Toward a Pi-Calculus Based Verification Tool for Web Services Orchestrations

Faisal Abouzaid

Ecole Polytechnique de Montreal,
2500-chemin de Polytechnique,
Montreal (Quebec) H3T 1J4, Canada
mohammed-faical.abouzaid@polymtl.ca

Abstract. Web services constitute a dynamic field of research about technologies of the Internet. WS-BPEL 2.0, is in the way for becoming a standard for defining Web services orchestration. To check the good behaviour of the produced compositions, but also to check equivalence between services, formalization is necessary. In this paper a contribution to the field of Formal Verification of Web services composition is presented using a $\pi$-calculus-based approach for the verification of composite Web services by applying model checking methods. We adopt the possibility of exploiting benefits from existing works by translating a Business Process Language such as BPEL to a system model of the $\pi$-calculus for which analysis and verification techniques have already been well established and there are existing tools for model checking systems. We therefore present the basis of a framework aimed to specification and verification, related to some temporal logic, of Web services composition.

1 Introduction

Web services are a typical example of technologies supporting Service Oriented Architectures (SOA). They allow to develop applications taking advantage of the Internet infrastructure. These distributed applications communicate by messages and use standardized protocols all based on XML. These Web services can be composed to form more complex applications. Such a composition is called an orchestration, if there is a coordinator between interacting services, and a choreography if not. These new concepts raise some challenges: How to be sure that a composite Web service works correctly? How to make sure that two or several Web service interact correctly and that the result of the interaction is acceptable in comparison with the initial specification? How to make sure that two services are compatible so that one can substitute them one for the other, where necessary? To answer these question a formal specification of the compositions is needed and justified by the need for developing reliable services which fulfill the requirements of the users.

These problems are not recent, nor specific to Web services. Techniques and many tools exist which make it possible to specify formally and to check properties on various systems. However, Web services have particular characteristics which require to be studied specifically. For example, concerning the composition of Web services, a service can dynamically establish a transaction with a service whose behavior is not
known in advance. An abstract representation of the behaviors of all the participants, in an adequate formalism, can help to reason in an automatic way. In the same way the formalism can help to design a reliable automatic composition of services. On another side, mobility is essential in the world of Web services: a service can communicate with customers whose address is not known in advance, but discovered during the execution. Some formalisms do not make it possible to express it correctly, the $\pi$-calculus does.

In this work we study the problem of formally design Web Services compositions expressed by means of a BPM language such as BPEL, using $\pi$-calculus. We compare different formalism and show that process algebras and in particular $\pi$-calculus, are well suited for the purpose of formally specifying such compositions. We aim to provide a theoretical and concrete framework to design and formally verify Web Services compositions. Our purpose is to develop a framework dedicated to verification and design of robust and complex Web services that is based on $\pi$-calculus formalism. The first step in this way is to provide a $\pi$-based semantics for Web services orchestrations expressed by the means of BPEL, which seems to be the most popular BPM language. Based on this semantics, we propose a mapping between main BPEL constructs and $\pi$-calculus. We present the $\pi$-logic, an extension of the $\mu$-logic, associated with $\pi$-calculus that allows us to express properties we want to verify. We illustrate our method by a significant example. Finally we present a basic architecture for our verification framework based on model-checking of this formalism.

In this paper, we will proceed in the following way: after having introduced Web services and their compositions in section two, we justify the need for formalization in section three, and we briefly discuss various formalisms used for this purpose. We then highlight the fact that process algebras are well suited for Web services formalization. In Section four, a brief presentation of the syntax and semantics of the $\pi$-calculus is given. The $\pi$-logic is also presented. In section five we present a mapping guide from BPEL to $\pi$-calculus and we show its relevance with a concrete example. In section six we discuss fundamental basis for a framework dedicated to formalization and verification of composite Web services. In section seven we conclude our presentation by reporting some conclusive remarks and a summary of the works. We also give some hints for future works.

