Towards a Formal Model for Agile Service Discovery and Integration

Hagen Overdick, Frank Puhlmann, and Mathias Weske

Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
{overdick,puhlmann,weske}@hpi.uni-potsdam.de

Abstract. As the fundamental web services technologies are becoming mature, web service composition, orchestration and choreography are gaining increasing attention. The movement from static interactions between already known partners as in BPEL to dynamically discovered and agile business partners is irresistible facing ever changing environments while aiming for specifically optimized collaborations. However, the techniques and models currently used lack fundamental formal foundations making them inadequate especially for modeling non-functional aspects. As a step toward the vision of dynamically discovered and agile processes, this paper proposes a formal approach to unambiguously define the syntax of service orchestrations and choreographies by representing key elements of service–oriented computing in a process algebra, the π-calculus. The results include a formal description of correlations in the context of service choreography as well as a formal representation of an orchestration pattern derived from BPMN and BPEL. The results provide a better understanding of service-based processes in terms of a formal algebra that will open the door for automated discovery and binding of potential business partners via service equivalence and mobility.

1 Introduction

As the fundamental web services technologies are becoming mature, web service composition, orchestration and choreography are gaining increasing attention. The movement from static interactions between already known partners as in BPEL [1] to dynamically discovered and agile business partners is irresistible facing ever changing environments while aiming for specifically optimized collaborations. However, the techniques and models currently used lack fundamental formal foundations making them inadequate especially for modeling non-functional aspects.

In this paper we try to remedy this situation by proposing a formal approach to unambiguously define service orchestrations and choreographies. It has recently been shown that the π-calculus, a process algebra, is capable to formally specify all workflow patterns [2]. Based on that work, this paper introduces formalisms to represent key elements of service oriented computing. For instance

a unique representation of correlations by merging correlation identifiers and response channels in the context of choreography as well as an orchestration pattern named *Event-based Rerouting*. The resulting formal and unambiguous characterization of processes in service-oriented environments is useful for a precise understanding of these processes, as well as enabling further research on automated discovery and binding via service equivalence and mobility [3, 4].

The paper is organized as follows. Section 2 introduces the π-calculus as well as formal workflow modeling by example, thus representing the state-of-the-art. Section 3 discusses the representation of services in the π-calculus, including correlations, invocation and data flow. Section 4 describes a formal service orchestration by example, using the results from section 3. It furthermore introduces a behavioral pattern named *Event-based Rerouting*, which is not contained in the workflow patterns collection so far [5]. The paper concludes with a short summary and discussion of related work.

2   The π-calculus

The π-calculus is a process algebra intended to describe mobile systems [6]. Mobile systems are made up of components which communicate and change their structure as a result of interaction. The core concepts of the algebra are processes and names. A π-calculus process is an entity which can communicate with other processes by the use of names. A name is a collective term for existing concepts like links, pointers, references, identifiers, etc. Names could be unbound (global) or bound to specific processes, i.e. they have a scope. The scope of a bound name can be dynamically expanded or reduced during the lifetime of the system by communicating names between processes. As this paper has a limited size, we can only introduce the notation of the π-calculus that will be used. Further details can be found in [6–9].

Syntax. The π-calculus consists of an infinite set of process identifiers denoted as $\mathcal{K}$ and another infinite set of names denoted as $\mathcal{N}$, where names define links. The processes are defined as:

$$P \quad ::= \quad M \mid P|P \mid \mathbf{v}zP \mid \, !P \; .$$

The composition $P|P$ is the concurrent execution of P and P, $\mathbf{v}zP$ is the restriction of the scope of the name z to P, which is also used to generate a unique, fresh name z and $!P$ is the replication operator that satisfies the equation $!P = P \mid !P$. M contains the summations of the calculus:

$$M \quad ::= \quad \mathbf{0} \mid \pi.P \mid M + M$$

where $\mathbf{0}$ is inaction, a process that can do nothing, $M + M$ is the exclusive choice between M and M, and the prefix $\pi.P$ is defined by:

$$\pi \quad ::= \quad \overline{x}\,\langle y\rangle \mid x(z) \mid \tau \mid [x = y]\pi \; .$$
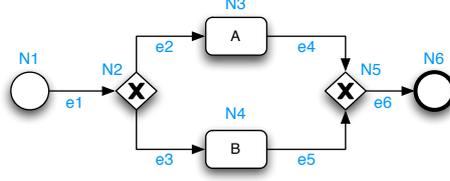
Fig. 1. A simple BPMN process.

