Formalizing Service Interactions

Gero Decker and Frank Puhlmann

Business Process Technology Group
Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
{decker,puhlmann}@hpi.uni-potsdam.de

Abstract. Cross-organizational business processes are gaining increased attention these days, especially with the service oriented architecture (SOA) as a realization for business process management (BPM). In SOA, interaction agreements between business partners are defined as choreographies containing common interaction patterns. However, complex interactions are difficult to specify, basically because a formal, common standard supporting all interaction patterns is missing. This paper takes the next steps by investigating formal representations of service interaction patterns in π-calculus and Petri nets. Since dynamic routing based on mobility is a central aspect of choreographies we argue that π-calculus is better suited for formalizing service interactions than place/transition and colored Petri nets.

1 Introduction

Service-oriented architectures (SOA) as a realization for business process management (BPM) aim at closely supporting business processes within a company and between business partners [1,2]. Services are employed to perform tasks within these processes and processes themselves can be exposed as services. We distinguish between orchestrations where one business partner enacts a set of services in a given order and choreographies which represent the interaction protocols between several business partners [3]. In a setting where the different business partners encapsulate their business logic as services, service interactions are at the center of attention. A lot of effort has been undertaken to identify the most common interaction scenarios from a business perspective, which have been published as *Service Interaction Patterns* in [4]. Barros et al categorize the patterns according to the number of participants in an interaction (bilateral vs. multi-lateral), the maximum number of exchanges (single-transmission vs. multi-transmission interactions) and whether the receiver of a response is necessarily the same as the sender of a request (round-trip vs. routed interactions).

The service interaction pattern are only described textually, together with business examples and design choices. The authors also come up with implementation examples using BPEL [5] and other standards from the WS-* stack. However, the textual descriptions do not allow choreographies to be modeled

else then by using textual descriptions again. The BPEL examples lack support for different service interaction patterns, those leaving the modeler with only a subset of possibilities. Furthermore, both kinds of descriptions lack support for formal reasoning on interaction properties like conformance, reliability, or deadlock freedom.

To overcome the limitations of expressiveness in existing notations and to allow formal reasoning, we propose the use of formal representations of service interaction patterns. When looking into BPM literature we see that Petri nets in all their different flavors dominate the research community. Therefore, we investigate if Petri nets are suitable for formalizing the service interaction patterns. Furthermore, in the last years π-calculus, a general purpose process algebra, came up in the BPM domain. We analyze if interaction and mobility, the core aspects of π-calculus, are also at the heart of the service interaction patterns. The paper is organized as follows. We start by investigating related work. This is followed by discussing the interaction patterns in Petri nets, where we investigate serious problems. However, these problems can be solved by using the π-calculus as shown in the next section. Finally, a conclusion is drawn and an outlook is given.

2 Related work

Recently several papers have been published that deal with formalizing web service choreographies, e.g. [6,7], or Busi et al [8]. All these approaches are based on process algebras other then π-calculus. Busi et al argue that mobility, a key feature of the π-calculus, is not needed for describing service choreographies. They assume that all interaction participants are known at design-time. They introduce their own process algebra for service orchestration and choreography and show how conformance between a set of orchestrations and a choreography can be proved. Petri net based approaches from Martens [9] or van der Aalst et al [10] make the same assumptions. Moreover, Petri nets already fail in representing all workflow patterns [11], leading to the development of a new orchestration language called YAWL [12].

However, all these publications and standards like WS-CDL [13] consider only one-way- and simple request-response-interactions. This is heavily criticized by Barros et al in [3]. Puhlmann and Weske have formalized all the workflow patterns [11] using the π-calculus in [14]. This allows for translating a huge range of service orchestrations into π-processes. Puhlmann et al have already sketched in [15] how π-calculus could be used for formalizing service invocations and represent correlations. There has not been a formalization of the service interaction patterns so far.

3 Interaction Patterns in Petri nets

Petri nets form a strong theoretical foundation for traditional intra-organizational business processes [16]. Many scientific publications regarding for instance dif-

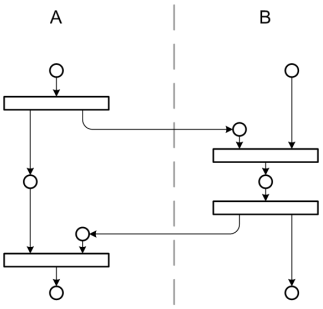

Fig. 1. Send/Receive as State/Transition net.

ferent kinds of correctness, adaptability, change, or performance ground this assertion [17,18,19,20]. Petri nets have even been used to model simple forms of interacting business processes [10,9]. However, during our investigations on formalizing service interaction patterns [4] in Petri nets we found several critical issues. This section summarizes and discusses them.