\section{Web Services}
A Web service is an application provided by a service provider running on the internet and accessible to the customers through standard internet protocols. The W3C consortium defines a Web service as an application or a component that is identified by an URI and whose interfaces and links are described in WSDL, an XML-based language. Definitions of Web services can be discovered by other services by means of the UDDI protocol and they can interact directly with other services by using XML language and SOAP protocol.

\subsection{Orchestration and Choreography Languages}
Because each service provides a limited functionality, it is then necessary to compose these basic Web services in order to build a more complex composite service, providing
Several communities act in production of standard languages to describe specifications of orchestration but we focus our work on the language which seems to join together around him greater unanimity, namely BPEL (in the past BPEL4WS) [1]. This language supported by IBM and Microsoft, is in the way to become a standard. It is a programming language for implementing the execution logic of a business process based on interactions between the process and its partners. A BPEL process defines how multiple service interactions with partners are coordinated internally to achieve a business goal (orchestration).

While orchestration depends on a coordinating service, choreography is concerned with global, multiparty peer-to-peer collaborations interoperable between any type of components. WS-CDL is a language in which a choreography description is specified[2]. A WS-CDL specification describes the observable behavior of collaborations between services. WS-CDL is not a programming language; it is not executable.

3 Formalization of Web Services Orchestrations

3.1 Formalization

Formalization of composite Web services is justified by the need for the checking properties which attest the correct behavior of the whole system. Many works are devoted to the field of modeling and verifying Web services and their composition. An approach to process behaviour analysis and modelling of these software architectures can be constructed to provide tools for verification and validation of specified properties of the model against design specifications and implementations.

Various researches concerning Web services composition and based on various formalisms were proposed among such as Petri Nets [3], ASM (Abstract State Machines) [4], Automata with guards [5] or CCS [6].

3.2 Formalisms

Abstract State Machines The interest of the ASM lies in their expressivity and their simplicity. They make it possible to conceive achievable specifications that makes it possible to check directly on the model. However, this technique is not adapted to applications that process lot of data and it is the case of Web service which can exchange very significant volumes of information.

Petri Nets Petri nets make it possible to model events and states in a distributed system. They make it possible to simply express sequentiality, concurrency and asynchronous control based on events. They present some advantages for workflow modeling such as offering a formal semantics, though graphic. Other advantages are that their semantics is based on states and not only on events and there exist lot of tools for analysis.

However, Petri nets are not free from problems [7]. Thus certain difficulties appear with usage such as difficult to represent multiple instances of a sub-process or to represent some complex synchronization patterns (cancellation pattern (cleaner)).
Process Algebras Most of the problems raised by the previous techniques find their solution by using process algebras. They are used in various domains, thanks to their great capacity of modeling and to their relative simplicity of writing. They make it possible to describe the evolution and the behavior of realizable interactions within concurrent systems and they often are represented by programming languages reduced to a simple expression. [8]

They are suitable to describe Web services, because they offer a formal description of dynamic processes, which facilitates their automatic verification. They allow a great expressivity and provide constructions that are adapted to composition because they have compositional properties, Finally their textual notation is adapted to the description of real size problems, although it is less readable than transitions systems.

Process algebras are useful both at the time of design and for 'reverse engineering'. They offer the possibility of automatically generating skeletons of code thanks to a translation from the algebra to an executable language.

CCS The Calculus of Communicating Systems [9] is a calculus for describing static networks of processes that synchronize via channels. CCS doesn’t allow to pass channel-names as arguments while the main advantage of \( \pi \)-calculus is that it does.

4 the \( \pi \)-calculus

4.1 Introduction

The \( \pi \)-calculus [10], is a process algebra that can describe mobile concurrent computation in an abstract way. It provides a way to define labelled transition systems which can exchange communication channels as messages. Name communication, together with the possibility of declaring and exporting local names (scope extrusion), gives this calculus a great expressive power.

4.2 The \( \pi \)-calculus Syntax

We refer to [10] for a detailed description of the \( \pi \)-calculus, but we will give here a brief introduction to its syntax.

