Seminar Reader

Business Process Management
Winter Term 2006/2007

Editors

Gero Decker
Hagen Overdick
Frank Puhlmann
Hilmar Schuschel
Mathias Weske

Contact
Business Process Technology Group
Hasso Plattner Institute for IT Systems Engineering
at the University of Potsdam
P.O. Box 90 04 60
D-14440 Potsdam, Germany

Preface

This seminar reader contains selected papers of the seminar part of the lecture Business Process Management II, held in winter term 2006/2007 at the Hasso-Plattner-Institute. Master students of IT systems engineering discussed and researched current topics in the area of business process management. Each paper contained in this seminar reader was accompanied by a conference-style talk.

We had a broad range of topics, ranging from theoretical foundations over semantic web services up to investigations on practical implementations. All topics have been grounded in lectures given beforehand, so the students had a very broad foundation and talks and papers dived right into challenging questions. The topics are divided into three major parts, resembling the parts of the lectures given beforehand. Part one introduced semantic services, were a paper about service interface adaptation is contained in this reader. Part two covered service-oriented architectures with a special focus on choreographies. Three papers based thereon discuss multi-lateral collaborations and the bridging of global and local models. Part three investigated formal aspects of service-oriented architectures using the pi-calculus. This reader contains papers about a new compatibility notion for services as well as implementation issues for a pi-calculus-based process engine.

We would like to thank the students that attended our lecture and contributed to this seminar reader – it was a very interesting and inspiring experience!

Gero Decker, Hagen Overdick, Frank Puhlmann, Hilmar Schuschel, Mathias Weske
Potsdam, February 2007

Table of Contents

Service Interface Adaptation
Enabling Automation with Semantics

Uwe Kylau

Hasso-Plattner-Institute for IT-Systems Engineering at the University of Potsdam,
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
uwe.kylau@hpi.uni-potsdam.de

Abstract. The vision of a service marketplace is to establish business relations between service consumers and providers in a Service-oriented Architecture (SOA). However, in more complex scenarios there is often a mismatch between business protocols, i.e. the course of interaction that is taken to deliver certain functionality. In order to close this gap and address the widest range of potential consumers, a provider must adapt its service interfaces to consumer demands.
This document is based on previous work in the area of service interface adaptation. It presents a particular approach, which is augmented with semantics to enable a certain degree of automation. At the end it also gives reasons, why a full automation is hard to realize.

1 Introduction

Web service marketplaces are today's most promising application scenarios for a Web/XML-based Service-oriented Architecture (see e.g. [3]). Taken into account that an SOA always has real world effects (according to [4]), one can draw a simple conclusion. These marketplaces are controlled by the same economic principles as those observed in their real world counterpart. Thus, adapting to consumer expectations is a considerable requirement for the development of service marketplaces.

The term adaptation can generally be subsumed with: provide a flexible business model. In order to deliver certain capabilities, a provider should offer various possibilities to interact with the consumer, and if necessary, should be able to quickly establish new ways of interaction. In most cases these interactions are bi-directional and non-trivial, i.e. consist of several steps that are connected with complex control flow. Hence, a process-like character is often recognizable. In [4] the interaction with a service is described in a similar fashion. An informational model deals with structure and semantics of exchanged data. Next to that a service also needs a behavioral model that gives information about which actions (steps) to perform. Moreover, the "[..] temporal relationships and temporal properties of actions and events [..]"[1] must be known, which are characterized

[1] [4], p.18

by the process. Both models are mostly referred to as structural and behavioral interface of a service. Together they form the overall service interface.

Structural interfaces can be expressed with various Web Services specification technologies and represented in various notations. Common nowadays is to supply a Web Service description as a WSDL document ([5]) written in XML ([9]). Nonetheless, semantically enriched WSDL like SAWSDL ([10]) and complete upper ontologies like OWL-S ([6]) or WSMO ([12]) are gaining more and more importance. The latter combine structural and behavioral interface on a higher level of description. The low-level standard for behavioral interfaces is BPEL ([8]).

Almost all major Web Services specifications have been implemented and incorporated into integrated tool suites. Accordingly, from the technical side there should be no obstacles for the development of a sophisticated service marketplace. Unfortunately, the problems arise when analyzing the domain in detail. Especially, the interaction with the consumer is of interest at this point. Consumers would like to have flexibility on all aspects of the interaction. They want to decide when, where and how to interact. All this is no real problem in a SOA. Services are available most of the time and from every place that has access to the Internet. Furthermore, Web Services technologies are integrated on all well-established platforms, which run on a larger number of devices. But there is one aspect that imposes difficulties. Consumers also want to have flexibility on what is exchanged during the interaction and in which order. This ranges from *'let us only exchange what is necessary'* to *'give me all the information you can provide'*, even to *'I want to process each item separately, instead of all at once'*.

The degree of flexible interaction with a service depends on the flexibility that is defined (or better designed) in the service interface. The service capabilities are fixed, as well as structure and behavior. In a SOA this fact holds for both, the provider and the consumer. It is unreasonable to demand from the consumer that he waits with implementing his side of the interaction protocol until the providing service is determined. On the other hand, a provider cannot anticipate all possible ways of interacting with its service. The usual case will be that both sides have their relatively inflexible interface, one side demanding certain functionality and the other one providing it (see Fig. 1). A mismatch between those interfaces is likely to occur. The classic market rules now expect the provider to adapt his interface in such a way, so that it behaves like the interface required by the consumer. It is a key requirement for a service marketplace to be able to perform this service interface adaptation.

Adapting a published service interface to an interface that is requested by a service consumer is a non-trivial task. Both, messages and behavior (control flow), must be mediated by an adaptor. Until now, especially the problem of adapting behavioral interfaces has received little attention in the area of service-oriented computing. One approach that is presented in [1] deals with it in a suboptimal way. The adaptation between two interfaces is defined manually and the adaptor is then generated from the result of this definition. The approach in this paper intends to overcome the latency of manual work and describes a way

to automate the adaptation definition by extending the existing approach with semantics.

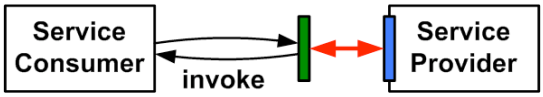

Fig. 1. The diagram depicts the conceptual interface mismatch between service consumer and provider. The expected service interface is shown green and the provided interface blue.

The rest of this document is structured as followed. Section 2 presents an algebra and visual notation for service interface adaptation that maps (transforms) behavioral interfaces. In Sect. 3 the algebra and visual notation is enhanced with semantic annotations and an algorithm is introduced that allows for automation of the interface mapping procedure. Afterwards Sect. 4 gives information about related work in the area of service interface adaptation and Sect. 5 concludes the paper.

2 Algebra and Visual Notation for Service Interface Adaptation

In the following, an approach is outlined that deals with the problem of service interface adaptation. Due to space limitations only the main points of the approach are introduced. Please refer to [1] for a detailed description.

The approach defines an interface transformation algebra founded on a simple service interface model. Altogether, the algebra consists of six operators to formulate interface transformation expressions. These, in turn, provide the input for the generation of adaptors. A prototypical implementation of the approach was also produced. It simulates message exchange between two parties that is mediated by adaptors.

For the purpose of better visualizing an interface mapping, the approach uses a visual notation that is based on UML State Charts ([11]). Figure 2 shows an example of this notation. The example will be used throughout the rest of the document and shall be explained here. Two behavioral service interfaces are depicted. The state chart at the top models the interaction protocol of the provided service. On reception of a *purchase order* the service is triggered. It then sends back an *order response* wherein all or a subset of the requested line items are accepted or rejected. Outstanding line items are processed further. When their status is available, it is communicated in one or more *order updates*. Then *payment and shipment information* is sent to the requestor and after receiving a *notification* of the payment, the successful payment is synchronized with the requestor.

On the opposite side another state chart models the interaction protocol that the service consumer understands and expects. This is the required interface. As can be seen, it also starts with the reception of a *purchase order*. The next step requires the provider to send back an *order response* that contains acceptance information on all the requested line items. When this is done the consumer is prepared to receive *payment information* and *shipment information* separately and concurrently. Thereafter he or she sends a *notification* as soon as the payment is performed, whereby the interaction is considered to be finished.

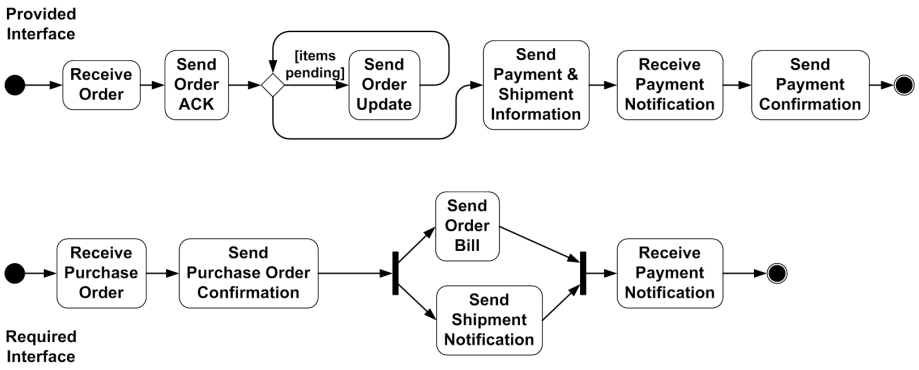

Fig. 2. This diagram shows an example of a mismatch between provided (top) and required interface (bottom).

It is obvious that the two interfaces do not match. The next sections will describe how the interface adaptation by transformation is accomplished, but beforehand it must be clearly pointed out that there is one limitation. The approach only works in scenarios where the required interface encompasses the same, or less, functionality (service capabilities) as offered by the provided interface. Otherwise, the adaptor would have to supply the missing functionality, which would result in a dispersal of business logic. However, this does not rule out cases where incoming data of the required interface is completely irrelevant to the remaining course of interaction and hence can be discarded.

2.1 Service Interface Model

The model of the service interface contains relatively few concepts. An interface I consists of a set of actions and a set of runs that prescribe an order on the execution of actions (control flow). Loops are a special case in the interface model. They are limited to consecutive executions of a single action.

Definition 1. *An action A is tuple A = (AN, MT, D) where AN is the name of the action (action identifier), MT is the message type involved during a send or receive and D is the direction of the action. A direction value of IN is used for receiving actions, while OUT is used for sending actions.*

It is possible that not all actions of an interface are executed during an interaction, e.g. when there is conditional branching. A complete interaction sequence is defined as a trace through the interface. The overall set of traces describes the behavior of the interface. Traces are grouped into disjoint groups gt_1, gt_2, ..., where each group consists of all traces that contain the same set of action instances.

Definition 2. *A run r over an interface I for a given group of traces gt is a partial order $<_r$ such that:*

$$\forall a_1, a_2 \in Actions(gt)\ a_1 <_r a_2 \leftrightarrow (\forall t \in gt a_1 <_t a_2)$$

Actions(gt) denotes the common set of actions of the traces in gt and $<_t$ is the order relation within a trace t.

In Fig. 3 an example interface along with its runs is illustrated.

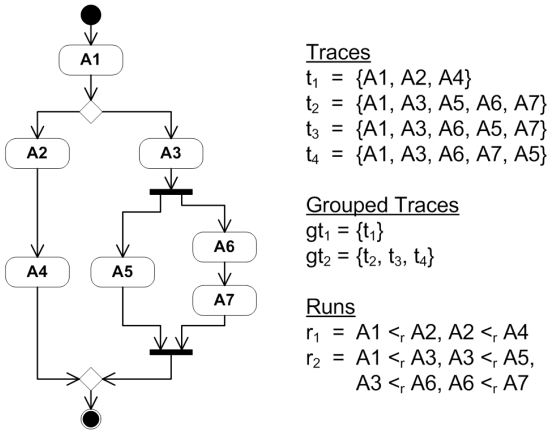

Fig. 3. The figure illustrates an abritrary service interface together with the corresponding traces, grouped traces and runs.

2.2 Interface Transformation Algebra and Visual Notation

In this section the operators of the algebra are introduced together with their visual notation. The operators are employed to transform a source interface into a target interface, which is not to be mistaken with provided and required interface. They are precisely defined in terms of input parameters and output result (see [1]). The following will only give a brief description that is necessary to understand the rest of the document.

Fig. 4 shows the visual notation of the six operators with *SI* being the source interface and *TI* being the target interface.