3.1 Petri nets

Petri nets as originaly introduced in [21] are bipartite graphs consisting of places and transitions that are connected via directed edges. Places can contain tokens that can be consumed and produced by transitions. Places are visualized by circles, transitions by rectangles, and tokens by small solid circles. Transitions and places can be assigned to different actors, which are graphically represented by swim lanes. Tokens represent control flow as well as data flow. Tokens passed from one actor to another will represent messages in our formalizations. More information about different kinds of Petri nets used in BPM can be found for instance in [16], colored nets are discussed e.g. in [22].

3.2 Simple Interaction Patterns

A send/receive pattern can be modeled in Petri nets as shown in figure 1. Now imagine that several instances of the interaction take place at the same time. In this case more tokens flow through the net. However, there is no information about which tokens belong to the same instance (if we use simple place/transition nets). Therefore, these simple nets do not support correlation. Figure 2 illustrates how the one-from-many receive pattern in the basic version (stop condition = success condition = n messages received) would be modeled using Petri nets.

Transitions have to be enabled before they can fire. I.e. n tokens have to be available in place p1 and one token in p2 before t1 can consume these tokens and produce a token on place p3. Once t1 has fired t2 cannot fire any more because there is no token in p2. We have to stress the *can fire* because t1 is *not forced*

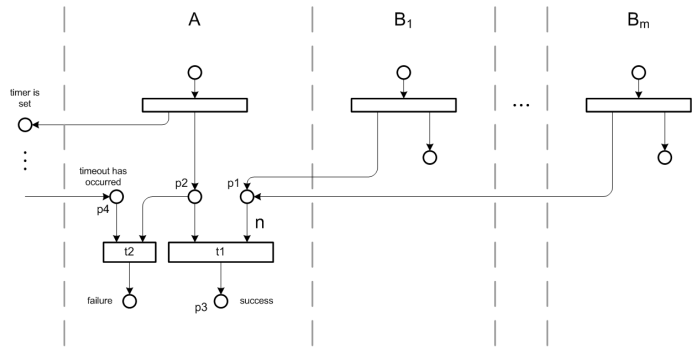

Fig. 2. One-from-many Receive.

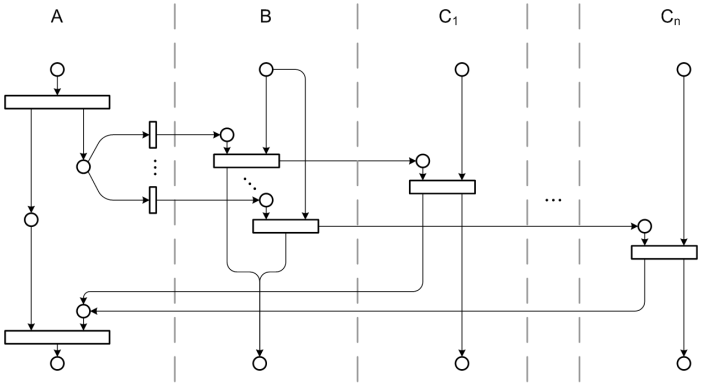

Fig. 3. Request with Referral as Place/Transition net.

to fire. It could happen that t1 could fire but does not and then the time out occurs (a token is produced in place p4) and t2 fires.

This problem also comes up in the racing incoming messages pattern. This pattern defines that the continuation is activated *as soon as* the first message arrives. However, in the case of Petri nets the transition is not forced to consume the first token but could also consume the second instead. There is a solution to this problem though: Timed Petri [16] nets introduce a notion of temporal ordering of tokens.

3.3 Issues Regarding Complex Interaction Patterns

Figure 3 shows a formalization for the *request with referral pattern* using a place/transition net. Since we cannot encode anything in the token we have to indicate the third interaction partner by putting a token into the appropriate place for B. As we might face a huge number of potential interaction partners,

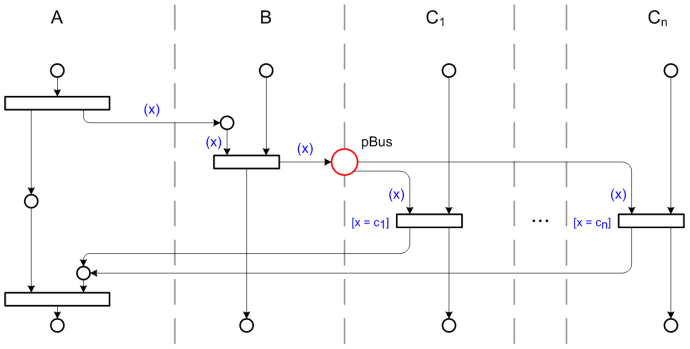

Fig. 4. Request with Referral as Colored Petri net.

we have to model a huge number of places and transitions. However, this is not feasible.

