* ])
collection_type ::= type_name |
**connection** |
**behaviour**

behaviour_declaration ::= **process(** behaviour_name < parameters >? **,** transition**).**

parameters ::= ( parameter < **,** parameter >* )
parameter ::= value |
connection |
behaviour

value ::= type_name(variable)
connection ::= **connection**(variable)
behaviour ::= **behaviour**(variable)

action ::= **value**(value_name**,** parameter_literal) |
**instanciate**(process) |
**sequence**(action**,** action) |
**parallel_split**([ action**,** action < **,** action >* ]) |
**if_then**(condition**,** action) |
**if_then_else**(condition**,** action**,** action) |
**defered_choice**([ action**,** action < **,** action >* ]) |
**send**(connection_literal) |
**send**(connection_literal**,** parameter_literal) |
**receive**(connection_literal) |
**receive**(connection_literal**,** parameter_literal) |
**iterate**(collection_literal**,** action) |
**iterate**(collection_literal**,** parameter**,** action) |
**unobservable** |
**terminate**

process ::= behaviour_literal |
behaviour_name < parameters_literal >?

parameters_literal ::= **(** parameter_literal < **,** parameter_literal >* **)**
parameter_literal ::= value_literal |
connection_literal |
behaviour_literal

value_literal ::= value |
type_name**(value(**value_name**))** |
type_name**(**string**)** |
type_name**(**integer**)** |
type_name**(**boolean**)** |
collection_literal
collection_literal ::= value |
type_name**(value(**value_name**))** |
type_name**(array(**[ < value_literal < **,** value_literal >* >? ]**))** |
type_name**(list(**[ < value_literal < **,** value_literal >* >? ]**))**
connection_literal ::= connection |
**connection(**string**)**
behaviour_literal ::= behaviour |
**behaviour(**action**)**
condition ::= variable |
variable operator variable |
variable operator value_literal |
value_literal operator variable |
value_literal operator value_literal |
**(** or | and **)**
or ::= evaluation **;** evaluation
and ::= evaluation **,** evaluation
evaluation ::= condition |
or |
and
operator ::= == |
\== |
< |
=< |
> |
>=

value_name ::= string
type_name ::= name
behaviour_name ::= name
name ::= <**a-z**> < <**a-z**> | <**A-Z**> | <**0-9**> | _ >*

variable ::= _ < <**a-z**> | <**A-Z**> | <**0-9**> | _ >*
string ::= **'** < <**a-z**> | <**A-Z**> | <**0-9**> >* **'**
integer ::= < <**0-9**> >+
boolean ::= **true** | **false**

**Legend**

<...>? 0 ou 1
<...>* 0 or more
<...>+ 1 or more
<**a-z**> a | b | c | ... | z
<**A-Z**> A | B | C | ... | Z
<**0-9**> 0 | 1 | 2 | ... | 9

# B Appendix B. First layer formalization and implementation

## B.1 Inactive process

– **Syntax and semantic :**

$$\mathbf{0 \equiv terminate}$$

*Type definition rule :* $\frac{}{terminate{:}BEHAVIOUR}$

– **Implementation :**

**action**(terminate).

## B.2 Process instantiation

– **Syntax and semantic :**

$$\mathbf{P \equiv instanciate(p)}$$

*Type definition rule :* $\frac{p{:}BEHAVIOUR}{instanciate(p){:}BEHAVIOUR}$

– **Implementation :**

**action**(instanciate(_action_1)) :-
(_action_1 = behaviour(_action_2) ->
**action**(_action_2);
**process**(_action_1, _action_2), **action**(_action_2)).

## B.3 An action preceeds a process (sequence)

– **Syntax and semantic :**

$$\mathbf{P.Q \equiv sequence(instanciate(p), instanciate(q))}$$

*Transition rule :* $\frac{}{sequence(instanciate(p),instanciate(q)) \overset{instanciate(p)}{\longrightarrow} instanciate(q)}$

*Type definition rule :* $\frac{p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{sequence(instanciate(p),instanciate(q)){:}BEHAVIOUR}$

– **Implementation :**

**action**(sequence(_action_1, _action_2)) :-
**action**(_action_1),
**action**(_action_2).

### B.4 Positive prefix

– **Syntax and semantic :**

$$\mathbf{x(y) \equiv receive(X,\ Y)}$$

*Transition rule :* $\dfrac{}{sequence(receive(X,Y),instanciate(p)) \overset{receive(X,Y)}{\longrightarrow} instanciate(p)}$

*Type definition rule :* $\dfrac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(receive(X,Y),instanciate(p)){:}BEHAVIOUR}$

– **Implementation :**

```
action(receive(_connection)) :-
    (clause(receive(_connection), true) ->
        retract(receive(_connection));
        action(receive(_connection))).

action(receive(_connection, _value_1)) :-
    (clause(receive(_connection, _value_2), true) ->
        retract(receive(_connection, _value_2)), _value_1 = _value_2;
        action(receive(_connection, _value_1))).
```