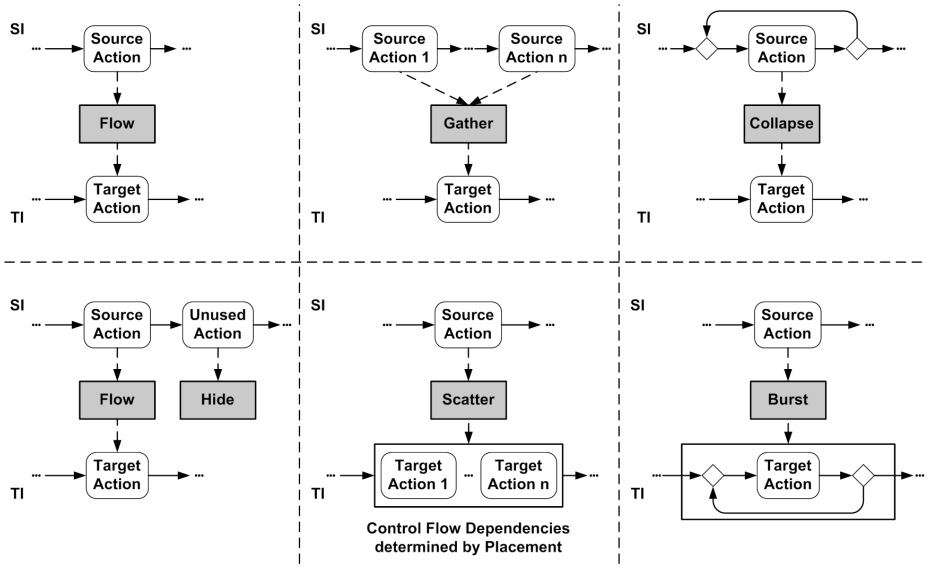

Fig. 4. These diagrams depict the visual notation for the operators of the algebra.

The *Flow* operator is the simplest one. It takes a source action and replaces it with a target action, i.e. in every run of *SI* each occurrence of the source action is replaced with an occurrence of the target action to form the runs of *TI*. The message of source message type is transformed into a message of the target message type via a data transformation function. The details of this procedure are not in the focus of the approach, but in [1] reference is made to XSLT ([7]) and work in the area of Web data transformation ([2]).

The *Hide* operator is used to drop an action from the source interface so it gets hidden in the target interface.

The *Gather* operator comprises a family of operators $Gather_n$ where n is the number of source actions. *Gather* takes several source actions as input and replaces them with a single target action, i.e. each occurrence of a respectively ordered set of actions in the runs of *SI* is replaced with a target action to form the runs of *TI*. It should be noted that the consecutive order of the source actions may solely be interrupted by actions that are transformed into target actions that occur before the designated gathering target action. The source messages are collectively transformed into a single target message.

The *Scatter* operator is the reversed counterpart of the *Gather* operator. It also comprises a family of operators $Scatter_n$ where n indicates the number of target actions that replace a single source action. The order of the target actions is determined by an input parameter *Placement*. There is no particular notation yet, but a proposal made use of nested control flow operators. A short example of a placement expression was given with: $SEQ(TA_2, PAR(TA_1, TA_3))$. $TA_1, ..., TA_3$ are actions of the target interface. SEQ and PAR denote sequential

and parallel control flow. The *Scatter* operator also defines a data transformation function that splits the source message into several target messages.

Collapse and *Burst* are the last two operators. *Collapse* is used to replace multiple occurrences of a source action with a single target action. The data transformation function converts a list of equally typed messages into a single message of the target message type. *Burst* works the other way round. It replaces a single source action with multiple occurrences of one target action and splits the source message into a list of target messages.

Applying the algebra to the problem domain of interface adaptation results in a mapping between a provided and required interface. The outcome of such a mapping is illustrated in Fig. 5. A number of interface transformation expressions are graphically represented. Their algebraic notation is given in the list below, with RI being the required and PI being the provided interface. The terms F_n denote an appropriate data transformation function.

1. $E_1 = Flow(RI,$ *Receive Purchase Order,* $F_1,$ *Receive Order*$)$
2. $E_2 = Collapse(PI,$ *Send Order Update,* $F_2, TempAction)$
3. $E_3 = Gather(E_2,$ *Send Order ACK,* $TempAction, F_3,$ *Send Purchase Order Confirmation*$)$
4. $E_4 = Scatter(PI,$ *Send Payment & Shipment Information,* $F_4, PAR($*Send Order Bill, Send Shipment Notification*$))$
5. $E_5 = Flow(RI,$ *Receive Payment Notification,* $F_5,$ *Receive Payment Notification*$)$
6. $E_6 = Hide(PI,$ *Send Payment Confirmation*$)$

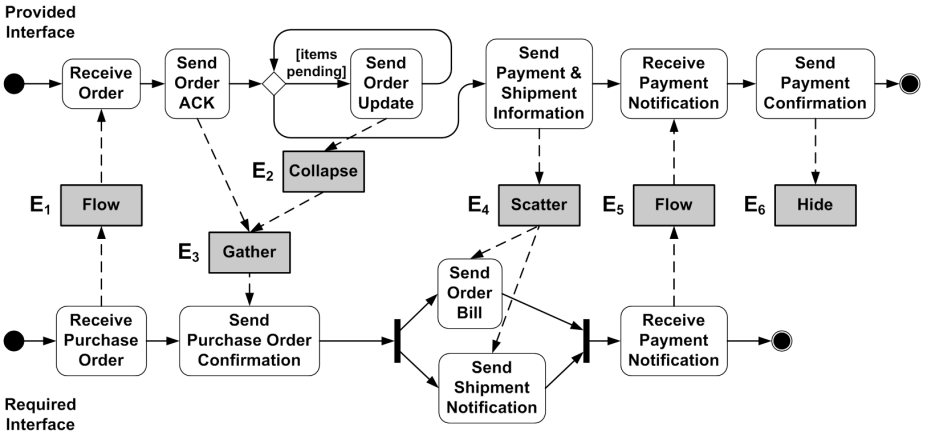

Fig. 5. This diagram shows an example transformation between provided (top) and required interface (bottom).

3 Automating the Interface Mapping by Applying Semantics

So far, the approach presented in the last section only allows for automated generation of adaptors. The interface mapping has to be done manually, before the resulting interface transformation expressions can be provided to a generator. However, this procedure is rather unsuitable for the application domain introduced in Sect. 1. A service consumer does not want to wait until a human worker has adapted the interfaces. The consumer wants instant access. Consequently, the interface mapping must be automated, too.

In this section a preliminary enhancement of the approach is outlined. It employs semantic service matchmaking to identify corresponding actions in the provided and required interface. Appropriate transformation operators can then be determined according to the runs of the interfaces. F-Logic ([13]) was chosen as notation for semantic expressions, because it is not too hard to learn and relatively easy to understand.

3.1 Assumptions for Enhancing the Approach with Semantics and the Changed Service Model

As the presented enhancement marks only a staring point for a comprehensive semantic approach, the following assumptions had to be made. First, the assumption of the original approach shall be mentioned again: The required interface must not introduce any further functionality. This is essential for the approach to work.

Additionally, it must now be guaranteed that each action contains sufficient semantic annotations. In particular, these are preconditions and effects associated with an action. Preconditions are used for *sending* actions, because they describe the conditions that have to hold before a message can be send. On the other hand, effects are used for *receiving* actions, as they formulate the conditions that hold after a message is received. The need for effects on *sends* and preconditions on *receives* did not became apparent until now. Since it cannot be totally ruled out, the enhanced action model was conceived for the general case.

Definition 3. *An action A is a tuple A = (AN, MT, D, P, E) where AN is the name of the action (action identifier), MT is the message type, D is the direction of the action, P is a set of preconditions and E is a set of effects. Both, P and E, must be given in F-Logic.*

It was stated above that semantic annotations have to be *sufficient*. The reason for this requirement can be found in the problem to clearly distinguish two actions from another. There is a conceptual equivalence between two actions, in cases where they share message type (MT) and direction (D). Thus, a computer could not decide which one is suitable to map a corresponding action in the source interface. This imposes difficulties on semantic matchmaking, unless there are *sufficient* semantic annotations that distinguish such actions.

For the purpose of simplicity and to make first correctness checks easy, interfaces with loops are not allowed. Loops are difficult to identify and matchmaking conditions that aim at *Burst* or *Collapse* operators is complicated to specify. Furthermore, both interfaces are required to use the same ontology as their type system. This avoids the additional step of mapping data types to ontological concepts.

An almost completely annotated example is provided in Fig. 7. It slightly differs from the example in the previous section, as there is only an option to send an order update once. There also had to be defined a special *NOP action* that does nothing, except identifying the conditions that hold in the branch where the action is located. This is necessary to semantically describe each branch that can be taken in a conditional branching.

Definition 4. *A NOP action N is an action, where N = (AN, MT, D, P, E), AN = "NOP", P is a set of preconditions, E is empty and MT, D are not set (**null**).*

3.2 The Automation Algorithm

The main principle of the automation algorithm is to map actions that need data to actions that provide data. E.g., *receiving* actions in the provided interface are mapped to *receiving* actions in the required interface, because during interaction incoming messages adhere to the definitions of the required interface and must be transformed into messages that are understood by the provided interface. For "sending" actions this principle works vice versa.

In order to find out which actions of the opposite interface are suitable to provide required data, semantic matchmaking is performed. The result is a mapping where target actions of potential operators are mapped to source actions. Each $1 : n$ relation in this initial mapping corresponds to a *Gather* operator. All $1 : 1$ relations must be reversed. Thus, source actions are now mapped to target actions. Each $1 : n$ relation in the reversed mapping corresponds to a *Scatter* operator. The remaining $1 : 1$ relations indicate *Flow* operators.

Automatic generation of data transformation functions is out of scope of this approach, but could be implemented according to [2]. The same applies to the semantic matchmaking algorithm. Moreover, the determination of placement for the *Scatter* operator is assumed to be given implicitly by the target interface of any such operator. A complete version of the algorithm in Java-related pseudocode is given in App. A.3.

During the first phase, the algorithm iterates over both interfaces. In the provided interface only *receiving* actions are interesting. Hence, for each *receiving* action, possible source actions are searched in the required interface. For this purpose, the effects of all *receives* in the required interface are matched against the effects of the currently selected *receive* in the provided interface. If a full or partial match is encountered, the action of the required interface is marked as "contributing" to the selected action. At the end of each iteration the union of effects of all "contributing" actions are checked for collisions and matched

against the effects of the selected action. If this check results in a full match, the selected action is mapped to the "contributing" actions.

After processing all *receives* in the provided interface, the second iteration does the same with the *sends* in the required interface. In Fig. 6 the beginning of the example interfaces is shown. The algorithm first selects *Send Purchase Order Confirmation* and tries to find possible source actions for this action. Starting with *Send Order ACK* the preconditions of all *sending* actions in the provided interface are checked. In this case, the preconditions of *Send Order ACK* partially match the preconditions of *Send Purchase Order Confirmation*. It is no full match, because `lessorequal(...)` does not conform to `equal(...)`. The remaining *sends* in the provided interface are processed and the result is a set of "contributing" actions with three entries: *Send Order ACK*, *Send Order Update* and *NOP*. *Send Purchase Order Confirmation* is mapped to these three actions. The second iteration continues until there are no more *sends* available in the required interface.

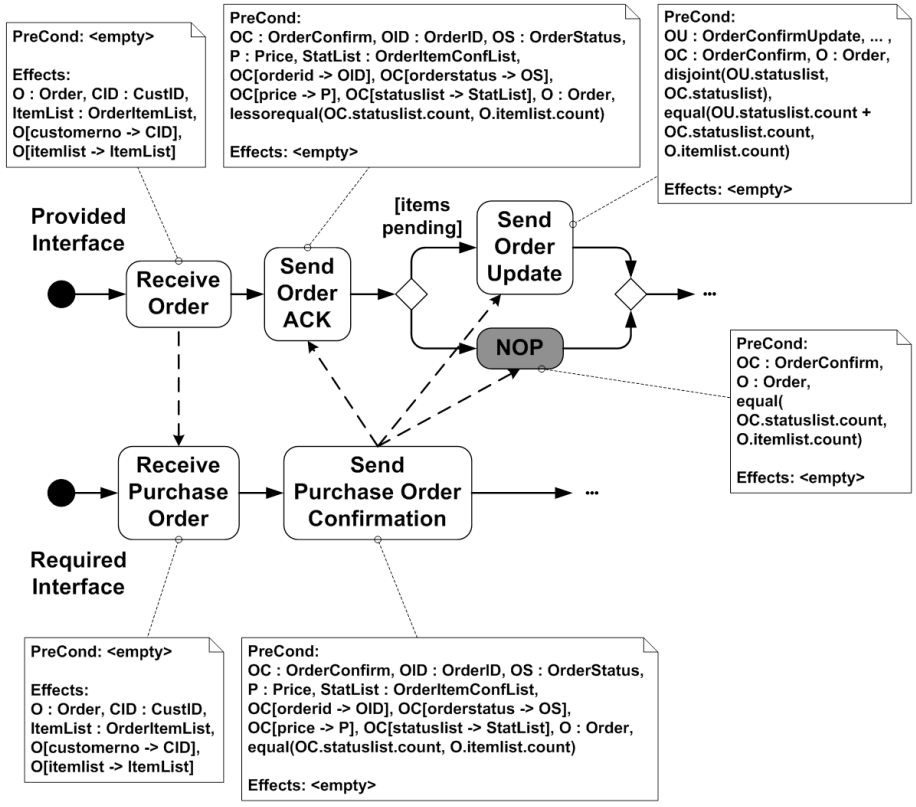

Fig. 6. The diagram illustrates an example mapping between provided (top) and required interface (bottom).