Another approach is to encode an identifier of the third interaction in the token, which is supported by colored Petri nets. That way, we introduce something like a bus (see place pBus in figure 4) where every potential interaction partner is connected to. Guard conditions prevent a transition from consuming tokens that were not meant to be received by the corresponding actor. Figure 4 illustrates the idea. We have drastically reduced the number of places and transitions compared to the previous formalization using state/transition nets. Nevertheless, there are still some drawbacks to this approach:

- We still have to model connections between every set of potential interaction partners. In the web there might be thousands or more potential interaction partners in every interaction.
- Due to the different combinations of interaction partners in an interaction the arc and guard expressions might become very complex.
- We cannot model change. Because of the static nature of Petri nets we have to know every potential interaction at design-time. This does not reflect the reality where new potential interaction partners appear and also disappear at run-time.

As a conclusion, ten out of the thirteen service interaction patterns incorporate sending messages. In all of these cases it is explicitly stated that the receiver might not be known at design-time. Even using colored Petri nets we have seen that modeling this mobility with the nets is not feasible when dealing with many potential interaction partners (which has to be assumed). The next section will introduce a solution based on the π-calculus semantics of mobility, that overcomes the limitations of Petri nets.

4 Formalizing the Interaction Patterns using Pi-Calculus

At the center of π-calculus we deal with processes that interact with each other. The communication channels as well as the messages sent over these channels are called names. Channels can be passed as messages to other processes and be used for interaction later on. This capability is called link mobility. It allows smart solutions for formalizing the service interaction patterns. The following subsections introduce how.

4.1 The Pi-Calculus

The π-calculus is an algebra for the formal description and analysis of concurrent, interacting processes with support for link passing mobility. It is based on names and interactions used by processes defined according to [23]. The syntax of the π-calculus processes is given by:

$$\begin{aligned} P ::=& M \mid P|P' \mid \mathbf{v}zP \mid !P \\ M ::=& \mathbf{0} \mid \pi.P \mid M + M' \\ \pi ::=& \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi \ . \end{aligned}$$

The informal semantics is as follows: $P|P'$ is the concurrent execution of P and P', $\mathbf{v}zP$ is the restriction of the scope of the name z to P, and $!P$ is an infinite number of copies of P. $\mathbf{0}$ is inaction, a process that can do nothing, $M + M'$ is the exclusive choice between M and M'. The output prefix $\overline{x}\langle\tilde{y}\rangle.P$ sends a sequence of names $\tilde{y}$ over the co-name $\overline{x}$ and then continues as P. The input prefix $x(\tilde{z})$ receives a sequence of names over the name x and then continues as P with $\tilde{z}$ replaced by the received names (written as $\{^{na\tilde{m}e}/_{\tilde{z}}\}$). Matching input and output prefixes might communicate, those leading to an interaction. 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 equals y. We utilize upper case letters for process identifiers and lower case letters for names. The abbreviation $\sum_1^m(M)$ is used to denote the summation of m choices, $\prod_1^m(P)$ denotes the composition of m parallel copies of P, and $\{\pi\}_1^m$ denotes m subsequent executions of π. Furthermore defined processes from the original paper on the π-calculus are used for parametric recursion, that is $A(y_1, ..., y_n)$ [24]. An introduction can be found for instance in [25] or [26].

4.2 Interactions in the Pi-Calculus

In the pattern representations each interaction participant is modeled as a π-calculus process. In the case of bilateral interactions we named them A and B, in the case of multi-lateral interactions A, B_i and P where $i = 1, 2, \cdots$. Since timers and exception handling are explicitly called for in the patterns we introduce an environmental process $\mathcal{E}_X$ per interaction participant ($X = A, B, B_i, P$). It is left open how timeouts and exception handling are implemented. $\overline{settimer_{\mathcal{E}_X}}\langle timer\rangle$

is supposed to set a new timer where a timeout is thrown by sending on channel $timer$. Exceptions can be thrown by sending on channel $\overline{fault_{\mathcal{E}_X}}$.

In the π-calculus a message represented by a name is sent and received at the same time, resulting in an interaction. I.e. if a process wants to send a message then it blocks until a receiver actually receives the message. Therefore, the π-calculus assumes synchronous communication as well as reliable and guaranteed delivery as the default case. If we want to model that a message sent from A to B can get lost or that the delivery can be delayed we might introduce a medium M in the following way:

$$\begin{aligned} A &= \overline{b_1}\,\langle msg\rangle\,.\mathbf{0} \\ M &= b_1(msg).((\overline{b_2}\,\langle msg\rangle\,.\mathbf{0} + \tau_0.\mathbf{0}) \mid M) \\ B &= b_2(msg).\mathbf{0} \end{aligned}$$

The medium decides non-deterministically if the message is either discarded or forwarded to B. Due to the sequence within M, sending msg in A and receiving msg in B do not happen at the same point in time, which models a potential delay as well as asynchronous communication. Another property of the medium concerning the order of delivery is that first in first out(FIFO) is not guaranteed any longer. Introducing such a medium allows for reasoning if a process still meets its requirements even in the case of delayed or unreliable message delivery. However, we assume that the underlying infrastructure (represented by the environmental processes $\mathcal{E}$) guarantees reliable FIFO delivery. In the cases where this cannot be met we assume that an exception $\overline{fault_{\mathcal{E}}}$ is raised for the whole composition.