The output prefix $\overline{x}\langle y\rangle.P$ sends the name y over the name x and then continues as P. The input prefix $x(z)$ receives any name over x and then continues as P with z replaced by the received name (written as $\{^{name}/_z\}$). The unobservable prefix $\tau.P$ expresses an internal action of the process, and the match prefix $[x=y]\pi.P$ behaves as $\pi.P$, if x is equal to y.

Throughout this paper, upper case letters are used for process identifiers and lower case letters for names. Two pre–defined, static names $\top$ and $\bot$ denote true and false. More additional process identifiers and names that represent special functions are introduced later on. Furthermore defined processes from the original paper on the π-calculus are used for parametric recursion, that is $A(y_1,...,y_n)$ [6].

The abbreviation $\sum_1^m(M)$ is used to denote the summation of m choices; e.g. $\sum_1^3(M_i) = M_1 + M_2 + M_3$. $\prod_1^m(P)$ is used to denote the composition of m parallel copies of P, e.g. $\prod_1^3(P) = P \mid P \mid P$. Also, $\{\pi\}_1^m$ denotes m subsequent executions of π, e.g. $\{\pi\}_1^3 = \tau.\tau.\tau$. All abbreviations could be used with an indexing variable, e.g. $\prod_{i=1}^3(d_i(x)) = d_1(x) \mid d_2(x) \mid d_3(x)$. Round brackets are used to define the ordering of a process definition. Given $\tau.P$ for instance, P might be expanded to $M + M'$ by using the summation rule from the π-calculus grammar. To avoid ambiguity, round brackets are put around the expanded symbol, e.g. $\tau.(M + M')$ instead of $\tau.M + M'$.

Semantics. The behavior of the π-calculus is defined by a reduction relation, $\longrightarrow$, on processes. The essence is captured in the axiom $(\overline{x}y.P_1 + M_1)\mid(x(z).P_2 + M_2) \longrightarrow P_1\mid P_2\{^y/_z\}$. The axiom states that whenever two processes can communicate, they will communicate and all other capabilities are rendered void. There exists other axioms, such as for structural congruence, which allow the conversion of π-calculus processes [9].

Representing Workflows in the π-calculus. We introduce the application of the π-calculus with a rather simple example shown in figure 1. The notation of the example is BPMN and the process will be mapped to π-calculus expressions. The process consists of two tasks A and B, which are placed between two XOR-gateways. As a first mapping task, all flow objects, i.e. events, gateways, tasks, are assigned an unique π-calculus process identifier. The start event is assigned to $N1$, the first XOR-gateway to $N2$, the task A to $N3$, the task B to $N4$, the

second XOR-gateway to $N5$, and the end event to $N6$. All sequence flows are mapped to a unique π-calculus name from $e1$ to $e6$ (see figure 1).

By referring to the generic structure for π-calculus processes that represent basic workflow activities [2], we can derive the next steps:

$$\{x_i\}_{i=1}^m.\{[a = b]\}_1^n.\tau.\{\overline{y_i}\}_{i=1}^o.\mathbf{0} \ . \tag{1}$$

Each basic activity consists of pre- and postconditions for routing the control flow as well as an unobservable action τ that represents the functional perspective of the activity. The precondition is split into two part: (1) $\{x_i\}_{i=1}^m$ denotes that the activity waits for m incoming names, and (2) $\{[a = b]\}_1^n$ denotes n additional guards that have to be true to execute the activity. The postcondition denotes the triggering of o outgoing names.

By using the definition from equation 1, we can define the π-calculus process $N1$ for the start event from the example:

$$N1 = \tau_{N1}.\overline{e1}.\mathbf{0} \ .$$

$N1$ has no preconditions; therefore the responding part from equation 1 is omitted. The functional part is represent by τ_{N1}, where the index $N1$ denotes that this τ belongs to the process $N1$. The postcondition of $N1$ signals $e1$, which will trigger the process $N2$.