The next phase involves the selection of algebra operators as desrcibed above. In the example in Fig. 6 there is one $1:1$ mapping that is reversed and will result in a *Flow*, and one $1:n$ mapping that will result in one or more *Gather* operators. To achieve this, the algorithm checks, if all three associated actions are contained in each run of the provided interface. As the scenario involves conditional branching, this check will fail. Now, the algorithm iterates over all runs of the provided interface and determines for each run the subset of actions that are contained in the run. The found subsets are stored in a distinct list (no duplicates allowed). After the iteration has finished, a *Gather* operator is inserted for each stored subset (see Fig. 7). The number of *Scatter* operators is determined in the same way during processing of the reverse mapping.

At the end the algorithm will look for *sends* in the provided interface and *receives* in the required interface that are not included in the mapping. These actions can be hid and the algorithm can finish.

3.3 Limitations of the Enhanced Approach

There are several limitations in the presented approach that are mainly caused by its preliminary status. Because of the control flow restriction (no loops), only a subset of all possible interfaces can be covered. In consequence, *Burst* and *Collapse* operators were left out and cannot be produced by the automation algorithm. In addition, the approach does not allow two actions to be identical, as they cannot be distinguished during semantic matchmaking and would possibly lead to incorrect results. But, this restriction forbids service interfaces where one action needs to be performed twice, maybe with two different instances of the same type. The mapping produced in such a case would be quite confusing. On the other hand, it is not really ensured that semantic description are sufficient to perform semantic matchmaking. If the semantic descriptions are not detailed enough, the algorithm might produce incorrect results or no results at all.

4 Related Work

The problem of behavioral interface adaptation has been addressed in the field of component-based software engineering. In [14] a notion of compatibility between components is defined, where behavioral interfaces are described as finite state machines. For a given adaptor and two incompatible component interfaces a check is performed that verifies, if the adaptor can mediate between the interfaces. The generation of adaptors is based on links between similar operation parameters, but this does not involve semantic matching.

In the area of Web services, research focus has been on identifying "mismatch patterns" between two interfaces ([15], [16]). The adaptation is done via templates or by mapping between operational services (corresponding to the notion of actions in this approach). But, again these approaches do not consider automation or the use of semantics to enable automation.

A possible candidate for automated interface adaptation is the work presented in [17]. It deals with automatic composition of Web services. The composition is goal-oriented and achieves a coordination between two or more component services. It could be adopted to solve the problem of interface adaptation. For the required interface a corresponding compatible interface, in fact the consumer interface, must be determined. Then a composition goal has to be extracted from the semantic descriptions. With the goal and two service interfaces a composite service can be generated that mediates the interaction between consumer and provider.

5 Conclusion

This document presented an enhancement to the approach in [1]. The algebra and visual notation of [1] was introduced and their contribution to the problem of service interface adaptation was outlined. Acting on a few assumptions an algorithm was described that allows for automatic generation of interface transformation expressions. It was also pointed out that the approach has several limitations, which accumulate to the fact that a totally correct interface mapping cannot be ensured. For this reason, it must be stated that the approach in its current state is not feasible for automated service interface adaptation. However, a semi-automated application should be possible, e.g. executing the algorithm and manually checking the result.

Altogether, the adoption of semantic matchmaking techniques provide a promising approach to support automated service interface adaptation. Nonetheless, the future goal of any work in this areas must be full automation. Mismatches between service interfaces must be adapted on demand and within seconds, because they occur in a highly dynamic service marketplace scenario. To even go a step further, for a given behavioral interface it should be possible to generate all interfaces that can be adapted to this one, including appropriate adaptors. Then, a service consumer can simply choose the interface that suits him best and the adaptor is at hand immediately.

In order to get to this point, there are some things that should be further investigated. First, the approach has to be completed in terms of algebra support and support of control flow, so it can cover a broader set of behavioral interfaces. The algorithm should be implemented in a tool to easily test it with complex real-world scenarios, as this is the simplest way of identifying possible pitfalls. Regarding the problem of expressive semantic descriptions, it might be necessary to develop a methodology for annotating service interfaces with semantics. Such a methodology should encompass the whole lifecycle of service interface definition and would have to guide providers, as well as consumers, from initial design to publication and change management.

References

1. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation Business Process Management (2006) 65–80

2. Popa, L., Velegrakis, Y., Miller, R., Hernández, M., Fagin, R.: Translating Web Data In Proceedings of the 28th International Conference on Very Large Databases (VLDB'02) (2002) 598–609

3. Barros, A., Dumas, M., Bruza, P.: The Move to Web Service Ecosystems, BPTrends (2005), http://www.bptrends.com/publicationfiles/12-05-WP-WebServiceEcosystems-Barros-Dumas.pdf

4. MacKenzie, C., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0, OASIS (2006), http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf

5. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1, W3C Note (2001), http://www.w3.org/TR/wsdl/

6. The OWL Services Coalition: OWL–S: Semantic Markup for Web Services, (2003), http://www.daml.org/services/owl-s/1.0/owl-s

7. Kay, M.: XSL Transformations (XSLT) Version 2.0, W3C Recommendation (2007), http://www.w3.org/TR/xslt20/

8. Andrews, T., et al.: Business Process Execution Language for Web Services Version 1.1, (2003), http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

9. Bray, T., Paoli, J., Sperberg–McQueen, C., Maler, E., Yergeau, F., Cowan, J: Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation (2006), http://www.w3.org/TR/xml11

10. Farrel, J., Lausen, H.: Semantic Annotations for WSDL and XML Schema, W3C Candidate Recommendation (2007), http://www.w3.org/TR/sawsdl/

11. Unified Modeling Language Specification Version 1.4.2 OMG (2004), http://www.omg.org/docs/formal/04-07-02.pdf

12. Roman, D., Lausen, H., Keller, U., et al.: Web Services Modeling Ontology (WSMO), WSMO Final Draft (2006), http://www.wsmo.org/TR/d2/v1.3/

13. Kifer, M., Lausen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of the ACM **42(2)** (1995) 741–843

14. Yellin, D., Strom, R.: Protocol Specification and Component Adaptors, In ACM Transactions on Programming Languages and Systems **19(2)** (1997) 292–333

15. Alonso, G., Pautasso, C., Björnstad, B.: CS Adaptability Container, Deliverable #11, EU FP5 Project "ADAPT" (2004)

16. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H., Toumani, F.: Developing Adapters for Web Services Integration, In Proceedings of the 17th International Conference on Advanced Information System Engineering (2005) 415–429

17. Pistore, M., Roberti, P., Traverso, P.: Process-level composition of executable Web services: "on-the-fly" versus "once-for-all" composition, The Semantic Web: Research and Applications **3532** (2005) 62–77

A Appendix

A.1 Service Interface Example with Semantic Annotations

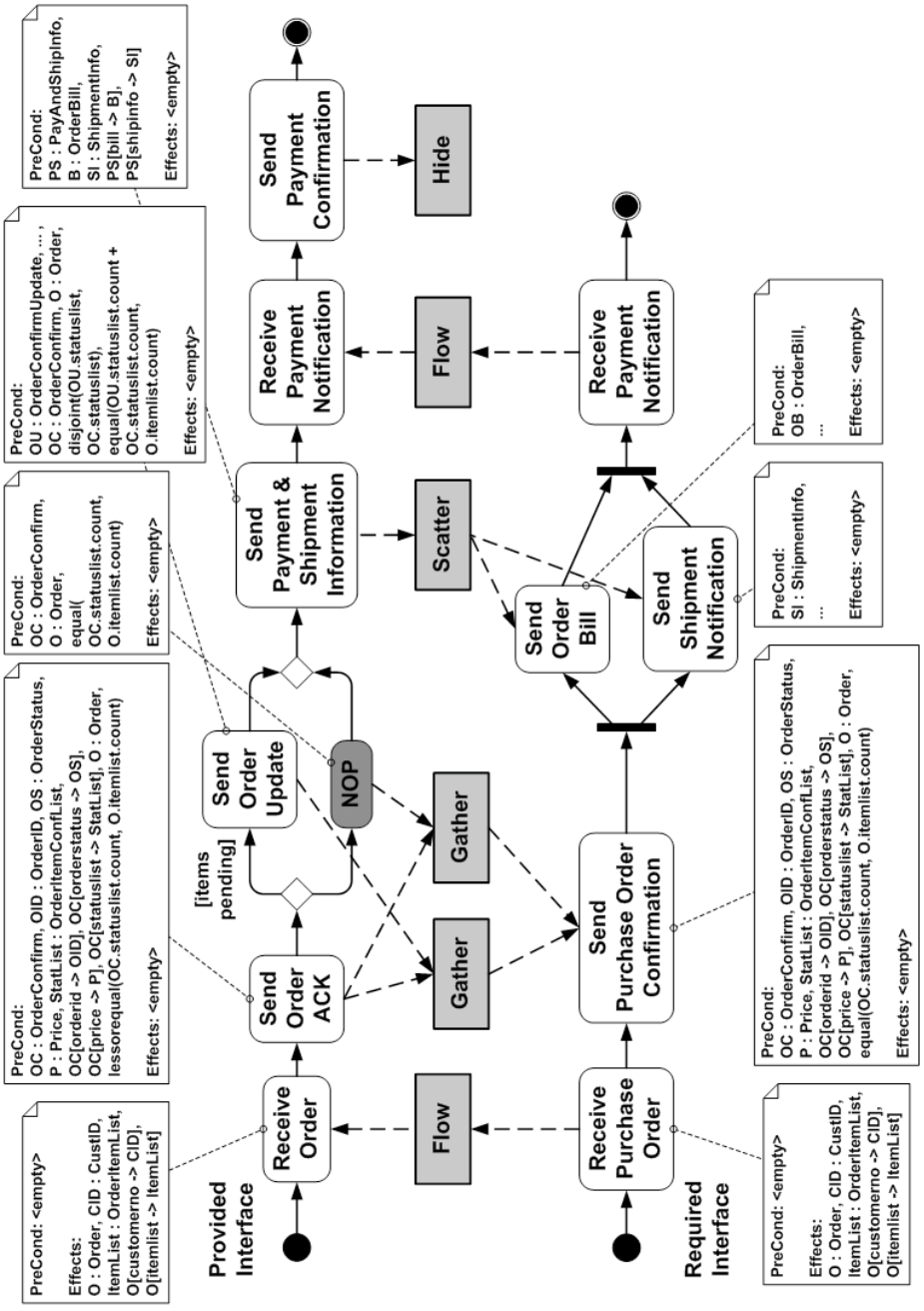

Fig. 7. Example of semantically annotated service interfaces with transformation between them.