The following subsections present formalizations for every service interaction pattern. As proposed in [26] we omit the termination symbol $\mathbf{0}$ in process definitions for simplicity. The pattern descriptions at the beginning of each subsection are directly taken from [4].

4.3 Single-transmission Bilateral Interaction Patterns

Send: A party sends a message to another party. The pattern definition distinguishes between blocking send and non-blocking send. In the case of blocking send the sending process cannot proceed until it can be sure that the message has been received. As already mentioned above this blocking behavior is inherent to π-calculus. Blocking send is given by:

$$\begin{aligned} A &= \overline{b}\,\langle msg\rangle\,.A' \\ B &= b(msg).B' \end{aligned}$$

This pattern formalization leaves it open if the receiver of the message is known at design-time or not. If we define the system as

$$S = (\mathbf{v}\ b)(A \mid B)$$

then A knows the link to B at design-time. If we define it as

$$S = (\mathbf{v}\ lookup)(lookup(b).A \mid (\mathbf{v}\ b)(B \mid D))$$

then A would get the link to B at run-time. In this case D could be something like a UDDI directory where we can lookup the receiver. A' and B' represent the so called continuations mentioned in the pattern descriptions. We continue with non-blocking send:

$$A = \overline{b}\,\langle msg\rangle \mid A'$$
$$B = b(msg).B'$$

Strictly speaking we could omit the formalization for B. However, for illustration purposes we provide one possible implementation for B to have a valid choreography. Most interaction patterns describe the interactions from the perspective of one single participant. In order to get a minimal choreography we have to plug several patterns together (e.g. send for A and receive for B).

Receive: A party receives a message from another party. This pattern is given by:

$$A = a(msg).A'$$
$$B = \overline{a}\,\langle msg\rangle .B'$$

Send/Receive: A party A engages in two causally related interactions. In the first interaction A sends a message to another party B (the request), while in the second one A receives a message from B (the response). In order to keep track of the relation between the two interactions we have to introduce some kind of correlation mechanism. In π-calculus we can create an infinite number of new names for a process that are not known to any other process. We create a new name for every set of correlated interactions. If we then use this new channel for communication we can be sure that any message that is sent over this channel is correlated to the other interactions. In the following example A creates a new name a used for receiving the response. A blocks until the corresponding message has been received. Blocking send/receive is given by:

$$A = (\mathbf{v}\ a)\overline{b}\,\langle a, req\rangle .a(resp).A'$$
$$B = b(a, req).\tau_B.\overline{a}\,\langle resp\rangle .B'$$

As already mentioned above the formalization of B is only one valid example. E.g. we could imagine interactions with other processes between receiving the request from A and sending the response. Non-blocking send/receive:

$$A = (\mathbf{v}\ a, h)(A_1|A_2)$$
$$A_1 = \overline{b}\,\langle a, req\rangle .(\overline{h} \mid A_1')$$
$$A_2 = h.a(resp).A_2'$$
$$B = b(a, req).\tau_B.\overline{a}\,\langle resp\rangle .B'$$

The new name h has to be introduced since the pattern descriptions demands that the request has been sent before a response can be received.

4.4 Single-transmission Multilateral Interaction Patterns

Racing incoming messages: A party expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes. Normally names are not typed in π-calculus. In order to retrieve the type of a message we could explicitly receive a second name representing the type. We opted for a more elegant way: for each type a channel is created and thus the channel a message is sent over determines the message's type. In the following formalization we assume that there are two different types of messages. Each B_i can send a message over channel a_1 if it is of the first type or over channel a_2 for the second type. Depending on the type of the message the continuation for A is either A'_1 or A'_2. The pattern distinguishes between discarding remaining messages and keeping them for further interactions. If remaining messages are not discarded, the patterns is defined by:

$$\begin{aligned} A &= (a_1(msg).A'_1 + a_2(msg).A'_2) \\ B_i &= (\overline{a_1}\,\langle msg\rangle\,.B'_i + \overline{a_2}\,\langle msg\rangle\,.B'_i) \end{aligned}$$

Once again the formalization for B_i is just an example. In this case every B_i can sent messages of every type. If we want to model that the continuation of A depends on the category of the sender we could define $B_i = \overline{a_1}\,\langle msg\rangle\,.B'_i$ and introduce another category $C_i = \overline{a_2}\,\langle msg\rangle\,.C'_i$. A generic formalization for an arbitrary number of different types/categories would be $A = \sum_{i=1}^{n} a_i(msg).A'_i$