The process $N2$ represents an XOR-gateway which routes the control flow depending on some conditions. $N2$ has the name $e1$ as a precondition and either $e2$ or $e3$ as postconditions. However, the generic structure for basic workflow activities from equation 1 only supports serial AND-split triggering by calling the names $y_1...y_o$ sequentially. A modified version of the structure that supports XOR, OR, as well as AND splits is as follows:

$$\{x_i\}_{i=1}^m.\{[a = b]\}_1^n.\tau.\prod_{i=1}^o [c = \top]\overline{y_i}.\mathbf{0} \ . \tag{2}$$

For each possible postcondition, a parallel process part is introduced by $\prod_{i=1}^o [c = d]\overline{y_i}.\mathbf{0}$. Each part has a match prefix which either enables (true) or disables the postcondition (false). If the match prefix is true, the corresponding name is signaled. All other parts are discarded. A π-calculus process that behaves like an OR/XOR-split as a postcondition is then defined by i.e. $[c = \top]\overline{y_1}.\mathbf{0} \mid [d = \top]\overline{y_2}.\mathbf{0}$. If no match prefixes overlap, the split behavior is XOR, otherwise OR. An XOR split could also be realized by using the summation operation ($\sum$) instead of concurrency. That would also allow non-deterministic choices. An AND-split simply has no match prefixes at all: $\overline{y_1}.\mathbf{0} \mid \overline{y_2}.\mathbf{0}$.

By using the definition from equation 2, $N2$ can be defined:

$$N2 = e1.\tau_{N2}.([c_1 = \top]\overline{e_2}.\mathbf{0} \mid [c_2 = \top]\overline{e_3}.\mathbf{0})$$

As the business process diagram from the example contains no conditions, we assume two global names c_1, and c_2 that hold the expressions. Per definition

of the exclusive choice pattern, we assume c_1 and c_2 to be disjoint; i.e. if $c_1 = \top \Rightarrow c_2 = \bot \wedge c_1 = \bot \Rightarrow c_2 = \top$.

The π-calculus processes for the tasks A and B from the example are straightforward as they simply contain one pre- and postcondition each:

$$N3 = e2.\tau_{N3}.\overline{e4}.\mathbf{0}, N4 = e3.\tau_{N4}.\overline{e5}.\mathbf{0} \ .$$

The XOR-join gateway $N5$ is rather simple, as we can adapt the pattern simple merge from [2]. As the simple merge pattern has only one precondition per definition (i.e. $d.\tau_d.D'$), we need an additional π-calculus process, which triggers $N5$ if either $e4$ or $e5$ are signaled:

$$N5 = \mathbf{v}x(e4.\overline{x}.\mathbf{0} + e5.\overline{x}.\mathbf{0} \mid x.\tau_{N5}.\overline{e6}.\mathbf{0}) \ .$$

The left hand side of the definition contains the additional process. It generates a fresh name x, which is used to signal the precondition of the XOR-join, which is contained at the right hand side. As the name x is unique to $N5$, only $e4$ or $e5$ could trigger the precondition and enable $\tau_e 6$.

The last process, $N6$, is now trivial:

$$N6 = e6.\tau_{N6}.\mathbf{0} \ .$$

3 Representing Services in the π-calculus

We have shown that the few concepts of the π-calculus are readily suited to model workflows, which can also be used to describe complex orchestrations in the context of service–oriented architectures. Indeed, all workflow patterns described in [5] can be formalized with basic π-calculus expressions, as shown recently [2]. However, so far we have only discussed how the routing of activities, i.e. web service calls, can be realized in the π-calculus in a straightforward way. This discussion will be continued in the next section. First, the invocation and providing of services in the π-calculus will be introduced.

3.1 Invoking Services

As with activities, services are also invoked by names in the π-calculus. In the BPMN notation, message flow is used for this purpose, whereas sequence flow controls the routing within a process. We define two distinct subsets from the set of names, $\mathcal{N}_\mathcal{S} \subset \mathcal{N}$ that contains all names used for service communication, and $\mathcal{N}_\mathcal{C} \subset \mathcal{N}$ that contains all names used for control flow routing. The intersection of $\mathcal{N}_\mathcal{S}$ and $\mathcal{N}_\mathcal{C}$ has to be empty: $\mathcal{N}_\mathcal{S} \cap \mathcal{N}_\mathcal{C} = \varnothing$. To enable different kinds of service invocations, we furthermore extend the τ prefix of the π-calculus with a placeholder denoted as $\square$. Each τ inside an activity definition that contains service invocations is then replaced by the placeholder.