A.2 Ontology (type system) of the Service Interface Example in F-Logic

```
ID :: string.
CustID :: ID.
ArtID :: ID.
OrderID :: ID.
BillID :: ID.

ItemNo :: integer.
Quantity :: integer.

Price :: decimal.

ACCEPTED : OrderStatus.
REJECTED : OrderStatus.

Order[ customerno => CustID,
       itemlist => OrderItemList].

OrderItemList[ count => Quantity,
               items =>> OrderItem].

OrderItem[ itemno => ItemNo,
           articleno => ArtID,
           amount => Quantity].

OrderConfirm[ orderid => OrderID,
              orderstatus => OrderStatus,
              price => Price.
              statuslist => OrderItemConfList].

OrderItemConfList[ count => Quantity,
                   conf_items =>> OrderItemConfirmation].

OrderItemConfirmation[ item => OrderItem,
                       itemprice => Price,
                       status => OrderStatus].

OrderConfirmUpdate[ orderid => OrderID,
                    newprice => Price,
                    statuslist =>> OrderItemConfirmation].
```

```
OrderBill[ billID => BillID,
           orderid => OrderID,
           amount => Price,
           accountinfo => AccountInfo,
           duedate => date].

ShipmentInfo[ orderid => OrderID,
              deliveryid => DeliveryID,
              shipmentdate => datetime
              exp_deliverydate => date].

PayAndShipInfo[ bill => OrderBill,
                shipinfo => ShipmentInfo].

PaymentNotify[ billID => BillID,
               paymentdate => datetime].

PaymentConfirm[ billID => BillID,
                orderID => OrderID].
```

A.3 Automation Algorithm in Pseudo-Code

The algorithm on the following pages is given in Java-like pseudo-code. Two functions are defined: `main` and `getSourceActions`. All other functions should be self-explanatory. Please refer to Sect. 3 for a textual description of the algorithm.

```
main:

input       :     PI, RI : Interface; // provided + required intf.
output      :     RES : vector<InterfaceTransformationExpression>;

{
  // mapping target actions to source actions (and vice versa)
  action_mapping : map<Action, vector<Action>>;
  reverse_action_mapping : map<Action, vector<Action>>;

  source_actions : vector<Action>;
  unused_actions_PI, unused_actions_RI : vector<Action>;

  transform_func : DataTransformationFunction;

  unused_actions_PI = actions(PI);
  unused_actions_RI = actions(RI);

  foreach ta in actions(PI) do {
    // look for "receives" in provided interface
    if (direction(ta) = IN) then {
      // get source actions from required interface
      source_actions = getSourceActions(ta, RI);
      // for a "receive" we need source actions in the
      // required interface
      if (source_actions = null) then
        exit(-1);
      else {
        action_mapping.put(ta, source_actions);
        unused_actions_PI.remove(ta);
        unused_actions_RI.remove(source_actions);
      }
    }
  }

  foreach ta in actions(RI) do {
    // look for "sends" in required interface
    if (direction(ta) = OUT) then {
      // get source actions from provided interface
      source_actions = getSourceActions(ta, PI);
      // for a "send" we need source actions in the
      // provided interface
      if (source_actions = null) then
        exit(-1);
      else {
        action_mapping.put(ta, source_actions);
        unused_actions_RI.remove(ta);
        unused_actions_PI.remove(source_actions);
      }
    }
  }
```

```
foreach ta in action_mapping.keyset() do {
  // GATHER: ta is mapped to several source actions
  if ( length(action_mapping.get(ta)) > 1 ) then {
    subset            : vector<Action>;
    action_subsets    : vector<vector<Action>>;

    // for "receives" add GATHER from RI -> PI
    if (direction(ta) = IN) then {
      // check, if each run of RI contains all source actions
      if ( !actionsInAllRuns(action_mapping.get(ta), RI) ) then {
        foreach run in runs(RI) do {
          subset = getActionsInRun(action_mapping.get(ta), run);
          if( isActionSetComplete(subset, ta) ) then {
            if(!action_subsets.contain(subset)) then
              action_subsets.add(subset);
          } else exit(-1);
        }
        foreach set in action_subsets do {
          // compute data transformation function from
          // source actions subset to target action
          transform_func = getDataTransfFunc(set, new vector(){ta});
          RES.add(new Gather(RI, set, transform_func, ta));
        }
      } else {
        // compute data transformation function from
        // all source actions to target action
        transform_func = getDataTransfFunc(action_mapping.get(ta),
                                           new vector(){ta});
        RES.add(new Gather(RI, action_mapping.get(ta),
                           transform_func, ta));
      }
    // for "sends" add GATHER from PI -> RI
    } else {
      // check, if each run of PI contains all source actions
      if ( !actionsInAllRuns(action_mapping.get(ta), PI) ) then {
        foreach run in runs(PI) do {
          subset = getActionsInRun(action_mapping.get(ta), run);
          if( isActionSetComplete(subset, ta) ) then {
            if(!action_subsets.contain(subset)) then
              action_subsets.add(subset);
          } else exit(-1);
        }
        foreach set in action_subsets do {
          // compute data transformation function from
          // source actions subset to target action
          transform_func = getDataTransfFunc(set, new vector(){ta});
          RES.add(new Gather(PI, set, transform_func, ta));
        }
      } else {
        // compute data transformation function from
        // all source actions to target action
        transform_func = getDataTransfFunc(action_mapping.get(ta),
                                           new vector(){ta});
        RES.add(new Gather(PI, action_mapping.get(ta),
                           transform_func, ta));
      }
    }
  }
```

```
  else if ( length(action_mapping.get(ta)) = 1 ) then {

    source_action : Action;
    // get the single action that is associated with ta
    source_action = action_mapping.get(ta).getFirst();

    if ( reverse_action_mapping.contains(source_action) ) then {
      // add ta to set of actions assoc. with source_action
      reverse_action_mapping.get(source_action).add(ta);
    } else {
      // add a new set of actions assoc. with source_action
      reverse_action_mapping.put(source_action,
                                      new vector(){ta} );
    }
  } else {
    // this case should be caught in the loops above
    exit(-1);
  }
}

foreach sa in reverse_action_mapping.keyset() do {
  // FLOW: only one action maps to sa
  if ( length(reverse_action_mapping.get(sa)) = 1 ) then {

    // compute data transformation function from
    // source action to target action
    transform_func = getDataTransfFunc(new vector(){sa},
                                       reverse_action_mapping.get(sa));

    if (direction(sa) = IN) then
      // for "receives" add FLOW from RI -> PI
      RES.add(new Flow(RI, sa, transform_func,
                           reverse_action_mapping.get(sa)));
    else
      // for "sends" add FLOW from PI -> RI
      RES.add(new Flow(PI, sa, transform_func,
                           reverse_action_mapping.get(sa)));

  }
  // SCATTER: several target actions map to sa
  else if ( length(reverse_action_mapping.get(sa)) > 1 ) then {
    subset            : vector<Action>;
    action_subsets    : vector<vector<Action>>;
    // the placement (i.e. control flow) is determined by
    // the target interface (PI in 1st case, RI in 2nd)

    // for "receives" add SCATTER from RI -> PI
    if (direction(ta) = IN) then {
      // check, if each run of RI contains all target actions
      if ( !actionsInAllRuns(reverse_action_mapping.get(sa), RI) ) then {
        foreach run in runs(RI) do {
          subset = getActionsInRun(reverse_action_mapping.get(sa), run);
          if( isActionSetComplete(subset, sa) ) then {
            if(!action_subsets.contain(subset)) then
              action_subsets.add(subset);
          } else exit(-1);
        }
        foreach set in action_subsets do {
          // compute data transformation function from
          // source action to target actions subset
          transform_func = getDataTransfFunc(new vector(){sa}, set);
          RES.add(new Scatter(RI, sa, transform_func, set));
        }
```

```
        } else {
          // compute data transformation function from
          // source action to all target actions
          transform_func = getDataTransfFunc(new vector(){sa},
                                             reverse_action_mapping.get(sa));
          RES.add(new Scatter(RI, sa, transform_func,
                                             reverse_action_mapping.get(sa)));
        }
      // for "sends" add SCATTER from PI -> RI
      } else {
        // check, if each run of PI contains all target actions
        if ( !actionsInAllRuns(reverse_action_mapping.get(sa), PI) ) then {
          foreach run in runs(PI) do {
            subset = getActionsInRun(reverse_action_mapping.get(sa), run);
            if( isActionSetComplete(subset, sa) ) then {
              if(!action_subsets.contain(subset)) then
                action_subsets.add(subset);
            } else exit(-1);
          }
          foreach set in action_subsets do {
            // compute data transformation function from
            // source action to target actions subset
            transform_func = getDataTransfFunc(new vector(){sa}, set);
            RES.add(new Scatter(PI, sa, transform_func, set));
          }
        } else {
          // compute data transformation function from
          // source action to all target actions
          transform_func = getDataTransfFunc(new vector(){ta},
                                             reverse_action_mapping.get(sa));
          RES.add(new Scatter(PI, sa, transform_func,
                                             reverse_action_mapping.get(ta)));
        }
      }
    } else {
      // this case should not occur
      exit(-1);
    }
  }

  // looking for actions to HIDE

  foreach a in unused_actions_PI do {
    // only "sends" can be hid in the provided interface
    if (direction(a) = OUT) then
      RES.add(new Hide(PI, a));
    else
      // this case should not occur (handled in first loop)
      exit(-1);
  }

  foreach a in unused_actions_RI do {
    // only "receives" can be hid in the required interface
    if (direction(a) = IN) then
      RES.add(new Hide(RI, a));
    else
      // this case should not occur (handled in second loop)
      exit(-1);
  }

  return RES;

}
```

```
getSourceActions:

                  // a target action and the source interface
input       :     TA : Action; SI : Interface;
output      :     RES : vector<Action>;

{
  // vector to store a set of actions, that all together might
  // be suitable to provide necessary data for TA
  potential_src_actions : vector<Action>;
  // vector to store a set of preconditions or effects
  unified_conditions : vector;

  // the result of a semantic matchmaking
  matching_result : enum{FULL_MATCH, PARTIAL_MATCH, NO_MATCH};

  potential_src_actions = new vector<Action>();

  foreach sa in actions(SI) do {

    // for "receives" only preconditions are of interest
    if ( direction(TA) = IN ) then {
      // perform matchmaking with TA and sa
      matching_result = match(precond(TA), precond(sa));

      // sa fully matches TA
      if ( matching_result = FULL_MATCH) then {
        RES.add(sa);
      }
      // sa partially matches TA
      else if (matching_result = PARTIAL_MATCH) then {
        // add sa to set of potentially matching src. actions
        potential_src_actions.add(sa);
      }
    }
    // for "sends" only effects are of interest
    else {
      // perform matchmaking with TA and sa
      matching_result = match(effects(TA), effects(sa));

      // sa fully matches TA
      if ( matching_result = FULL_MATCH) then {
        RES.add(sa);
      }
      // sa partially matches TA
      else if (matching_result = PARTIAL_MATCH) then {
        // add sa to set of potentially matching src. actions
        potential_src_actions.add(sa);
      }
    }
  }
```

```
  // compute the union of semantic descriptions
  // (preconditions and/or effects) of potential src. actions
  // and fully matching src. actions (already stored in RES)
  //
  // The following steps ensure, that only a set of actions
  // is returned, which contributes all necessary data to TA.
  // Anyhow, there might be actions in the set that seem to
  // contribute necessary (parts of the) data to TA, but
  // which actually should not be mapped to TA at all.
  // It all depends on the expressiveness and provided detail
  // of the semantic description.
  unified_conditions = semanticUnion(new vector(){RES,
                                                  potential_src_actions});
  if ( direction(TA) = IN) then {
    // match TA's preconditions against this union
    matching_result = match(precond(TA), unified_conditions);
  } else {
    // match TA's effects against this union
    matching_result = match(effects(TA), unified_conditions);
  }
  // test, if the unified preconditions or effects
  // fully match with TA
  if (matching_result = FULL_MATCH) then {
    RES.add(potential_src_actions);
  }

  // source actions have been found
  if (length(RES) > 0) then {
    return RES;
  }
  // no source actions have been found
  else {
    return null;
  }

}
```

The SOA revisited for multilateral collaborations

Martin Probst

Hasso-Plattner-Institute for Software Systems Engineering, Potsdam
martin.probst@student.hpi.uni-potsdam.de

Abstract. The Service Oriented Architecture (SOA) is a popular architectural style that introduces the notion of publishing, discovering and consuming services. In multilateral collaborations, several participants take part in choreographies that are not controlled by a single entity in the system. This paper presents a survey of possible scenarios how participants publish and discover their services to enter a conversation.

1 Introduction

Service Oriented Architectures (SOA) are by now a widely accepted and used architectural approach to the design and implementation of IT systems. In a service oriented architecture, the system at hand is decomposed into a set of services. These services are independent, well-defined subsystems that are aligned with the business use of the services. Services are then published, can be discovered, and consumed by clients or other participants in the system.

As of today, most SOA implementations are deployed within organizations, as opposed to inter-organizational deployments, and services follow a basic request and response pattern. By combining multiple services in so-called *Orchestration*s, services can be aggregated to form applications or services of higher value.

This notion of composing services however has limitations. Orchestrations usually imply a client-server relationship between participants, and have a single executing or driving participant, that controls execution. This concept does not map well to existing business to business processes. Long running, complex interactions between services, so called *Conversations*, are executed in many business to business scenarios. These scenarios typically do not have a single, executing instance, but are rather conversations between two or more independent agents.