In the following formalization all remaining messages are received but not taken care of. The number of messages is not known at run-time:

$$\begin{aligned} A &= a_1(msg).(A'_1 \mid A_{discard}) + a_2(msg).(A'_2 \mid A_{discard}) \\ A_{discard} &= !a_1(msg) \mid !a_2(msg) \\ B_i &= (\overline{a_1}\,\langle msg\rangle\,.B'_i + \overline{a_2}\,\langle msg\rangle\,.B'_i) \end{aligned}$$

In order to be able to receive an arbitrary number of messages we have to use the ! operator which stands for an infinite number of processes that run in parallel. Therefore, A never terminates. It can be seen as a drawback of π-calculus that discarding an unknown number of messages cannot be modeled in a different way.

One-to-many send: A party sends messages to several parties. The messages all have the same type (although their contents may be different). If the number of recipients is known at design-time:

$$\begin{aligned} A &= (\mathbf{v}\ h)(\prod_{i=1}^{n} \overline{b_i}\,\langle msg_{B_i}\rangle\,.\overline{h} \mid \{h\}_1^n.A') \\ B_i &= b_i(msg).B'_i \end{aligned}$$

If the number of recipients is known at run-time:

$$\begin{aligned} A =& (\mathbf{v}\ h)(A_1(h) \mid h.A') \\ A_1(h) =& hasnext(hn). \\ & ([hn = true]next(b).(\mathbf{v}\ i)(\overline{b}\langle msg\rangle.i.\overline{h} \mid A_1(i)) \\ & + [hn = false]\overline{h}) \\ B_i =& b_i(msg).B_i' \end{aligned}$$

For this pattern we have introduced the concept of an iterator that iterates over a set of recipients. The process *hasnext* returns either *true* or $false$ to tells us if there are more elements. *next* returns the next recipient. An iterator could be defined using data structures such as a stack (see e.g. [25]).

One-from-many receive: A party receives a number of logically related messages that arise from autonomous events occurring at different parties. The arrival of messages needs to be timely so that they can be correlated as a single logical request. The interaction may complete successfully or not depending on the set of messages gathered. The pattern introduces the notion of a stop condition and a success condition. In the simplest flavor of the pattern the interaction succeeds if n messages have been received. In this case we stop the interaction as soon as this success condition is met or a timeout has occurred:

$$\begin{aligned} A =& (\mathbf{v}\ timer, kill)(\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.\{a(msg)\}_1^n.\overline{exec} \\ & \mid (exec.\overline{kill}.A' + kill) \mid (timer.\overline{kill}.\overline{fault_{\mathcal{E}_A}} + kill)) \\ B_i =& \overline{a}\langle msg\rangle.B_i' \end{aligned}$$

In order to model arbitrary stop and success conditions we can use non-deterministic choices:

$$\begin{aligned} A =& (\mathbf{v}\ exec, timer)(\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.(!a(msg).(\underbrace{\overline{exec} + \tau_0}_{stop\ condition})) \\ & \mid timer.\overline{exec} \mid exec.(\underbrace{A' + \overline{fault_{\mathcal{E}_A}}}_{success\ condition})) \\ B_i =& \overline{a}\langle msg\rangle.B_i' \end{aligned}$$

An infinite number of messages can be received over channel a. Each time a message arrives we check if the stop condition is already met. If this is the case then the process sends over channel $\overline{exec}$. Now it is checked if the success condition is met. The pattern description defines that the success of the interaction depends on the set of messages. This is not explicitly modeled in the formalization.

One-to-many send/receive: A party sends a request to several other parties, which may all be identical or logically related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe and

some parties may even not respond at all. The interaction may complete successfully or not depending on the set of responses gathered. If the recipients are known at design-time and we assume arbitrary stop and success conditions the formalization of one-to-many send/receive looks as follows:

$$\begin{aligned} A =&(\mathbf{v}\ timer, exec)(\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.\prod_{i=1}^{n} A_i \\ &\mid timer.\overline{exec} \mid exec.(A' + \overline{fault_{\mathcal{E}_A}})) \\ A_i =&(\mathbf{v}\ a)(\overline{b_i}\,\langle a, req\rangle\,.a(resp).(\overline{exec} + \tau_0)) \\ B_i =&b_i(a, req).\tau_{B_i}.\overline{a}\,\langle resp\rangle\,.B_i' \end{aligned}$$

If the recipients are known at run-time we introduce an iterator once again:

$$\begin{aligned} A =&(\mathbf{v}\ timer)(\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.A_1 \mid timer.\overline{exec} \mid exec.(A' + \overline{fault_{\mathcal{E}_A}})) \\ A_1 =&hasnext(hn).([hn = true]next(b).(A_1 \mid A_2(b)) + [hn = false]\tau_0) \\ A_2(b) =&(\mathbf{v}\ a)(\overline{b}\,\langle a, req\rangle\,.a(resp).(\overline{exec} + \tau_0)) \\ B_i =&b_i(a, req).\tau_{B_i}.\overline{a}\,\langle resp\rangle\,.B_i' \end{aligned}$$