### B.5 Negative prefix

– **Syntax and semantic :**

$$\mathbf{\bar{x}y \equiv send(X,\ Y)}$$

*Transition rule :* $\dfrac{}{sequence(send(X,Y),instanciate(p)) \overset{send(X,Y)}{\longrightarrow} instanciate(p)}$

*Type definition rule :* $\dfrac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(send(X,Y),instanciate(p)){:}BEHAVIOUR}$

– **Implementation :**

```
action(send(_connection)) :-
    (_connection == connection('request') ->
        request(_value);
        assert(receive(_connection))).

action(send(_connection, _value)) :-
    (_connection == connection('request') ->
        request(_value);
        assert(receive(_connection, _value))).
```

### B.6 Silent prefix

– **Syntax and semantic :**

$$\tau \equiv \textbf{unobservable}$$

*Transition rule :* $\dfrac{}{sequence(unobservable,instanciate(p)) \overset{unobservable}{\longrightarrow} instanciate(p)}$

*Type definition rule :* $\dfrac{p{:}BEHAVIOUR}{sequence(unobservable,instanciate(p)){:}BEHAVIOUR}$

– **Implementation :**

**action**(unobservable).

### B.7 Paralell actions

– **Syntax and semantic :**

$$\textbf{P} \text{ — } \textbf{Q} \equiv \textbf{parallel\_split([instanciate(p), instanciate(q)])}$$

*Transition rule :*

$$\frac{p \xrightarrow{\alpha} p'}{parallel\_split([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} parallel\_split([instanciate(p'),instanciate(q)])}$$

$$\frac{q \xrightarrow{\alpha} q'}{parallel\_split([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} parallel\_split([instanciate(p),instanciate(q')])}$$

*Type definition rule :* $\dfrac{p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]){:}BEHAVIOUR}$

– **Implementation :**

```
action(parallel_split(_actions)) :-
    action(parallel_split(_actions, [])).

action(parallel_split([_action_1 | _action_2], _threads)) :-
    thread_create(action(_action_1), _id),
    (_action_2 == [] ->
        thread_join([_id | _threads], _);
        action(parallel_split(_action_2, [_id | _threads]))).
```

### B.8 Non-determinist summation

– **Syntax and semantic :**

$$\textbf{P} + \textbf{Q} \equiv \textbf{defered\_choice([instanciate(p), instanciate(q)])}$$

*Transition rule :*

$$\frac{p \xrightarrow{\alpha} p'}{defered\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(p')}$$

$$\frac{q \xrightarrow{\alpha} q'}{defered\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(q')}$$

*Type definition rule :* $\dfrac{p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]){:}BEHAVIOUR}$

– **Implementation :**

```
action(defered_choice(_actions)) :-
    length(_actions, _length),
    _max is _length+1,
    random(1, _max, _choice),
    ith(_choice, _actions, _action),
    action(_action).
```

### B.9 Matching

– **Syntax and semantic :**

$$[x = y]\ \mathbf{P} \equiv \mathbf{if\_then(C,\ instanciate(p))} \\ \mathbf{ou\ if\_then\_else(C,\ instanciate(p),\ instanciate(q))}$$

***Transition rules :***

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} instanciate(p)} \text{ if C is true}$$

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} terminate} \text{ if C is false}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(p)} \text{ if C is true}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(q)} \text{ if C is false}$$

***Type definition rule :***

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR}{if\_then(C,instanciate(p)){:}BEHAVIOUR}$$

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{if\_then(C,instanciate(p),instanciate(q)){:}BEHAVIOUR}$$

– **Implementation :**

```
action(if_then(_condition, _action)) :-
    (call(_condition) ->
        action(_action);
        true).

action(if_then_else(_condition, _action_1, _action_2)) :-
    (call(_condition) ->
        action(_action_1);
        action(_action_2)).
```

**Legend**

- *CONNECTION*
  Type corresponding to an inter-process communication channel,
- *BEHAVIOUR*
  Type corresponding to a process,
- *TYPE*
  Generique type corresponding to any base type : *CONNECTION, BEHAVIOUR, BOOLEAN, FLOAT, INTEGER, STRING*.

# C Appendix C. Workflow patterns formalization

## C.1 Sequence

This pattern corresponds to the *sequence* operator of the first layer.

## C.2 Parallel Split

This pattern corresponds to the *parallel_split* operator of the first layer.

## C.3 Synchronization

```
process(
    synchronize(connections(_connections)),
    behaviour(
        iterate(connections(_connections), connection(_connection),
            receive(connection(_connection))))).
```

## C.4 Exclusive Choice

This pattern corresponds to the *if_then_else* operator of the first layer.

## C.5 Simple Merge

This pattern corresponds to the sequence (*sequence* operator) of the *if_then_else* operator and any other behaviour.