The \( \pi \)-calculus consists of a set \( N \) of names (for actions) and action prefixes \( \alpha \) that are a generalization of actions. An action prefix \( \pi \) represents either sending or receiving a message (a name), or making a silent transition (\( \tau \)). Actions syntax and the set of \( \pi \)-calculus process expressions are given in Table 1.

The meaning of each process defined above is as follows:

<table>
<thead>
<tr>
<th>Null: 0 is the deadlocked process which cannot involve with any transition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preixed sum: ( \sum_{i \in I} \pi_i P_i ) can proceed to ( P_j ) by taking the transition of the action prefix ( \pi_j ). Transitions and nondeterministic choices are described as preixed sums.</td>
</tr>
<tr>
<td>Parallel composition: ( P_1 \parallel P_2 ) is a process consisting of ( P_1 ) and ( P_2 ) which will operate concurrently, but may interact with each other through actions/co-actions.</td>
</tr>
<tr>
<td>Restriction: ( (\nu a)P ) means that the action/co-action ( a ) or ( \overline{a} ) in ( P ) can neither be observed outside, nor react with a or ( \overline{a} ) outside the scope of ( P ).</td>
</tr>
</tbody>
</table>
### Table 1. $\pi$-calculus Actions and Process syntax

<table>
<thead>
<tr>
<th>Action syntax</th>
<th>Process syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P ::= x(y)$ receive $y$ along $x$</td>
<td>$P ::= 0$ (null)</td>
</tr>
<tr>
<td>$\pi(y)$ send $y$ along $x$</td>
<td>$\sum_{i \in I} \pi_i P_i$ (Prefixed sum)</td>
</tr>
<tr>
<td>$\tau$ silent action</td>
<td>$P_1</td>
</tr>
<tr>
<td></td>
<td>$(\nu a)P$ (Restriction)</td>
</tr>
<tr>
<td></td>
<td>$[x = y]P$ (Match)</td>
</tr>
<tr>
<td></td>
<td>$!P$ (Replication)</td>
</tr>
</tbody>
</table>

**Match**: $[x = y]P$ behaves like $P$ if names $x$ and $y$ are identical, and otherwise like $0$.

**Replication**: $!P$ means that the behavior of $P$ can be arbitrarily replicated.

Structural operational semantics of the $\pi$-calculus is given by reaction and transition rules as shown in Table 2.

### Table 2. reaction and transition rules of the $\pi$-calculus

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU</td>
<td>$\tau . P + M \rightarrow P$</td>
</tr>
<tr>
<td>PAR</td>
<td>$P \rightarrow P'$</td>
</tr>
<tr>
<td>STRUCT</td>
<td>$P \equiv P' \rightarrow Q'Q \equiv Q'$</td>
</tr>
<tr>
<td>REACT</td>
<td>$\pi(y).P</td>
</tr>
<tr>
<td>RES</td>
<td>$(\nu x)P \rightarrow (\nu x)P'$</td>
</tr>
</tbody>
</table>

### 4.3 Model Checking in the $\pi$-$\mu$-Calculus

Some logics have been proposed [11], [12] to express properties of $\pi$-calculus processes. These logics are extensions, with $\pi$-calculus actions, name quantifications and parameterizations, of standard action based logics [13]. The $\pi$-logic [14] extends the modal logic introduced in [11] with some expressive modalities. Its syntax is given by:

$$\Phi ::= true \mid \sim \Phi \mid \Phi \& \Phi' \mid EX\{\mu\}\Phi \mid < \mu > \Phi \mid EF\Phi$$

The interpretation of the logic formulae is as follows:

- $P \models true$ holds always;
- $P \models \sim \Phi$ if and only if $notP \models \Phi$;
- $P \models \Phi \& \Phi'$ if and only if $P \models \Phi$ and $P \models \Phi'$;
- $P \models EX\{\mu\}\Phi$ if and only if there exists $P'$ such that $P \xrightarrow{\mu} P'$ and $P' \models \Phi$; This is the next operator.
\[ P \models < \mu > \Phi \text{ if and only if there exist } P_0, \ldots, P_n, n \geq 1, \text{ such that } P = P_0 \overset{\mu}{\rightarrow} P_1 \ldots \overset{\mu}{\rightarrow} P_{n-1} \overset{\mu}{\rightarrow} P_n \text{ and } P_n \models \Phi; \text{ This is the weak next operator.} \]

\[ P \models EF \Phi \text{ if and only if there exist } P_0, \ldots, P_n \text{ and } \mu_1, \ldots, \mu_n, \text{ with } n \neq 0, \text{ such that } P = P_0 \overset{\mu_1}{\rightarrow} P_1 \ldots \overset{\mu_n}{\rightarrow} P_n \text{ and } P_n \models \Phi. \]

 Derived operators can be defined like this:

- \( \Phi|\Phi' \) stands for \( \sim (\sim \Phi \land \sim \Phi') \). It is the OR operator;
- \( AX \{\mu\}\Phi \) stands for \( \sim EX \{\mu\} \sim \Phi \). This is the dual version of the strong next operator;
- \( [\mu]\Phi \) stands for \( \sim < \mu > \sim \Phi \). This is the dual version of the weak next operator;
- \( AG\Phi \) stands for \( \sim EF \sim \Phi \). This is the always operator, whose meaning is that \( \Phi \) is true now and always in the future.

**Example of formula:** For instance, deadlock freeness can be specified as follows:

\[
\text{NoDeadLock} = AG(\langle in?* > true \mid \langle out!* > true)
\]

It asserts that every time (always) it is possible to perform an input (on channel \( in \)) or an output (on channel \( out \)), ensuring that the system will never block. Note that we use the HAL syntax (see Section 5.3) which is slightly different from the previous one.

### 4.4 \( \pi \)-calculus Tools

The main existing tools for model-checking the \( \pi \)-calculus are the "Mobility Workbench (MWB)" [15] and the HAL toolset [14].

MWB is a model-checker for the polyadic \( \pi \)-calculus which allows handling and analysis of concurrent mobile systems. It allows checking of bisimulation which can be very useful for verifying Web services compatibility.

HAL is a promising tool which exploits a novel automata-like model which allows finite state verification of systems specified in the \( \pi \)-calculus. The HAL environment includes modules which support verification of behavioral properties of \( \pi \)-calculus agents expressed as formulae of suitable temporal logics.

For our example, we will use the HAL tool, which provides a Web based interface.

### 5 Mapping BPEL Constructs to \( \pi \)-calculus

#### 5.1 Web Service Composition Operators

Among basic constructions common to the majority of the Web service composition languages, one finds the following operations: service invocation (\( invoke \)), messages reception (\( receive \)), answer (\( reply \)), sequencality (\( sequence \)) or parallelism (\( flow \)). Mechanisms for compensation (\( compensate \)), and errors handling (\( fault handler \)), are also used.
Table 3. Mapping BPEL activities to $\pi$-calculus

<table>
<thead>
<tr>
<th>Basic Activity</th>
<th>$\pi$-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke</td>
<td>$\mathit{invoke} = \mathit{xs} \mathit{h} \mathit{~i}$</td>
</tr>
<tr>
<td>reply</td>
<td>$\mathit{reply} = \mathit{xs} \mathit{h} \mathit{~o}$</td>
</tr>
<tr>
<td>receive</td>
<td>$\mathit{receive} = \mathit{xs} \mathit{h} \mathit{~u}$</td>
</tr>
<tr>
<td>empty</td>
<td>$\mathit{empty} = \mathit{ys} \mathit{h} \mathit{~u}$</td>
</tr>
<tr>
<td>Concurrent Activities</td>
<td>$\mathit{flow}(A_1, A_2) = (\mathit{ys}')(\mathit{ys}')(A_1 \mathit{ys}''(\mathit{ys}'')(A_2 \mathit{ys}'''(\mathit{ys}''))) \mathit{ys}(\mathit{ys})$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{sequence}(A_1, A_2) = (\mathit{ys}')(\mathit{ys}')(A_1 \mathit{ys}''(\mathit{ys}'')) \mathit{ys}(\mathit{ys})$</td>
</tr>
</tbody>
</table>

5.2 Formalization

The first step in the formalization process is to map BPEL specifications into $\pi$-calculus processes. The mapping is required to provide explicit process representation behind that of the BPEL constructs activities and other process definitions. The semantics used here is based on the work presented in [16].