4.5 Multi-transmission Interaction Patterns

Multi-responses: A party A sends a request to another party B. Subsequently, A receives any number of responses from B until no further responses are required. The trigger of no further responses can arise from a temporal condition or message content, and can arise from either A or B's side. In the following formalization we do not explicitly model the temporal condition mentioned in the pattern description:

$$\begin{aligned} A &= (\mathbf{v}\ a)\overline{b}\,\langle a, req\rangle\,.A_{receive} \\ A_{receive} &= a(resp).\tau_A.(A_{receive} + A' + \overline{b}\langle stop\rangle.A') \\ B &= b(a, req).B_{send} \\ B_{send} &= \tau_B.(\overline{a}\,\langle resp\rangle\,.(B_{send} + B') + b(stop).B') \end{aligned}$$

In $A_{receive}$ we see a non-deterministic choice with three alternatives. The first alternative can be taken if another message arrives from B. The second alternative is chosen if the message content of the last message indicated the termination of the interaction. The third option models the possibility that A decides to stop the interaction. These three options also appear in B_{send}. Either another message is sent or B decides to terminate the interaction or a stop-message is received from A.

Contingent requests: A party A makes a request to another party B1. If A does not receive a response within a certain timeframe, A alternatively sends a request

to another party B2, and so on. A retrieves the list of invocation targets at runtime. For that purpose we use an iterator once again. An import design issue of this pattern is whether or not responses should be considered after the timeout has already occurred. If we do not consider responses from services we invoked earlier:

$$
\begin{aligned}
A =& hasnext(hn).\\
&([hn = true]\, next(b).(\mathbf{v}\ a)\overline{b}\,\langle a, req\rangle\,.(\mathbf{v}\ timer)\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.\\
&(a(resp).A' + timer.A)\\
&+ [hn = false]\,\overline{fault_{\mathcal{E}_A}})\\
B_i =& b_i(a, req).\tau_{B_i}.\overline{a}\,\langle resp\rangle\,.B_i'
\end{aligned}
$$

When considering responses from services invoked earlier, the pattern is as follows:

$$
\begin{aligned}
A =& (\mathbf{v}\ exec)(hasnext(hn).\\
&([hn = true]\, next(b).(\mathbf{v}\ a)\overline{b}\,\langle a, req\rangle\,.(\mathbf{v}\ timer)\overline{settimer_{\mathcal{E}_A}}\langle timer\rangle.\\
&(a(resp).\overline{exec} + timer.(A \mid (a(resp).\overline{exec} + h)) + h)\\
&+ [hn = false]\,\overline{fault_{\mathcal{E}_A}})\\
&\mid exec.(A' \mid\, !\overline{h}))\\
B_i =& b_i(a, req).\tau_{B_i}.\overline{a}\,\langle resp\rangle\,.B_i'
\end{aligned}
$$

Atomic multicast notification: A party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain timeframe. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number. We introduce the two names *accept* and $reject$. If A receives a $reject$-message, the multicast notification has failed and the continuation A_2' is taken. If m *accept*-messages are received before a $reject$-message arrives the multicast notification has succeeded and the continuation A_1' is taken:

$$
\begin{aligned}
A =& (\mathbf{v}\ a, h, exec)((\prod_{i=1}^{n} \overline{b_i}\,\langle a, notification\rangle\,.a(resp).\\
&([resp = accept]\overline{h} + [resp = reject]\overline{exec}\,\langle reject\rangle))\\
&\mid \{h\}_1^m.\overline{exec}\,\langle accept\rangle\\
&\mid exec(resp).(([resp = accept]A_1' + [resp = reject]A_2') \mid \prod_{i=1}^{n} \overline{b_i}\,\langle resp\rangle))\\
B_i =& b_i(a, notification).\tau_{B_i}.\\
&(\overline{a}\,\langle accept\rangle\,.b_i(resp).([resp = accept]B_{i1}' + [resp = reject]B_{i2}') +\\
&\overline{a}\,\langle reject\rangle\,.B_{i2}' + b_i(resp).B_{i2}')
\end{aligned}
$$

4.6 Routing Patterns

Request with referral: Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties $(P1, P2, \cdots, Pn)$ *depending on the evaluation of certain conditions. While faults are sent by default to these parties, they could alternatively be sent to another nominated party (which may be party A).* While the pattern descriptions talks about a number of parties P_i, the following formalization only presents the case of one party P for better readability:

$$\begin{aligned} A &= (\mathbf{v}\ a)\overline{b}\,\langle a, p, req\rangle\,.a(resp).A' \\ B &= (\mathbf{v}\ msg)b(a, x, req).\tau_B.\overline{x}\,\langle a, msg\rangle\,.B' \\ P &= p(a, msg).\tau_P.\overline{a}\,\langle resp\rangle\,.P' \end{aligned}$$