A *Choreography* is the abstract description of a set of conversations. It defines the set and order of allowed interactions between the participants in these conversations. Choreographies are characterized by the lack of a central controlling instance and therefore a more collaborative approach (cf. [Pel03]).

This paper investigates issues raised by the notion of a multilateral Service Oriented Architecture. In particular, the question of how participants can publish, discover and consume interactions in multilateral scenarios is analyzed.

2 Related work

A prerequisite for the work presented in this paper are two important concepts, *service compatibility* and *semantic web services*.

2.1 Service compatibility

Service compatibility is not limited to pure syntactic compatibility (cf. [Mar03]). Two services are compatible if they can interact in a successful way, where successful can be defined as a set of properties. An example set might be the following, while other or more properties are imaginable.

- no deadlocks occur
- the communication protocol between the services matches
- all participants reach a valid terminal state after the interaction
- all services can reach a successful terminal state

Public view An initial definition of service compatibility was the public view, as introduced in [LRS02]. The public view of a web service is its interface to other services, the messages it sends or receives and their order in its execution life time. Public view has not been strictly specified but rather gives an intuitive understanding of the concept. It has little use in practice as it does not allow for automated comparison or checking of complex choreographies.

Operating guidelines To address the limitations of the public view concept, [Mar05] proposes the idea of *Operating Guidelines*. With operating guidelines, a web service is seen as an automaton. The states of the automaton represent legal interaction states of the service, and transitions between the states represent sent or received messages. By this, the automaton represents all legal behaviours of the service, and clients that follow the service's operating guidelines can interact successfully with the service.

Operating guidelines have the advantage of being relatively easy to generate from other, more complex models, like Petri nets or BPMN descriptions. Also, operating guidelines automata can easily be checked for compatibility by following the automatons state transitions, which is always O(n) relative to the number of nodes in the automaton. On the other hand they are limited to communication protocols without loops and the automata can get very large for complex services.

2.2 Semantic web services

The term semantic web service describes a web service that was annotated with a semantic meaning, so that clients can request a certain operation without actually specifying a concrete web service implementing it. Various technologies have been proposed to implement semantic web services (cf. [MPM+04], [dBLK+05],

[AFM+05], [RLK05]). Multilateral collaborations require ways to identify the semantics of single services, so that participants and/or a service broker can select services to take part in choreographies while ensuring semantic correctness (service matchmaking).

3 Choreography Knowledge

This section will investigate four scenarios, in which multiple participants engage in a conversation. The scenarios have been selected to cover different types of choreographies by the criterium of *choreography knowledge*. The term choreography knowledge here denotes both the structure of the process implemented by a certain choreography and the actual identity of participants in the process.

A multilateral collaboration requires the coordination of more than two participants. In order to make multilateral collaborations possible, single participants need to be able to receive messages from multiple other services, possibly services that were not known to them when the conversation was instantiated. This capability is sufficient if another participant manages the conversation. Therefore for a single participant, it is possible to ignore the multilateral nature of the choreography. If choreography knowledge is optional to some of the participants, the overall architecture of a system will be greatly influenced by the question which participants need to have knowledge of the choreography.

If a participant holds knowledge about the choreography, this knowledge has to be communicated to other participants or the broker. Also, this choreography knowledge will have to be adapted to changes in the system, and compatibility between participants will have to be maintained, thus requiring significantly 'smarter' participants. The knowledge about a choreography may also be used to handle error conditions.

The participants that share choreography knowledge will need to communicate about their process models and agree on compatibility. Also, only participants with choreography knowledge will be able to set up other participants to take part in the conversation.

Choreography knowledge defines a view on the system. If a participant only knows its own interface to the system, it has a local view of the whole conversation - it does not know about interactions between other participants (cf. [ZDtH+06]).

Possible multilateral SOA systems can be divided into the following groups using the criterium of choreography knowledge:

Single participant a single participant holds the choreography knowledge, other participants have a local view

Global view multiple participants have a global view of the collaboration, i.e. they know about interactions between participants outside of their local view.

Participant knowledge participants have a local view of the system, but have requirements on the identity of their communication partners. They are

aware of the multilateral nature of the conversation, but do not require a global view.

Participant identity is generally an orthogonal property to the knowledge of the choreography's process, but still closely related. A global view of a conversation is only meaningful if the actual participants that communicate can be identified.

3.1 Scenario I: Marketplace

The first scenario is called *Marketplace*, because it is actually implemented and in use today by services such as the Amazon Marketplace. In it, a **single participant** has the choreography knowledge.

A provider, such as an online store, publishes its service, e.g. the process of selling goods to consumers. The process consists of the selection of goods by a consumer and shipment of the goods to the consumer (payment is not considered for the example). This basic service can be extended into a multilateral conversation if the online store allows other retailers to use its infrastructure, but still handles the presentation of goods. Other retailers can now register with the online store for a 'goods provider' service.

In this scenario, shown in figure 1, one of the participants – the online store – manages the conversation composition. The other participants, consumers and retailers, have to be aware that they need to interact with different communication partners, but they do not need further knowledge of the process.

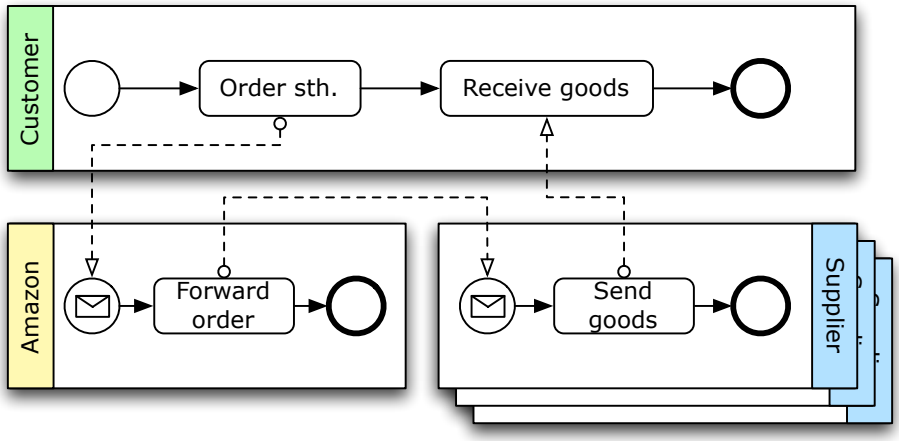

Fig. 1. Amazon Marketplace scenario

The online store transparently handles the conversation and might also act as a service broker, which is plausible for non-technical reasons. If the online

store defines the rules of the conversation and manages the conversation, it also carries the responsibility for the technical infrastructure, and would therefore probably provide a broker. Requests to take part in conversations do not require any special information beyond the semantic service description and a means to ensure process compatibility, e.g. an operating guidelines description of the local interface.

This scenario can easily be extended to contain more complex interactions between participants or more participant roles. It is also possible to layer the role of the online store, e.g. to have multiple participants with choreography knowledge that provide a simple service to the other managing participants. The important property is that only one participant is concerned with the particular choreography it manages, and the other participants do not need to know about the choreography.

3.2 Scenario II: Rent Deposit

The rent deposit scenario describes the case that some participants have **participant knowledge**. It is in that way an extension of the first scenario, as the single participants do not need to have knowledge of the global process, but only of their interface.

In this scenario, a landlord wants to rent a flat to a lodger. To make sure he will receive his rent, he requires the lodger to make a rent deposit with a bank. The lodger is responsible for the choreography, as the online store was in section 3.1, but the landlord requires that the confirmation of the rent deposit is sent to him by a bank. This requires an extended description of the service interfaces in the choreography. Lodger and landlord have to negotiate what a bank is in their context. A service broker that performs a matchmaking will need to receive information from the lodger about the required role of the participant, so requests will consist of the normal service identification plus a semantic identification of the involved participants.

3.3 Scenario III: Construction Planning

The third scenario extends the second insofar as participants require knowledge of the full choreography, i.e. a **global view**.

A client wants to construct a building. She assigns an architect with creating a construction plan. The architect submits the plan to the building authority and sends it back to the client afterwards. The client then assigns the construction work to a construction company. This company is required to ask the building authority for permission. Both the architect and the client are required to know about the interaction between the construction company and the building authority (figure 3).

This scenario requires a way to specify at least parts of the interaction. The client needs a way to express the interaction behaviour between other participants. Also, participants need to communicate this knowledge over boundaries, e.g. the architect has to make sure he communicates with the same building

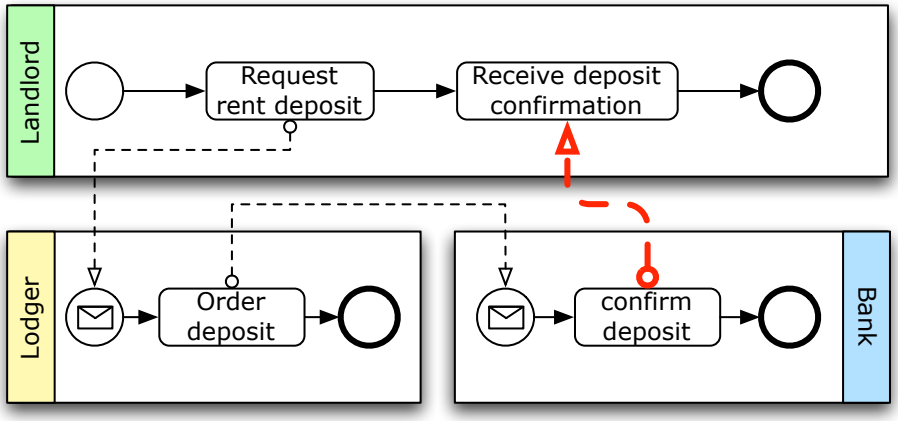

Fig. 2. Rent Deposit

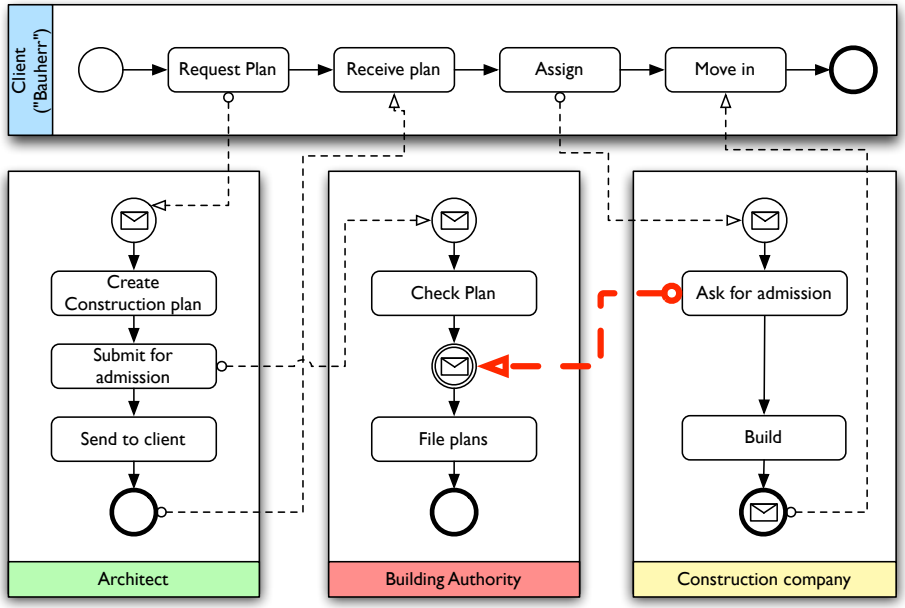

Fig. 3. Construction planning

authority as the construction company. This knowledge might be represented as a Let's dance plan with semantic annotations both for the operations and the participants (cf. [ZDtH+06]), which will need to be included in requests to a service broker.

3.4 The role of the service broker

The idea of a service broker contradicts with the concept of a truly multilateral choreography. If participants in a conversation are completely independent of each other, and no participant controls execution of the choreography, it is highly unlikely that one can find a single point where all participants register their services. However the functionality provided by a service broker, that is service matchmaking, will still be needed.

The service broker will loose its special position in the architecture and become a service itself. Most likely single participants of a choreography will provide broker-like functionality as local hubs that allow the discovery of other participants. Therefore, none of the diagrams depicts an explicit broker.