Note that $\mathit{u}$ denotes a set of values a process sends or receives. The $\pi$-process corresponding to a BPEL process executes an activity and flags out $y!\mathit{u}$ to signal its termination in order to support sequential composition.

It is very natural to map basic construct from BPEL to $\pi$-calculus. Thereby, an $\mathit{invoke}$ or $\mathit{reply}$ statement will be translated using an output action while a $\mathit{receive}$ statement will be translated using a parallel composition operator. A $\mathit{flow}$ activity can be translated using a parallel or a prefix ed operator (see Table 3 for the mapping).

A $\mathit{sequence}$ activity can be translated using a parallel or a prefix ed operator (see Table 4 ).

The mapping presented here is limited to some usual constructs. We refer to [16] for a detailed specification of fault handling and scope specification. More complex patterns from the world of workflow have been translated but not presented here.

5.3 An Example

In order to illustrate the use of the $\pi$-calculus and the HAL tool on a concrete way, we present the simple example of a company which receives from a customer information about an order. The company treats the order and then transmits it to the supplier who in his turn reply by sending information on the delivery, the company will retransmit
Table 4. mapping BPEL structured activities to $\pi$-calculus

<table>
<thead>
<tr>
<th>BPEL Activity</th>
<th>$\pi$-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(while condition =º exp = 'yes'º)</td>
<td>$(xy)_{y(\neg v_1): (j{\text{while}}(y}))$</td>
</tr>
<tr>
<td>switch(x; A_1; ...; A_n) =</td>
<td>$[x = a_1]A_1 \parallel (y)$, $[x = a_2]A_2 \parallel (y)$, $[A_0] \parallel (y)$</td>
</tr>
<tr>
<td>pick(x_1, i_1, A_1; x_2, i_2, A_2) =</td>
<td>$x_1(i_1)A_1(y)$, $x_2(i_2)A_2(y)$</td>
</tr>
</tbody>
</table>
| compensationHandler > | $\text{compensate} = z$,
| < activity > | $\text{onMessage partnerLink} = +$,
| < correlation set > | $\text{correlations}$
| < onMessage > | $\text{onMessage}$
| < compensationHandler > | $\text{onMessage}$
| < pick > | $\text{pick}$
| < standard-elements > | $\text{standard-elements}$
| < onMessage partnerLink > | $\text{onMessage}$
| < operations > | $\parallel$
| < correlations > | $\text{correlations}$
| < otherwise > | $\text{otherwise}$
| < sequence > | $\text{sequence}$
| < activity > | $\text{activity}$
| < while > | $\text{while}$
| < case > | $\text{case}$

these information to its customer. This company must thus define and document the trade process which it must implement.

We thus have 3 services that interact: A buyer who sends a purchase order number and a credit card number to the company. He receives in response a delivery date at his address. A seller (the company) receives the purchase order and the credit card number. He sends the parcel weight and the customer address to the supplier. Finally a shipper receives a parcel weight and a customer address. He returns a delivery date to the received address.

The BPEL specification and the mapping to $\pi$-calculus In this example, we give the BPEL specification of the Seller which plays the role of a coordinator between the three processes. The corresponding translation to $\pi$-calculus is also given. The verification of the correctness of the example was made using the HAL Tool. The syntax used in this tool is different from the standard one. The tool uses $x?y$ to denote an input, $x!y$ to denote an output, $\parallel$ for the composition operator and $(x)P$ to denote restriction. The syntax of the other operators is standard. The corresponding $\pi$-calculus translation of each BPEL component of our example is given by Table 5.