Correlations. A common problem in the area of service–oriented computing is the description of correlations between service invokers and service providers. Usually, some kind of correlation identifier is placed inside each request and reply [10]. The invoker as well as the provider have to take care to match all requests. In the π-calculus, the unique identifier of a request is also the channel used for reply from the service. By merging these two concepts, a clear representation of the correlations is straightforward. A new unique identifier is a π-calculus name from the set of $\mathcal{N}_\mathcal{S}$ which is created with the $\mathbf{v}$ operator. The application is described below.

Synchronous Invocation. A synchronous invocation of a service contains an outgoing message to the service as well as an incoming response:

$$\Box = \mathbf{v}c(\overline{w}\langle c\rangle.c.\tau) \ .$$

A service that is bound to $w \in \mathcal{N}_\mathcal{S}$ is invoked by $\overline{w}$ with a fresh name that acts as a correlation identifier as well as a response channel. The process holds until the response is signaled over the name c. Thereafter, it continues as τ. The whole placeholder process part $\Box$ can then replace a τ in a process definition to represent a service invocation, e.g. $N3$ from the previous section is extended to $N3 = e2.((\mathbf{v}c)\overline{w}\langle c\rangle.c.\tau_{N3}).\overline{e4}.\mathbf{0}$.

Asynchronous Invocation. The asynchronous invocation of a service is achieved by separating the transmission and the receipt of an invocation into different processes. A placeholder $\Box_1$ is defined by $\overline{w}\langle c\rangle.\tau$ and a second one as $\Box_2 = c.\tau$. The two placeholders can be placed in two different processes which must share the fresh name c.

By reconsidering the service invocation descriptions, it shows that all service invocation in the π-calculus is indeed asynchronous. This is essential, as the name of the response channel must be transmitted to the service in order to work. The notation of synchronous invocation simply means that there is no additional activity between the invocation and response.

3.2   Providing Services

The π-calculus provides an easy notation for describing state preserving services, where the correlation is bound to a unique π-calculus name. A π-calculus process that acts as a service is defined by:

$$SERVICE = !w(c). \cdots .\overline{c}.\mathbf{0} \ .$$

Each time a request for $SERVICE$ is received via the global name w, the service is replicated (comparable to instantiated). The service can be as complex as required (represented by $\cdots$). After all computation has been done, the response is sent. As service invocations in the π-calculus are generally asynchronous, there is no differentiation between a synchronous and asynchronous service. The service always requires a response channel to work.

Interestingly, by using a unique name for the correlation as well as response channel, the service also holds the state between complex operations. As the name is unique and bound to the invoker, there can be no confusion. An example is a service that can be invoked again with the unique name, those interacting in a complex choreography:

$$SERVICE = !w(c). \cdots . \bar{c}.c. \cdots . \bar{c}. \cdots .$$

When $SERVICE$ receives an invocation at the unique name c, it knows exactly which replication instance is triggered, if that instance exists. As only the invoker has access to c, no confusion is possible. A concrete, exemplary client/service choreography can be denoted as follows:

$$SERVICE = !w(c).\tau_{S1}.\bar{c}.c.c.\tau_{S2}.\bar{c}.\mathbf{0}$$
$$\square_{CLIENT} = \mathbf{v}c(\overline{w}\langle c\rangle.c.\tau_{C1}.\bar{c}.\tau_{C2}.\bar{c}.c.\tau_{C3}) .$$

The client signals an initial invocation to $SERVICE$ with a fresh name c by $\overline{w}\langle c\rangle$. $SERVICE$ then spawns of a new instance by replication which does some internal computation represented by τ_{S1} and afterward replies on the channel c. The client in response computes something denoted by τ_{C1} and sends two requests over the name c. The service then executes τ_{S2} and responds again on the channel c. After the client received the last response, it does some internal computation τ_{C3} and finishes.

3.3 Supporting Data

One important characteristics of service invocations is the transmission and reception of messages. Messages, or arbitrary data-structures, are also represented by names in the π-calculus. A name thereby holds a reference to a π-calculus process which represents a data structure. For the ease of representation in this paper, we simply denote the type of the data structure with a colon and an XML-type. For instance, $mess : string$ denotes a name $mess$ that references a process that contains a data structure for text strings. By using the polyadic extension of the π-calculus, we can group different names and transmit or receive them with one output or input prefix. An example that invokes a service at the name $s \in \mathcal{N}_{\mathcal{S}}$ with a parameter of the type string and receives a response containing a double as well as a date is denoted as:

$$\square = \mathbf{v}c(\bar{s}\langle c, request : string\rangle.c(rate : double, validuntil : date).\tau) .$$