4 Conclusion and Outlook

The scenarios presented in this paper represent real-world use cases that need to be addressed by service oriented architectures. Basic technology to describe the scenarios and communicate them is available, so that combining the technologies to address the scenarios is possible. But the major hurdle in the adoption of multilateral choreographies is non-technical, as error handling, responsibility for process outcome, legal liability etc. are much bigger problems.

4.1 Automated planning

A desirable technical solution would be the fully automated planning of a choreography. In this scenario, a participant would request to take part in a conversation and supply a semantic description of the choreography to a service broker. The service broker would then use planning algorithms to find a combination of the registered services that fulfills the choreography and return the planned conversation to the client.

While attractive, this scenario is quite unlikely in the near future. The automated planning is technically very challenging. Also, the existence of one service broker that knows about all other services is unlikely, as explained in section 3.4, so there might not be an instance that has sufficient knowledge to do a complete planning.

A truly automated planning of independent, inter-organizational choreographies would in the long term have to include all the organizational and legal features of today's regular business system. Especially in the areas of error handling and liability solutions would be needed. The 'real world' system includes an enormous set of rules, is guarded by the legal system and error and corner cases

are handled by humans. A system trying to emulate this would be a herculean task.

4.2 Plausible scenarios

A likely scenario in the near future is characterized by Scenario I. Current service providers will start to act as both providers and brokers. Well know and clearly defined choreographies will be published by these providers and other participants can register to take part in them. But these choreographies will still be mainly managed by the initial service provider. Another likely scenario are brokers that provide a set of known choreographies where participants can register for roles in these well known choreographies.

Bibliography

[AFM+05] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web Service Semantics-WSDL-S. *A joint UGA-IBM Technical Note*, 1, 2005.

[dBLK+05] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. The Web Service Modeling Language WSML. *WSML Final Draft D*, 16, 2005.

[LRS02] F. Leymann, D. Roller, and M.T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[Mar03] A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.

[Mar05] Axel Martens. Consistency between Executable and Abstract Processes. In *Proceedings IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005)*, pages 60–67, Hong Kong, China, March 2005. IEEE Computer Society.

[MPM+04] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, et al. Bringing Semantics to Web Services: The OWL-S Approach. *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, pages 6–9, 2004.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[RLK05] D. Roman, H. Lausen, and U. Keller. Web Service Modeling Ontology (WSMO). *WSMO Final Draft April*, 13, 2005.

[ZDtH+06] Johannes Maria Zaha, Marlon Dumas, Arthur ter Hofstede, Alistair Barros, and Gero Decker. Service Interaction Modeling: Bridging Global and Local Views. In *Proceedings 10th IEEE International EDOC Conference (EDOC 2006)*, Hong Kong, Oct 2006.

Bridging Global and Local Interaction Models Using Petri Nets: Generating Interface Processes out of Interaction Nets

Silvan T. Golega

Hasso-Plattner-Institut, Universität Potsdam, Germany
silvan.golega@hpi.uni-potsdam.de

Abstract. In a Service-Oriented Architecture (SOA), systems interact as independent units through message exchange without the need of central orchestration. Each participant takes care of the parts it contributes to the overall process. To avoid compatibility conflicts, global service interaction models, so-called choreographies, describe a bird's eye view of the interactions in highly distributed systems. Control flow constraints of the global view have to be accomplished by the local interface processes. To fill the gap between the global and the local view this paper introduces an abstract syntax for choreographies based on Petri nets and two algorithms capable of generating interface processes out of interaction nets considering all the control flow constraints of the global choreography.

1 Introduction

Various shifts can be observed in the area of collaborating systems. Services get more and more flexibly interchangeable and recombineable. Descriptions of semantics and process interfaces enforce these trends. The newest shift points to a move from centrally orchestrated towards highly distributed services. The combination of various local interface processes leads to compatibility problems. Many parties argue therefore to first create a global service interaction model, i.e. a choreography. From this global model local models and behavioral interfaces can be derived that are compatible automatically. However, the derivation of local models leaving intact all constraints that are expressed in the global model is not trivial.

This work aims at achieving various contributions in this area to fill the gap between global and local views. The paper introduces messages and participant roles as an extension to Petri nets. This extension enables Petri nets to model interactions between various business partners and to be used as a choreography language. Furthermore, a definition of strict enforceability is presented and used in a provable algorithm that derives local interface processes from strictly enforceable interaction nets. The last contribution is an algorithm for the generation of interface processes out of enforceable interaction nets.

The outline of the paper is the following. The next Chapter details prerequisites that are needed for the generation algorithms. It introduces the above

mentioned Petri net extensions as well as the terms *enforceability* and *strict enforceability*. Chapter 3 then describes the first of two algorithms. This algorithm generates interconnected local Petri nets out of a global interaction Petri nets. It is called *service-oriented approach*, because the overall control flow is not expressed in each local interface process. Instead the combination of the different participants enforces the execution constraints of the global model. It is valid for strictly enforceable nets. The second algorithm is called the *process-oriented approach*. Its adaptability for enforceable nets and its up- and downsides are discussed in chapter 4. A comparison of both algorithms, related work and an outlook to this work will be given in Chapter 5.

2 Background

2.1 Petri Nets in Distributed Systems

This chapter outlines how Petri nets can be used for choreographies. It motivates an extension that introduces participant roles and messages into Petri nets.

Petri nets[1] are a mathematical model for parallel systems with a well established graphical representation. They provide a strong mathematical background that has been investigated during several decades of research. Much work has been done to create a basis for reasoning, which could be useful in the context of choreographies. The Petri nets' ability to represent parallel systems and their simple mathematical definition suggest using them for automating purposes in the area of net transformations.

Unlike other process descriptions, choreographies do not display dependencies between actions, but control flow between interactions. Two main reasons for using choreographies are providing an overview of the communication between different roles and the derivation of local interface processes. The decision which activities have to be done to produce and process messages are totally left to the concrete participant. The participant is only required to be compatible and conform with the choreography. Graphical choreography languages like *ISDL* [3] or *Let's Dance* [2] introduce nodes with the semantics of messages and are lacking elements for activities. They have a strong focus on participant roles including them directly into their graphs.

The following paragraphs outline how these principles and foci on messages and participants can be applied to Petri nets.

"Original" Petri nets are sometimes considered to include arc weights and place capacities. This paper does not take these extensions into account. However, with some minor adaptions the presented algorithms should be usable for this class of Petri nets, too. Here a Petri net consists of the following quadruple (P, T, F, M_0)

– P, a finite, non-empty set of places;
– T, a finite, non-empty set of transitions, where P and T are disjoint: $P \cap T = \emptyset$;
– F, a set of arcs known as a flow relation, where no arc in F must connect two places or two transitions: $F \subseteq (P \times T) \cup (T \times P)$;
– $M_0 : P \to \mathbb{N}$, an initial marking, where for each place $p \in P$, there are $n \in \mathbb{N}$ tokens.

Two changes are evident compared to classical Petri nets. The first one is to interpret the transitions in the set T as interactions. The second is to introduce participant roles to the model. This has to be done separately for the global interaction model and for the local interface processes. The global interaction net will therefore be the tuple $(P, T, F, M_0, R, \sigma, \rho)$, where

– R is a finite, non-empty set of participant roles, disjoint from P and disjoint from T: $R \cap T = R \cap P = \emptyset$;
– $\sigma : T \to R$ is the sender relation, assigning each interaction $t \in T$ an participant role $\sigma(t) =: r \in R$;
– $\rho : T \to R$ is the receiver relation, assigning each interaction $t \in T$ an participant role $\rho(t) =: r \in R$, where $\sigma(t) \neq \rho(t)$.

To be able to provide an overview picture of the collaboration between the separate local interface processes yet another tuple will be introduced. The transitions here get the semantics of sending or receiving activities, a place between them symbolizes message exchange when containing a token. Each transition is mapped to one participant role, the executor of this transition. The overview net for the local interface processes therefore is the tuple $(P, T, F, M_0, R, \phi,)$, where

– R again is a finite, non-empty set of participant roles, disjoint from P and disjoint from T: $R \cap T = R \cap P = \emptyset$;
– Φ is the executor relation, assigning each interaction $t \in T$ an participant role $r \in R$: $\Phi \subset (T \times R)$

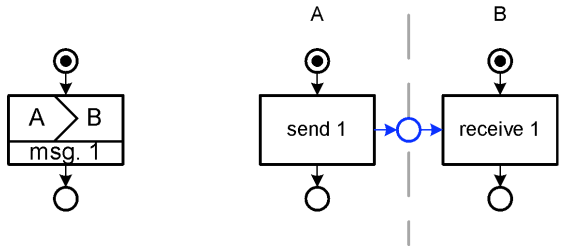

Fig. 1. A global interaction net and its connected set of local interface processes

A graphical notation for these net formalizations following the graphical syntax of Let's Dance for the interaction model and the concept of having *swim lanes* for each local interface process is illustrated in figure 1.

2.2 Enforceability

A global interaction model is locally enforceable if there exists a set of local interface processes that only contain interactions from the global model and collectively enforce all constraints from the global model. [4] provides a formal definition for enforceability. Goal of this work is to find an algorithm that derives a set of local interface processes from a global interaction model if it is enforceable. The output of an application of the presented algorithms to unenforceable interaction models is undefined and most probably not meaningful.

However, it seems to make sense to introduce another, a stricter enforceability function for the net $(P, T, F, M_0, R, \sigma, \rho)$.
Let $preT(Ts) \subseteq P$ be the set of all places preceding a set of transitions $Ts \subseteq T$:
$p \in preT(Ts) \Rightarrow \exists t \in Ts$, such that $(p, t) \in F$
Let $preP(Ps) \subseteq T$ be the set of all transitions preceding a set of places $Ps \subseteq P$:
$t \in preP(Ps) \Rightarrow \exists p \in Ps$, such that $(t, p) \in F$
Then $preP(preT(\{t\})), t \in T$ is the set of all transitions directly preceding t.
A global model is strictly enforceable if and only if all messages are to be sent by a participant that was involved either as sender or as recipient in all directly preceding interactions, or formally speaking a global model is strictly enforceable iff $\forall t \in T, f \in T : f \in preP(preT(\{t\})) \Rightarrow \sigma(t) \in \{\sigma(f)\} \cup \{\rho(f)\}$

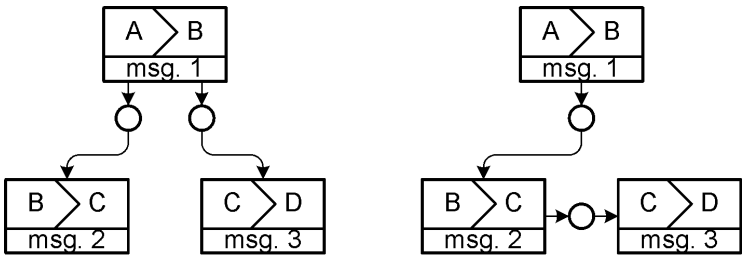

Fig. 2. An enforceable net and its transformation to a strictly enforceable net

Figure 2 shows an example of an enforceable but not strictly enforceable global net on the left. It is not strictly enforceable, because the sender C of message three is not directly involved in all directly preceding interactions, namely message one, neither as receiver nor as sender. However, the net is enforceable: Whenever C receives message two it knows from the structure of the net that message one must have been sent, too, and can send message three.

As strictly enforceable nets are easier to handle under certain aspects, it is sometimes desirable to transform enforceable but not strictly enforceable into

strictly enforceable nets. Often this can be done by overspecification as also displayed in figure 2. This paper does not outline how this transformation can be achieved, it is left subject to future work.

3 Service-Oriented Approach

Given a certain interaction net, there are basically several solutions thinkable for conforming interaction processes. They differ in terms of their representation of the control flow and their degree of overspecification. This chapter and the following depicts two rather different approaches. The first one, called the service-oriented approach, hides the control flow behind the overall structure of the collaborating local interfaces leaving open all degrees of freedom in execution. The second approach, discussed in the following section 4, limits these liberties by adding further constraints, but makes the complete control flow visible on each participant's runtime environment. It is therefore called the process-oriented approach. There are several other solutions possible, but these two have been valued to offer the best tradeoff between their strengths and weaknesses. For more details see their comparison in the conclusion, chapter 5.1.

Each of these solutions can be achieved using different algorithms. Three different techniques will be mentioned here, however only the first will be described in the following chapter 3.1. The "multiply and cut" technique duplicates the whole interaction net to each participant's interface process, later connects them and then takes away all spare elements. "Distribute and add" spreads all elements of the choreography to the swim lane of its executor leaving connections intact and then adds elements that are needed to fulfill the control flow conditions. The third modus operandi "tracing paths" simulates all possible execution paths of the choreography, noting down all possible execution traces, and later connecting these interface processes. "Multiply and cut" has been chosen to be discussed here in more detail, because it seems to imply the less complex steps.