If we want the follow-up responses to be sent to several P_i we could incorporate the pattern one-to-many send into B. The following formalization models that all faults are sent to A:

$$\begin{aligned} A &= (\mathbf{v}\ a, toa)\overline{b}\,\langle a, p, toa, req\rangle\,.(a(resp).A' + toa.\overline{fault_{\mathcal{E}_A}}) \\ B &= b(a, x, faultto, req).\tau_B.(\overline{x}\,\langle a, faultto, msg\rangle\,.B' + overline faultto.\overline{fault_{\mathcal{E}_B}}) \\ P &= p(a, faultto, msg).\tau_P.(\overline{a}\,\langle resp\rangle\,.P' + (\overline{faultto} \mid \overline{fault_{\mathcal{E}_P}})) \end{aligned}$$

In this pattern it becomes obvious that every participant needs his own exception handling mechanism which is implemented in the corresponding environmental processes $\mathcal{E}_A$, $\mathcal{E}_B$ and $\mathcal{E}_P$.

Relayed request: Party A makes a request to party B which delegates the request to other parties $(P1, \cdots, Pn)$*. Parties* $P1, \cdots, Pn$ *then continue interactions with party A while party B observes a* view *of the interactions including faults. The interacting parties are aware of this* view *(as part of the condition to interact).* Like we already did it for the last pattern we only model one party P:

$$\begin{aligned} A &= (\mathbf{v}\ a)\overline{b}\,\langle a, req\rangle\,.a(resp).A' \\ B &= b(a, req).\overline{p}\,\langle a, b, req\rangle\,.b(resp).B' \\ P &= p(a, b, req).\tau_P.(\mathbf{v}\ h)(\overline{a}\,\langle resp\rangle\,.\overline{h} \mid \overline{b}\,\langle resp\rangle\,.\overline{h} \mid h.h.P') \end{aligned}$$

Dynamic routing: A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the intermediate steps. Since the pattern description covers a broad range of possible interactions we will only focus on two aspects. The first aspect is routing the request to a third party based on the content of the

message. If there are n possible recipients C_1 to C_n known at design-time the formalization looks as follows.

$$A = \overline{b}\,\langle req_1 \rangle\,.A'$$
$$B = b(req_1).\tau_B.\sum_{i=1}^{n} \overline{c_i}\,\langle req_2 \rangle\,.B_i'$$
$$C_i = c_i(req_2).C_i'$$

The second aspect that is mentioned in the detailed pattern description covers routing a document from one participant to another where every participant has read-only access to the document the whole time but has to retrieve a write-token when wanting to modify the document. The following formalization shows how such data structures can be expressed in π-calculus:

$$\begin{aligned} Gen_D &= !(\mathbf{v}\ read, lock, x)\overline{create}\langle read, lock\rangle.D_{unlocked}(x) \\ D_{unlocked}(x) &= \overline{read}\langle x\rangle.D_{unlocked}(x) \\ &\quad + (\mathbf{v}\ write, ul)\overline{lock}\langle write, ul\rangle.D_{locked}(x, write, ul) \\ D_{locked}(x, write, ul) &= \overline{read}\langle x\rangle.D_{locked}(x, write, ul) \\ &\quad + write(y).D_{locked}(y, write, ul) \\ &\quad + ul.D_{unlocked}(x) \end{aligned}$$

Gen_D is a generator that can create an infinite number of memory cells. Every participant wanting access to this memory cell needs the *read* and *lock* channels. As soon as a participant locks the cell, this participant gets the *write* channel that he can use in order to modify the contents. He has to unlock the cell before anyone else can retrieve a *write* channels. Every time the cell is locked, new names for the *write* and *unlock* channels are created. That way it is ensured that only the participant who has currently locked the cell can write on it and can unlock it. Let us now assume a scenario where a participant A sends a document to a set of recipients $B_1 \cdots B_n$. In an inter-leaved parallel routing manner every participant is asked to modify the document.

$$\begin{aligned} A =& (\mathbf{v}\ h)(\prod_{i=1}^{n} \overline{b_i}\langle read, lock\rangle.\overline{h} \mid \{h\}_1^n.A') \\ B_i =& b_i(read, lock).lock(write, unlock).read(doc).\tau_{B_i}.\overline{write}\langle doc\rangle.\overline{unlock}.B' \end{aligned}$$

5 Conclusion and Outlook

In this paper we have shown how the service interaction patterns can be formalized. We investigated *traditional* approaches using different types of Petri nets as well as a new direction based on mobile process algebra represented by the π-calculus. We investigated and discussed short comings of Petri nets and tried to solve them using high level extensions. However, as Petri nets do not support the concept of mobility, required for dynamic routing in service interaction

patterns, they fail at representing even simple patterns in practice. Nevertheless, theoretically all service interaction patterns could be modeled using Petri nets by constructing infinite nets as ten out of thirteen patterns incorporate sending messages, where the receiver might not be known at design-time. To overcome the limitations of Petri nets, we investigated mobility as represented by the π-calculus to formalize the patterns. We were able to express all service interaction patterns in π-calculus processes. Therefore, our final conclusion is that π-calculus is better suited for expressing the service interaction patterns (see [27] for a definition of suitability).