Finally the complete translation (simplified and optimized for verification reasons) is as follows:

The seller receives $po$, $cc$ and $MyChan$ on the channel $c_1$, from the customer. He sends the parcel weight $w$ and a customer channel name $z$ on the channel $c_2$ to the shipper, or a fault name $sf$ to the fault handler:

$$\text{Seller}(c_1, c_2, f) = (w)(z)(sf)$$

In order to verify the correctness of the whole system, we need to specify the other processes involved in the intercation. We abstract from details of the translation from BPEL to $\pi$-calculus.
Table 5. Example of a mapping

<table>
<thead>
<tr>
<th>BPEL</th>
<th>π-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;faultHandlers&gt;</code>&lt;br&gt;<code>&lt;catch faultName=&quot;NoDelivery&quot;&gt;</code>&lt;br&gt;<code>&lt;invoke partner=&quot;customer&quot;</code>&lt;br&gt;<code>portType=&quot;deliverPT&quot;</code>&lt;br&gt;<code>operation=&quot;sendRefusal&quot;</code>&lt;br&gt;<code>inputContainer=&quot;refusal&quot;/&gt;</code>&lt;br&gt;<code>&lt;/catch&gt;</code>&lt;br&gt;<code>&lt;/faultHandlers&gt;</code>&lt;br&gt;<code>&lt;flow&gt;</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>&lt;/flow&gt;</code> &lt;br&gt;<code>&lt;sequence&gt;</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>&lt;/sequence&gt;</code></td>
<td>( P(\pi f) = f? \tau.y!/u ) &lt;br&gt;fault handling</td>
</tr>
</tbody>
</table>

The shipper receives a request from Buyer:<br>`<invoke name="TransferOrder"`<br>`operation="TransferOrder"`<br>`inputVariable="DeliveryDate"`<br>`outputVariable="orderDelivery"`<br>`portType="requestDeliveryPT"`<br>`/>`<br>`<flow>`<br>`...`<br>`</flow>`<br>`<sequence>`<br>`...`<br>`</sequence>` | \( A_2 \equiv (y)\tau.(w, cc, mc) . y!/u \) <br>exchange operation <br>cc: credit card number <br>mc: channel for response |

The fault Handler receives a fault name and processes it. It also need to cancel all pending activities. To do this it sends a cancel message to the scope.<br>`FaultHandler (n) = f? n.y!/u` |

The whole system is represented as follows:<br>`ProcessOrder() = (c_1)(c_2)(MyChan)`<br>`Buyer(c_1, MyChan) || Seller(c_1, c_2) || Shipper(c_2, z) || FaultHandler(n)` |

This example is an illustration of mobility and of the need for managing it since the shipper does not know in advance the delivery address for the parcel.

π-calculus specification of the entire system

The buyer sends a purchase order number \( po \), a credit card number \( cc \) and a channel name \( MyChan \) to the company. He receives in response a delivery date \( d \) on the channel \( MyChan \). He can also send a fault name, \( bf \), to the fault handler.

\[
\text{Buyer}(c_1, MyChan, f) = (po)(cc)(MyChan)(bf) (c_1!(po, cc, MyChan). (MyChan!d.y!/u + f!bf))
\]

The shipper receives a parcel weight and channel name He returns a delivery date on the received channel. In case of error he sends a message ‘delivery fail’, \( df \), to the fault handler.

\[
\text{Shipper}(c_2, z, f) = (d)(df)(c_2!(w, z). (z!\tau.d!/u + f!df))
\]

The fault Handler receives a fault name and processes it. It also need to cancel all pending activities. To do this it sends a cancel message to the scope.

\[
\text{FaultHandler}(n) = f? n.y!/u
\]
5.4 Properties in $\pi$-logic

Here are for example, some temporal properties that assert of correct behavior of the system described previously:

Let $P_1$ be the property: "will the date of delivery be always sent to the customer after he requests it?''.