3.1 Algorithm

This chapter describes an algorithm to generate the service-oriented solution for local interface processes. Its name results from the fact that the distinct receive and send transitions which one single participant role executes are not necessarily connected through control flow. Instead receive transitions just wait to get enabled statelessly and without directly belonging to a process instance. Receive and send transitions are called receives and sends for easier readability in this paper. An example for a set of local service-oriented interface processes and their global interaction net is given in figure 3.

The first step of the algorithm is to copy the whole interaction net into each swim lane. This is done in lines 6 - 28 in listing 1. All transitions as well as their incoming and outgoing flows are only copied, if their participant role corresponds either to their sender or their receiver participant. In the next step

places are added between the sending and receiving transitions representing the communication channels between the local interfaces (lines 31-37).

As some transitions and alternatives have been removed in the local processes some of the transitions might not get enabled when the removed way is chosen. Due to the strict enforceability this problem can only appear to receives, not to sends. The solution handles this by removing all incoming flow to receives except of communication channels. Receives will get enabled only by receiving a message. This is achieved through the *if*-clause in line 25. The model for local interface processes described in chapter 2.1 defines the set Φ only for transition-participant role pairs. Line 17 copies these pairs to the new executor relation. $(P_n, T_n, F_n, M_{0,n}, R, \Phi_n)$ defines the the new set of local interface processes that are described by the interaction net $(P, T, F, M_0, R, \sigma, \rho)$.

The proof for validity of this algorithm will not be presented in greater detail here, but a sketch of it: Because the interaction net is strictly enforceable all control flow constraints are expressed directly in the structure of the net. None of the direct predecessors of a send transition were removed – again because of the net's property of strict enforceability. So all control flow logic that enables sends remains after the generation process. A receives is enabled directly by its associated send. As there are only receive and send transitions all logic is still expressed in the derived net.

3.2 Evaluation

The strong advantage of this algorithm is its simpleness as well in the generating process as in its result. The algorithm produces almost no overhead (i.e. newly introduced elements that cannot be found in the global interaction net) and leads to very simple interface nets. However on the local side there are no real connected local process. Several parts of the process remain without any connection between them in a local view, only a bird's eye view reveals those control flow dependencies. This leads to the fact that without further knowledge of the above lying structure one can revice see if a process finished on the local side only from looking into the local interface process's runtime state. Furthermore it is only valid for strictly enforceable nets. Non-strictly enforceable nets like in figure 2 have a need of further supervison of implicit control flow dependencies.

4 Process-Oriented Approach

4.1 Algorithm

This chapter presents an algorithm for enforceable interaction nets that leaves the control flow dependencies between different participant roles intact and visible on each client's swim lane. An example for a derived model where this algorithm was used can be found in figure 4. Its respective interaction net is the same as displayed in figure 3.

Just as in the service-oriented approach the algorithm copies the whole interaction net into each swim lane. This is done in lines 6-28 in listing 2. Doing this the sender and receiver relations σ and ρ are removed, but this time all transitions are copied into the swimlanes. Transitions that are neither sends nor receives become no operation transitions, so-called *nops*, which do not carry any interaction semantics but which are needed for the control flow. The next step adds places between the sending and receiving transitions (lines 31-38). Tokens interchanged in these communication places represent messages between different participant roles.

An alternative algorithm described in the next section 4.2 would stop here. But several constraints of the global model now are not expressed in the local models anymore. E.g. in alternative execution paths, if one way begins with a nop, the other with a receive, the runtime environment will have no information if to trigger the enabled nop or if still to wait for incoming messages on the receive path alternatively. The proposed solution is to introduce so called *milestone edges* that inhibit the firing of nops until it is clear from a later message that they should fire. The algorithm to add these milestone edges is shown in listing 3.

4.2 Evaluation

The big advantages of this solution are its aptitude for all enforceable nets and the clearly visible processes in each local swim lane. If the local net allows it, termination of local execution can be revised easily with this solution. The negative consequence is the amount of overhead that is produced in the local interface processes. Unfortunately, the correctness of the algorithm was not yet proven for all enforceable nets, although it seems promising. One possible downside can be seen in the introduction of nops and milestone edges that have no direct counterpart in workflow nets.

To face the negative sides of this algorithm - the overhead and the introduction of new elements - several variants of the algorithm are possible. Two of them are mentioned here. One possibility is to skip the last step, the adding of milestone edges and duplicated alternative nops. The net then looses a big part of its overhead but also its validity for enforceable, non-strictly enforceable nets. However a supervisor in knowledge of the choreography and its implicit control dependencies could use this simplified version for non-strictly enforceable nets, too, if it itself controls triggering of the nop transitions.

The second alternative consists in duplicating receives that follow nops to all alternative branches to avoid the introduction of milestone edges. This rather complex algorithm will add a big amount of overhead to the net. The three variants of the process-oriented net are compared in table 1.

5 Conclusion and Related work

This last chapter puts the obtained concepts into a broader context. The next section compares the two mentioned generation approaches among each other,

Table 1. Comparison between different approaches to generate process-oriented interface processes

	overhead produced	new elements	duplicate transitions	validity
no action	none	none	no	strictly enforceable nets
duplication of receives	very high	none	yes	enforceable nets
adding of milestone edges	high	nop - transitions milestone edges	no	enforceable nets

section 5.2 lists related work and the concluding paragraphs outline possible future work.

5.1 Comparison of Approaches

The two solutions presented in the previous chapters feature different qualities. The service-oriented one appears to be adequate in asynchronous environments, where revisable termination states are not prioritized. It has the strong convenience of adding only a minimum of overhead to the local processes, does not introduce new elements, and does not add unnecessary control flow that would lead to overspecification. Unfortunately, this leads to the consequence that this approach is only capable of serving strictly enforceable nets. Enforceable but not strictly enforceable nets need an additional instance of execution level that lies over the Petri net blocking and triggering certain enabled transitions.

The process-based approach on the other hand can use overspecifiation to enforce the constraints of the global model in the local model. If it does, overhead in the local models augments, but in an exchange derived local models are valid from all enforceable interaction nets, being process-oriented and enabling certain additional possibilities like the check of the reach of a defined termination state. Table 2 provides a comparison of the two algorithms.

5.2 Related Work

In [5] a similar approach to derive local from global models is presented. Its algorithm is based on Let's Dance as choreography language. As the algorithms presented here produce standard Petri nets for each local participant, they have a slight advantage over the results in the very young language Let's Dance. However, the authors have formalized the Let's Dance control flow elements using π-calculus and thereby created a basis for intercompatibility and executability.

Paolo Traverso et al. provide in [6] a rather different process model where global and local views are developed hand in hand instead of generating one out

Table 2. Comparison between the the two algorithms that produce service-oriented and process-oriented interface processes

	local interface	revisable termination	produces overhead	new elements	coverage
service-oriented. interface	loosely coupled services	no	small	no	strictly enforceable nets
process-oriented. interface	one process	yes	big	yes	enforceable nets (not yet proven)

of the other. This way, compatibility and conformance issues are meant to be minimized. Nevertheless, several steps in this approach could be supported by automation and the presented algorithms could help a choreography designer to accomplish his work.

5.3 Outlook

Several possible extensions to this work are suggested in the previous chapters. This section tries to summarize them and to provide an outlook. Future work might include to find an algorithm that automates the generation of strictly enforceable nets out of any enforceable net. While this challenge is nontrivial, it would enable the service-oriented solution to be usable for any enforceable interaction net without the necessity to add further implicit execution constraints. Until now the process-oriented approach lacks of a proof for comprehensive validity for all possible interaction nets. This proof will be necessary to base further work on it. A future prototype implementation of an execution environment for the suggested Petri net extensions and for the presented algorithms will ease further research work in this field. However, by providing the algorithms in this paper a basis was founded contributing the first step to close the gap between global and local interaction models.

References

[1] C.A. Petri: Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[2] J.M. Zaha, A. Barros, M. Dumas, A. ter Hofstede: Let's Dance: A Language for Service Behavior Modeling. Technical Report FIT-2006, Faculty of IT, Queensland University of Technology, http://servicechoreographies.com

[3] Interaction System Design Language. Architecture and Services of Network Applications Group at the University of Twente, http://isdl.ctit.utwente.nl

[4] Artem Polyvyanyy, Bridging Global and Local Interaction Models Using Petri Nets: Enforceability. Potsdam: Hasso-Plattner-Institute for IT Systems Engineering, to appear.

[5] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker: Service Interaction Modeling: Bridging Global and Local Views. In Proceedings of the 10th International EDOC Conference, Hong Kong, 2006

[6] Paolo Traverso et al., Supporting the Negotiation between Global and Local Business Requirements in Service Oriented Development. http://astroproject.org/

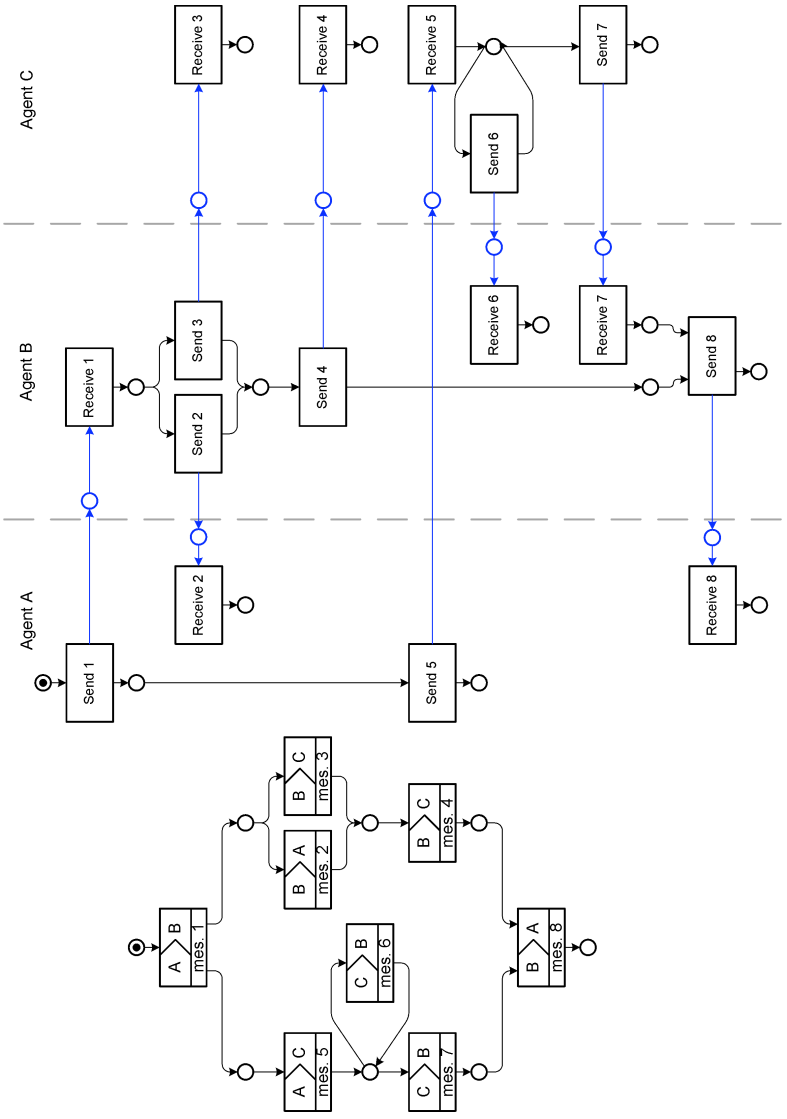

Fig. 3. A global interaction net an its service-oriented local interface processes