The formalizations presented in this paper can be the starting point for further work on a complete formal grounding of the intersection of the domains service oriented architectures and business process management using π-calculus. The very next step would be to show how the formalizations of the service interaction patterns can be integrated with the formalizations of the workflow patterns provided in [14]. Once we have both a choreography and corresponding orchestrations available as π-calculus processes we can proceed with introducing conformance checking, e.g. verifying if the behavior of individual orchestrations complies to the choreography. Another area of interest is the investigation of soundness criteria for choreographies. A first starting point could be the soundness criteria for workflows [17].

References

1. IBM: Web services architecture overview. (2000) `http://www-128.ibm.com/developerworks/webservices/library/w-ovr/`.
2. van der Aalst, W.M.P., ter Hofstede, A.H., Weske, M.: Business Process Management: A Survey. In van der Aalst, W.M.P., ter Hofstede, A.H., Weske, M., eds.: Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS, Berlin, Springer-Verlag (2003) 1–12
3. Barros, A., Dumas, M., Oaks, P.: A critical overview of the web services choreography description language (ws-cdl). BPTrends Newsletter **3(3)** (2005)
4. Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management. Volume 3649. (2005) 302–318
5. 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. (2003) `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`.
6. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing web service choreographies. In: Proceedings of First International Workshop on Web Services and Formal Methods, Elsevier (2004)
7. Gorrieri, R., Guidi, C., Lucchi, R.: Reasoning about interaction patterns in choreography. In: M. Bravetti et al. (Eds.): Second International Workshop on Web Services and Formal Methods, LNCS 3670, Springer Verlag (2005) 333348
8. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration: A synergic approach for system design. In: B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, Springer Verlag (2005) 228–240

9. Martens, A.: Analyzing web service based business processes. In Cerioli, M., ed.: Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05), Part of the 2005 European Joint Conferences on Theory and Practice of Software (ETAPS'05). Volume 3442 of Lecture Notes in Computer Science., Springer-Verlag (2005)
10. van der Aalst, W.M.P., Weske, M.: The P2P approach to Interorganizational Workflow. In Dittrich, K., Geppert, A., Norrie, M., eds.: Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), volume 2068 of LNCS, Berlin, Springer-Verlag (2001) 140–156
11. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14(3)** (2003) 5–51
12. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language (Revised version. Technical Report FIT-TR-2003-04, Queensland University of Technology, Brisbane (2003)
13. Kavantzas, N., et al.: Web service choreography description language (ws-cdl). Technical report (2004)
14. Puhlmann, F., Weske, M.: Using the π-calculus for formalizing workflow patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management. Volume 3649. (2005) 153–168
15. Overdick, H., Puhlmann, F., Weske, M.: Towards a formal model for agile service discovery and integration. In: Proceedings of the ICSOC Workshop on Dynamic Web Processes (DWP 2005), Amsterdam, The Netherlands (2005)
16. van der Aalst, W., van Hee, K.: Workflow Management. MIT Press (2002)
17. van Hee, K., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: W. van der Aalst, E. Best (Eds.): Applications and Theory of Petri Nets 2003, LNCS 2679, Springer Verlag (2003) 337–356
18. Dehnert, J., Rittgen, P.: Relaxed soundness of business processes. In Dittrich, K., Geppert, A., Norrie, M.C., eds.: CAiSE 2001, volume 2068 of LNCS, Berlin, Springer-Verlag (2001) 157–170
19. van der Aalst, W.M.P.: Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. Information System Frontiers **3** (2001) 297–317
20. Aalst, W.v.d.: Inheritance of Workflow Processes: Four problems - One Solution? In Cummins, F., ed.: Proceedings of the Second OOPSLA Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems, Denver, Colorado (1999.) 1–22
21. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Institut für Instrumentelle Mathematik, Bonn (1962)
22. Jensen, K.: Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1. Springer-Verlag, London, UK (1996)
23. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2003)
24. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes - Part I. (1989)
25. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press, Cambridge (1989)
26. Parrow, J.: An Introduction to the πCalculus. Elsevier (2001)
27. Kiepuszewski, B.: Expressiveness and suitability of languages for control flow modelling in workflows. (2002)