4 π-calculus Orchestration Refinements by Example

Now, enough background is given to show the mapping of a more complex example. Again, the process is visualized with the BPMN (figure 2). To make the

example more interesting, the modeled process consists of several workflow patterns, namely sequence, exclusive choice, simple merge, deferred choice as well as a new pattern called *Event-based Rerouting*. The example describes a process orchestration with web service interaction, both synchronous and asynchronous.
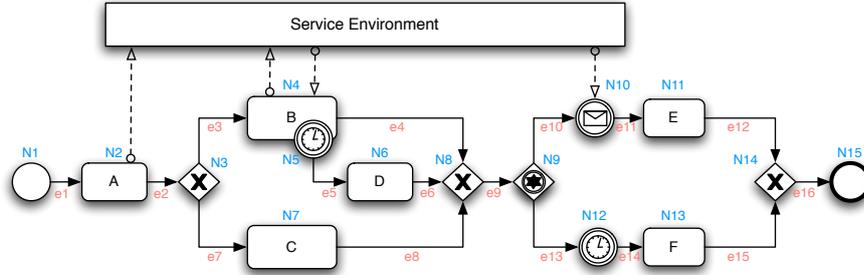

Fig. 2. A BPMN process consisting of several patterns.

The process starts with task A making an asynchronous call to a web service. Thereafter either task B or task C are executed. Task B contains a synchronous web service invocation and has an attached intermediate timer event, which interrupts task B after a certain amount of time has passed and B still has not finished, and continues the execution with task D. After task B, C, or D have been executed, a deferred choice is made. If an answer from the asynchronous call of task A is received before another timeout is reached, task E is executed, otherwise task F.

4.1   Event-based Rerouting

Before the actual mapping of the example process (figure 2), let us first go back to the tasks, as they are modeled in BPMN. As shown in figure 3(a), all tasks have an implicit hook for intermediate events, either directly attached as shown, or logically attached, i.e. via a surrounding sub-process. On occurrence of such an event, the control flow is immediately rerouted (*alternative* instead of *done* in figure 3(a)). This is currently not captured by any workflow pattern, as the rerouting may take place, before the completion of the activity modeled by the task. This is especially true in the SOA context, as web services cannot be canceled easily. Without this pattern, e.g. a timeout for a synchronous web service call can not be modeled. As mentioned above, we propose the name *Event-based Rerouting*.

Looking at the equations 1 and 2, one quickly realize that this immediate rerouting is not supported. Consequently, we need to adapt our equation. Continuing the procedure of expressing each BPMN-node as a π-process, the intermediate event will be expressed by a π-process as well. This event-process

(a) BPMN-Task with event

(b) Mapping to π-calculus
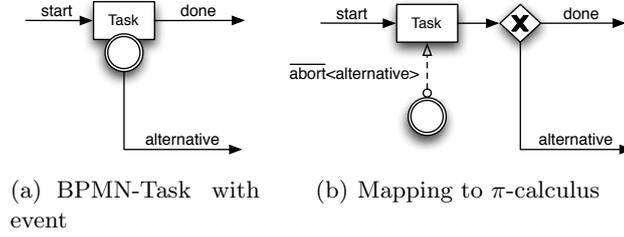
Fig. 3. Intermediate events in BPMN and their mapping to π-calculus

can send the task an alternative route via the *abort* name, which is immediately taken. Figure 3(b) tries to visualize how we propose to model a *Task* with the π-calculus as shown in the following. As before, the $\square$ denotes a placeholder for the actual process description. As $\square$ can simply be expanded, no new semantics are introduced by this abbreviation.

$$TASK(start, done, abort, \square) = start.TASK_{EXECUTE}$$
$$+ \, abort(alternative).\overline{alternative}.\mathbf{0} \quad (3)$$

Equation 3 defines a task via the process $TASK$, having the names *start*, *done*, and *abort* as parameters, plus the actual implementation denoted by $\square$. $TASK$ waits either for a signal on *start* to start the execution $TASK_{EXECUTE}$ or receiving an *alternative* name on *abort*, which is immediately signaled.

$$TASK_{EXECUTE} = (\mathbf{v}continue)(TASK_{ABORT} \mid TASK_{IMPL}) \quad (4)$$

$TASK_{EXECUTE}$ (equation 4) defines a private name *continue* and executes $TASK_{ABORT}$ and $TASK_{IMPL}$ in parallel.