We can express it as follows:

$$P_1 = AG([MyChan?d] \land EF([MyChan!d](true))$$

And using the In HAL syntax:

$$P1 = AG([mc?d]EF<mc!d>true)$$

In the second example, let $P_2$ be the property: "will the number of the credit card never be revealed to other people but the salesman?''. We translate it in HAL syntax by:

$$P2 = AG([c1?cc]EF<c1!cc>true \land <c2!cc>false)$$

6 A Framework for the Verification of Web Services Compositions

We are actually working on the specification and design of an environment for developing complex reliable composite Web services for which verification tools will be integral part. This platform should also be able to propose tools for ‘reverse engineering’ i.e. the possibility of creating specific cations in BPEL starting from formal models expressed in $\pi$-calculus.

We briefly present, here, the architecture of the system being implemented and which should enable us to check the relevance of the suggested concepts. An ambitious objective is to design an integrated platform for composition of Web services. This one will consist of:

- An editor for generic specifications (nonrelated to a particular language),
- A module for mapping of WSDL and BPEL (and other) specific cations to $\pi$-calculus,
- A verification tool : an interface with existing tools, MWB or HAL for instance,
- A tool for ‘reverse engineering’ that allows designing real specific cations from formal definitions,
- A runtime environment.

Figure 1 shows the basic architecture of the verification framework. Such an application receives as input a specification expressed in one of several BPM languages (BPML, BPEL...) and rules (properties) that will be verified. The tool will make it possible to automatically translate these specifications into $\pi$-calculus processes. Properties will be then checked, using an appropriate model-checker and in case of fault, a trace of the faulty executions would be generated.

We are also interested in the study of equivalences between Web services. We are developing and working on a complete theory of equivalence: definition, comparison with bisimulation and algorithms. It is very important to check such equivalence that decide of compatibility between Web services, in order to substitute a service to another, in case of fault or any other problem. Our environment aims to provide such tools.
7 Conclusion

Web services offer a remarkable potential of development. The problems are numerous, in particular for the verification of composite services behavior or equivalence between services. The existing techniques can be adapted and used here. Petri nets or ASM were studied and used abundantly in this context. However, because it allows specifying mobility in an easy manner (it is possible to transmit channel names that then can be used by any process receiving them), and because compositional properties of process are fundamental, the $\pi$-calculus is clearly shown to be a very suitable tool for Web services formalization. We have shown with some examples that this aspect is essential in formalization of orchestrations.

We have shown how a business process modelling language for Web services orchestrations can be modelled to a specification language. We used BPEL as our BPM language and the $\pi$-calculus as our target specification language and therefore, we have translated a program written in BPEL to a system model in the $\pi$-calculus. Once the system model is achieved, it is possible to apply model checking techniques within existing tools, thus, allowing an automatic verification of BPEL specifications. Techniques we have applied here for translating a BPEL specification can be applied for any BPM language for gaining a model in the $\pi$-calculus. We have also shown that the use of analysis methods and tools based on this formal model ($\pi$-calculus) in some real life setting, is not an evident task.

The goal of this paper has been to provide theoretical and concrete basis for the architecture of an environment for analyzing Web services composition. This framework integrates a tool for model-checking of specifications expressed in various languages and a tool for ’reverse engineering’ making it possible to conceive formal orchestrations starting from formal specification, expressed in the $\pi$-calculus. For its durability, a significant asset of such a tool is that it will have to be independent from any specification languages for orchestrations which are in perpetual evolution. For this purpose we introduced a pi-based semantics for BPEL, inspired from [16] to express the main construct of compositions languages. We have adapted the semantics to fit the HAL
This semantics allows us to specify systems from the real world and thus to verify them, using model-checking techniques.

We consider this paper as the first step towards the definition of a formal framework for reasoning on orchestrations and choreographies. Several research directions open in front of us. We are continuing our work by working on algorithm for mapping business process specifications onto $\pi$-calculus instructions. We are also exploring ways to define some behavioural equivalences on Web services that could be used to study their compatibility. Finally and to complete this work, we have to take into consideration data mapping.

References