## C.6 Multi Choice

```
type(tests, array(test)).
type(test, list([condition(_condition), behaviour(_behaviour)])).

process(
    choice(test(list([condition(_condition), behaviour(_behaviour)]),
        connection(_connection))),
    behaviour(
        parallel_split([
        if_then_else(condition(_condition),
                sequence(send(connection(_connection), true),
                    instanciate(behaviour(_behaviour))),
                send(connection(_connection), false))]))).

process(
    multi_choice(tests(_tests), connection(_connection)),
    behaviour(
        iterate(tests(_tests), test(_test),
            instanciate(choice(test(_test), connection(_connection)))))).
```

### C.7 Structured Synchronizing Merge

**type**(**synchronizings**, **array**(synchronizing)).
**type**(**synchronizing**, **list**([connection(_connection_1), connection(_connection_2)])).

**process**(
  **sync**(synchronizing(**list**([connection(_connection_1), connection(_connection_2)]))),
  **behaviour**(
    **sequence**(**receive**(connection(_connection_1), _condition),
    **if_then**(_condition, **receive**(connection(_connection_2)))))).

**process**(
  **synchronizing_merge**(synchronizings(_synchronizings)),
  **behaviour**(
    **iterate**(synchronizings(_synchronizings), synchronizing(_synchronizing),
      **instanciate**(sync(synchronizing(_synchronizing)))))).

### C.8 Multi Merge

**process**(
  **multi_merge**(connection(_connection), behaviour(_behaviour)),
  **behaviour**(
    **sequence**(**receive**(connection(_connection)),
    **parallel_split**([

      **instanciate**(**multi_merge**(connection(_connection), behaviour(_behaviour))),

      **instanciate**(behaviour(_behaviour))])))).

### C.9 Discriminator

This pattern corresponds to the use of the *synchronize* pattern with only one connection as parameter.

### C.10 Arbitrary Cycles

**process**(
  **cycle**(connection(_connection), behaviour(_behaviour)),
  **behaviour**(
    **parallel_split**([

      **sequence**(**receive**(connection(_connection)),
      **instanciate**(**cycle**(connection(_connection), behaviour(_behaviour)))),

      **instanciate**(behaviour(_behaviour))]))).

### C.11 Implicit Termination

This pattern corresponds to the omission of the *terminate* operator.

### C.12 Multi Instances without Synchronization

```
process(
    multi_instances(connection(_connection)),
    behaviour(
        sequence(receive(connection(_connection), process(_process)),
        parallel_split([

            instanciate(multi_instances(connection(_connection))),

            instanciate(_process)])))).
```

### C.13 Multi Instances with a Priori Design Time Knowledge or with a Priori Runtime Knowledge

```
type(processes, array(process)).

process(
    multi_instances_with_sync(processes(_processes)),
    behaviour(
        parallel_split([

            instanciate(multi_instances(connection('setProcess'))),

            iterate(processes(_processes), process(_process),
              send(connection('setProcess'),
              behaviour(sequence(instanciate(_process), send(connection('getSync')))))),

            iterate(processes(_processes), receive(connection('getSync')))])))).
```

### C.14 Multi Instances without a Priori Runtime Knowledge

```
process(
    multi_instances_with_sync(processes(_processes_1), connection(_connection)),
    behaviour(
        parallel_split([

            instanciate(multi_instances(connection('setProcess'))),

            iterate(processes(_processes_1), process(_process),
                send(connection('setProcess'),
                behaviour(sequence(instanciate(_process), send(connection('getSync')))))),

            sequence(iterate(processes(_processes_1), receive(connection('getSync'))),
            if_then(receive(connection(_connection), processes(_processes_2)),
                instanciate(multi_instances_with_sync(processes(_processes_2)))))])))).
```

### C.15 Deferred Choice

This pattern corresponds to the *deferred_choice* operator of the first layer.

### C.16 Interleaved Parallel Routing

```
process(
    interleaved_parallel_routing(connection(_connection_1), connection(_connection_2)),
    behaviour(
        sequence(receive(connection(_connection_1), connection(_connection_3)),
        sequence(receive(connection(_connection_2), connection(_connection_4)),
        deferred_choice([

            sequence(send(connection(_connection_1)),
            sequence(receive(connection(_connection_1)),
            send(connection(_connection_2)))),

            sequence(send(connection(_connection_2)),
            sequence(receive(connection(_connection_2)),
            send(connection(_connection_1))))])))))).
```

### C.17 Milestone

```
process(
    milestone_test(connection(_connection_1), connection(_connection_2)),
    behaviour(
        sequence(send(connection(_connection_1), connection(_connection_2)),
        receive(connection(_connection_2))))).

process(
    milestone_reached(connection(_connection_1)),
    behaviour(
        sequence(receive(connection(_connection_1), connection(_connection_2)),
        send(connection(_connection_2))))).
```

### C.18 Cancel Activity

```
process(
    cancel_activity(behaviour(_behaviour)),
    behaviour(
        if_then(\+ receive(connection('cancel')),
        /* The 'cancel' connection is managed by the evolver process */
            instanciate(behaviour(_behaviour))))).
```

### C.19 Cancel Case

```
process(
    cancel_case(behaviour(_behaviour)),
    behaviour(
        if_then_else(receive(connection('cancel')),
        /* The 'cancel' connection is managed by the evolver process */
            terminate,
            instanciate(behaviour(_behaviour))))).
```