$$TASK_{ABORT} = abort(alternative).\overline{alternative}.\overline{continue}\langle\bot\rangle.\mathbf{0}$$
$$+ \, \overline{continue}\langle\top\rangle.\mathbf{0} \quad (5)$$

$TASK_{ABORT}$ (equation 5) tries to receive an *alternative* name over *abort* to signal $\overline{alternative}$ and send $\bot$ via $\overline{continue}$. Alternatively, $TASK_{ABORT}$ sends $\top$ via $\overline{continue}$. The reduction semantics of the π-calculus guarantees that the decision can not be made until either *continue* is read by $TASK_{IMPL}$ (see below) or $\overline{abort}$ is written by a different process, i.e. an event node.

$$TASK_{IMPL} = \square.continue(flag).([flag = \top]\overline{done}).\mathbf{0} \quad (6)$$

$TASK_{IMPL}$ (equation 6) first executes the activity represented by $\square$. It then reads *continue*. If no read on *abort* occurred by this time, $TASK_{ABORT}$ is now

able to send $\top$ over $\overline{continue}$, otherwise a $\bot$ is transmitted. A $\top$ represents normal execution, so iff $\top$ is read, $\overline{done}$ is written. Here, $\bot$ represents the occurrence of an intermediate event, which is already handled by that time, thus the process simply ends.

4.2 Mapping to π-calculus

Now, let us start the mapping of the example process (figure 2) to π-calculus. Again, our approach is to model each BPMN-node as an individual process.

First, the gateways:

$$N3 = e2.\tau_{N3}.([c_{e3} = \top]\overline{e3} \mid [c_{e7} = \top]\overline{e7})$$
$$N8 = \mathbf{v}x(e4.\overline{x}.\mathbf{0} \mid e6.\overline{x}.\mathbf{0} \mid e8.\overline{x}.\mathbf{0} \mid x.\overline{e9}.\mathbf{0})$$
$$N14 = \mathbf{v}x(e12.\overline{x}.\mathbf{0} \mid e15.\overline{x}.\mathbf{0} \mid x.\overline{e16}.\mathbf{0})$$

All gateways, except for the deferred choice, are expressed with equation 2, just as before.

Next, the trivial tasks:

$$N6 = TASK(e5, e6, env_{ABORT}, \tau_{N6})$$
$$N7 = TASK(e7, e8, env_{ABORT}, \tau_{N7})$$
$$N11 = TASK(e13, e14, env_{ABORT}, \tau_{N11})$$
$$N13 = TASK(e14, e15, env_{ABORT}, \tau_{N13})$$

The trivial tasks ($N6, N7, N11, N13$) are represented by our new $TASK$ process, abstractly executing a τ. The name env_{ABORT} represents a name provided by the environment to signal a global abort, e.g. the shutdown of the workflow engine.

As we just introduced the concept of environmental names, let us now look at the start- and end-event of the process:

$$N1 = TASK(env_{START}, e1, env_{ABORT}, \tau_{N1})$$
$$N15 = TASK(e16, env_{DONE}, env_{ABORT}, \tau_{N15})$$

These are trivial tasks as well, but interact with the environment via env_{START} and env_{DONE}. These names provided the integration to a workflow engine to signal the start of the execution as well as the completion between the process and the workflow engine.

Next, the asynchronous web service invocation in $N2$:

$$N2 = TASK(e1, e2, env_{ABORT}, \overline{w_{req1}}\langle w_{resp1}\rangle.\tau_{N2})$$

The implementation follows the explanation from section 3.1. The web service is called via $\overline{w_{req1}}$ and the unique response name w_{resp1} is passed along. All of this is encapsulated within a $TASK$ process.

$N4$ is a synchronous web service invocation with an attached intermediate timeout event $N5$:

$$N4 = TASK(e3, e4, abort_{N5}, \overline{w_{req2}}\langle w_{resp2}\rangle.w_{resp2}.\tau_{N4}).\mathbf{0}$$
$$N5 = env_{TIMEOUT N5}.\overline{abort_{N5}}\langle e5\rangle$$

As before, the response name w_{resp2} is passed along the service call. The only difference to $N2$ is the immediate read on the response name. Also, note that the abort-name is different than before to allow for a communication between $N5$ and $N4$ in case the timeout gets triggered. The actual timeout is once again triggered by the environment ($env_{TIMEOUT N5}$). This is an application of the *Event-based Rerouting* pattern described before.

Last but not least, the deferred choice $N9$:

$$CHOICE_{N9,N10,N12} = e9.(w_{resp2}.\overline{e11}.\mathbf{0} + env_{TIMEOUT N12}.\overline{e14}.\mathbf{0})$$

Notice, that the nodes $N10$ and $N12$ are not modeled explicitly, but as part of the $CHOICE_{N9,N10,N12}$ process, hence the name. Again, we model the actual timeout as a signal by the environment. The simplicity of the equation is yet another good example of the expressiveness of the π-calculus.

5   Conclusion

In this paper, we have sketched how the π-calculus can be used in the service–oriented domain. Starting from our work on workflow pattern in π-calculus, the representation, orchestration, and choreography of services has been discussed. Interestingly, the π-calculus concept of mobility, which is based on communicating names that can be used as interaction channels, proved to be very useful to formally represent unique correlations. Furthermore, we introduced formal representations for service invocation, both for the client and the provider. In section 4 the orchestration of services was refined by introducing a formal representation of a pattern known from notations like BPMN or BPEL. We named it *Event-based Rerouting* pattern and introduced the precise semantics of a task containing it. Finally, the formalization of an example containing the new pattern as well as another common pattern in service–oriented orchestrations, a deferred choice modeled with an event–based gateway in the BPMN, has been discussed.

Related Work. Lucas Bordeaux and Gwen Salaün wrote a survey about using process algebra for web services [3]. They argued that the formal reasoning capabilities of process algebras are well suited in the service–oriented domain. Especially (relaxed) equivalence properties for services participating in a choreography and some kind of soundness for orchestration can be formally proved by using process algebra. They also concluded that the name passing notation of the π-calculus is definitely of interest in the context of web services. However, there exist no XML-based technology that allows for name passing yet.

Further discussions about behavioral compatibility of web–services can be found in [11]. L.G. Meredith and Steve Bjorg wrote a journal article about using mobile process algebra in the context of formal service descriptions where they emphasized the use of behavioral types as a new kind of service discovery mechanisms [4]. There have also been investigations on extending the π-calculus for representing and reasoning about long–running transactions in component–based distributed applications like web–service platforms [12]. A more practical approach of using CCS [13] to formalize web service choreography can be found in [14].

References

1. BEA Systems, IBM, Microsoft, SAP, Siebel Systems: Business Process Execution Language for Web Services Version 1.1. (2003)
2. Puhlmann, F., Weske, M.: Using the Pi-Calculus for Formalizing Workflow Patterns. In van der Aalst, W., Benatallah, B., Casati, F., eds.: BPM 2005, volume 3649 of LNCS, Berlin, Springer-Verlag (2005) 153–168
3. Bordeaux, L., Salaün, G.: Using Process Algebra for Web Services: Early Results and Perspectives. In Shan, M.C., Dayal, U., Hsu, M., eds.: TES 2004, volume 3324 of LNCS, Berlin, Springer-Verlag (2005) 54–68
4. Meredith, L., Bjorg, S.: Contracts and Types. Communications of the ACM **46** (2003) 41–47
5. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14** (2003) 5–51
6. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part I/II. Information and Computation **100** (1992) 1–77
7. Milner, R.: The polyadic π–Calculus: A tutorial. In Bauer, F.L., Brauer, W., Schwichtenberg, H., eds.: Logic and Algebra of Specification, Berlin, Springer-Verlag (1993) 203–246
8. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press, Cambridge (1999)
9. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Paperback edn. Cambridge University Press, Cambridge (2003)
10. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Boston (2003)
11. Bordeaux, L., Salaün, G., Berardi, D., Mecella, M.: When are Two Web Services Compatible? In Shan, M.C., Dayal, U., Hsu, M., eds.: TES 2004, volume 3324 of LNCS, Berlin, Springer-Verlag (2005) 15–28
12. Bocchi, L., Laneve, C., Zavattaro, G.: A Calculus for Long–Running Transactions. In Najm, E., Nestmann, U., Stevens, P., eds.: FMOODS 2003, volume 2884 of LNCS, Berlin, Springer-Verlag (2003) 124–138
13. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
14. Brogi, A., Canal, C., E.Pimentel, Vallecillo, A.: Formalizing Web Service Choreographies. In: Proceedings of First International Workshop on Web Services and Formal Methods. Electronic Notes in Theoretical Computer Science, Elsevier (2004)