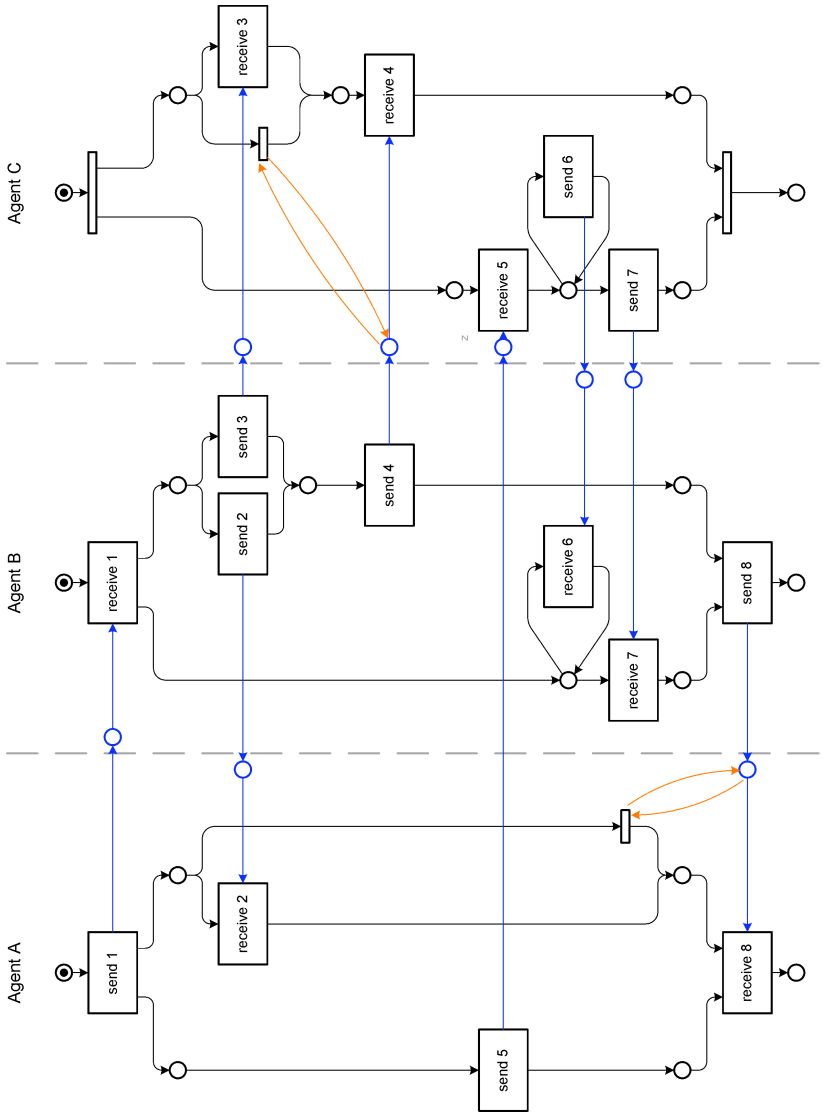

Fig. 4. Set of process-oriented local interface processes

Algorithm 1 Generating Service-Oriented Interface Processes

Given is an interaction Petri net $(P, T, F, M_0, R, \sigma, \rho)$,
the empty stets $\Phi, \Phi_n, P_n, T_n, F_n, M_{0,n}, T_S, T_R = \emptyset$
and the function $getOriginal: P_n \cup T_n \rightarrow P \cup T$

// Copy elements for each swimlane in new sets
for all $role \in R$ **do**
  for all $place \in P$ **do**
    $place_{role} := new()$
    $P_n := P_n \cup \{place_{role}\}$
    $M_{0,n}(place_{role}) := M_0(place)$
    $getOriginal(place_{role}) := place$
    $\Phi := \Phi \cup \{(place_{role}, role)\}$
  for all $transition \in T$ **do**
    if $\sigma(transition) = role \vee \rho(transition) = role$ **then**
      $transition_{role} := new()$
      // add executor relation
      $\Phi_n := \Phi_n \cup \{(transition_{role}, role)\}$
      $getOriginal(transition_{role}) := transition$
      if $\sigma(transition) = role$ **then**
        $T_S := T_S \cup \{transition_{role}\}$ // send transitions
      else if $\rho(transition) = role$ **then**
        $T_R := T_R \cup \{transition_{role}\}$ // receive transitions
  for all $p \in P_n, t \in T_n$, where $(getOriginal(p), getOriginal(t)) \in F \wedge (p, role) \in \Phi \wedge (t, role) \in \Phi_n$ **do**
    // do only add flows pointing to sends, not to receives
    if $t \in T_S$ **then**
      $F_n := F_n \cup \{(p, t)\}$
  for all $p \in P_n, t \in T_n$, where $(getOriginal(t); getOriginal(p)) \in F \wedge (p, role) \in \Phi \wedge (t, role) \in \Phi_n$ **do**
    $F_n := F_n \cup \{(t, p)\}$

// add communication places and find sends/receives
for all $(send, sender) \in \Phi_n$, where $send \in T_S \wedge \sigma(getOriginal(send)) = sender$ **do**
  for all $(receive, receiver) \in \Phi_n$, where $receive \in T_R \wedge \rho(getOriginal(receive)) = receiver$ **do**
    if $getOriginal(send) = getOriginal(receive) \wedge sender \neq receiver$ **then**
      $place_{send} := new()$
      $P_n := P_n \cup \{place_{send}\}$
      $F_n := F_n \cup \{(send, place_{send}), (place_{send}, receive)\}$
      $M_{0,n}(place_{send}) := 0$

$(P_n, T_n, F_n, M_{0,n}, R, \Phi_n)$ holds the elements of a set of local interface processes consistent with $(P, T, F, M_0, R, \sigma, \rho)$

Algorithm 2 Generating Process-Oriented Interface Processes

Given is an interaction Petri net $(P, T, F, M_0, R, \sigma, \rho)$,
the empty stets $\Phi, \Phi_n, P_n, T_n, F_n, M_{0,n}, T_S, T_R, T_N, P_I = \emptyset$
and the function $getOriginal: \ P_n \cup T_n \rightarrow P \cup T$

// Copy elements for each swimlane in new sets
for all $role \in R$ **do**
  for all $place \in P$ **do**
    $place_{role} := new()$
    $P_n := P_n \cup \{place_{role}\}$
    $M_{0,n}(place_{role}) := M_0(place)$
    $getOriginal(place_{role}) := place$
    $\Phi := \Phi \cup \{(place_{role}, role)\}$
  for all $transition \in T$ **do**
    $transition_{role} := new()$
    $T_n := T_n \cup \{transition_{role}\}$
    // add executor relation
    $\Phi_n := \Phi_n \cup \{(transition_{role}, role)\}$
    $getOriginal(transition_{role}) := transition$
    if $\sigma(transition) = role$ **then**
      $T_S := T_S \cup \{transition_{role}\}$ // send transitions
    else if $\rho(transition) = role$ **then**
      $T_R := T_R \cup \{transition_{role}\}$ // receive transitions
    else
      $T_N := T_N \cup \{transition\}$ // nop transitions
  for all $p \in P_n, t \in T_n$, where $(getOriginal(p), getOriginal(t)) \in F \wedge (p, role) \in \Phi \wedge (t, role) \in \Phi_n$ **do**
    $F_n := F_n \cup \{(p, t)\}$
  for all $p \in P_n, t \in T_n$, where $(getOriginal(t); getOriginal(p)) \in F \wedge (p, role) \in \Phi \wedge (t, role) \in \Phi_n$ **do**
    $F_n := F_n \cup \{(t, p)\}$

// add communication places and find sends/receives
for all $(send, sender) \in \Phi_n$, where $send \in T_S \wedge \sigma(getOriginal(send)) = sender$ **do**
  for all $(receive, receiver) \in \Phi_n$, where $receive \in T_R \wedge \rho(getOriginal(receive)) = receiver$ **do**
    if $getOriginal(send) = getOriginal(receive) \wedge sender \neq receiver$ **then**
      $place_{send} := new()$
      $P_n := P_n \cup \{place_{send}\}$
      $F_n := F_n \cup \{(send, place_{send}), (place_{send}, receive)\}$
      $M_{0,n}(place_{send}) := 0$
      $P_I := P_I \cup \{place_{send}\}$

addMilestoneEdges()

$(P_n, T_n, F_n, M_{0,n}, R, \Phi_n)$ holds the elements of a set of local interface processes consistent with $(P, T, F, M_0, R, \sigma, \rho)$

Algorithm 3 Adding milestone edges to the process-oriented interface processes

```
1: function addMilestoneEdges()
2: // for each nop transition
3: for all nop ∈ T_N do
4:    returnValue := addMilestoneRecursively(nop, nop)
5:
6:    // remove previous nops and their incoming and outgoing flow if milestones have
      been added.
7:    if returnValue = true then
8:       for all flow = (transition, place) ∈ F_n, transition = nop do
9:          F_n := F_n\{flow}
10:      for all flow = (place, transition) ∈ Fn, transition = nop do
11:         F_n := F_n\{flow}
12:      T_n := T_n\{nop}
13: end function
14:
15: // for simplicity this algorithm does not cover recursive loops. In an implementation
    they have to be avoided.
16: function addMilestoneRecursively(nop_1, nop_2)
17: returnValue = false
18: // for each place that follows nop_2
19: for all (t, place) ∈ F, t = nop_2 do
20:    // for each transition following that place
21:    for all (p, transition) ∈ F, p = place do
22:       // if transition is receive, add a milstone edge
23:       if transition ∈ T_R then
24:          addMilestoneEdge(nop_1, transition)
25:          returnValue := true
26:       // if transition is nop, follow the path recursively
27:       if transition ∈ T_N then
28:          r := addMilestoneRecursively(nop_1, transition)
29:          if r = true then
30:             returnValue := true
31: return returnValue
32: end function
33:
34: function addMilestoneEdge(nop, receive)
35: // get message place
36: for all (place, transition) ∈ F_n, transition = receive, place ∈ P_I do
37:    // duplicate place
38:    nop_receive := new()
39:    T := T ∪ {nop_receive}
40:    // add milestone edge
41:    F_n := F_n ∪ {(nop_receive, place), (place, nop_receive)}
42:    // duplicate existing flow
43:    for all (t, p) ∈ F_n, t = nop do
44:       F_n := F_n ∪ {(nop_receive, p)}
45:    for all (p, t) ∈ F_n, t = nop do
46:       F_n := F_n ∪ {(p, nop_receive)}
47: end function
```

Bridging Global and Local Interaction Models Using Petri Nets: Enforceability

Artem Polyvyanyy

Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam
D-14482 Potsdam, Germany
Artem.Polyvyanyy@student.hpi.uni-potsdam.de

Abstract. Service choreographies allow the derivation of new systems at a low cost. A conversational service consists of granular services that collaborate collectively through message passing interactions to reach a desired final state. A service choreography can be defined in a global view interaction model. Such a model captures all inter-service interaction constraints. Unfortunately, global models may capture behavioral constraints that can not be enforced locally – in local interaction models. These global models are referred to as locally unenforceable. In this paper we survey an enforceability property of global interaction models; in particular we aim at a formal definition of enforceability.

1. Introduction

Service Oriented Architecture (SOA) provides us with the view on the distributed system development. The system is designed as one composed of individual services run by individual businesses [1]. The potential of this approach is hidden in the numerous compositions that allow reuse of existing services. Further, services might collaborate collectively through message exchange to support inter-service data flows and the control flow of the combined service. Each service, in such an environment, can be seen as the entity exposing its public interface, including messages expected from and returned to the global environment. Such an interface might be a simple one-way message passing, request-response pattern or theoretically form patterns of any complexity. The service choreography is therefore a set of involved partner services with interface matching to denote message source and message destination services.

On a high abstraction level, an interaction model sets the constraints on the message exchange order between interaction participants: “A choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state.” [8]. In our work we assume that we already posses an interaction model, abstracting from its derivation method, and will solely concentrate on the study of its properties, in particular the enforceability property.

In an interaction (or choreography) model, interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services

[1]. Such a model subsumes the global knowledge about all possible conversations (choreography instances). We will refer to such a model as a global model. On the other hand, each party participating in the choreography supervises its own interaction behavior. This behavior can as well be captured in the interaction model. We will refer to such a model as a local model. Local models focus on the perspective of a particular service, capturing only those interactions that directly involve it. A possible usage scenario is one where global models are produced by analysts to agree on interaction scenarios from a global perspective, while local models are produced during system design and handed on to implementers of particular services [1].

Now, consider the execution of the conversational service. Service conversation should fit into the global interaction model. Conversational service execution is, thus, an execution of individual services with constraints on inter-service interactions ordering. However, it turns out that not all global models can be mapped into local ones in such a way that the resulting local models satisfy the following two conditions: (i) they contain only interactions described in the global model; and (ii) they are able to collectively enforce all the constraints expressed in the global model. In case when these conditions do not hold, this means, that in order to ensure all the constraints of the global model there should exist an external component that tracks execution for conformance with the global interaction model.

Two conditions presented in a previous passage subdivide a set of all global interaction models into two classes – those for which these conditions hold, and those for which they do not. Fulfillment of these two conditions would mean that execution of such choreography can be reduced to simultaneous execution of the participating services. Such global interaction models are called locally enforceable, as participant services can locally enforce all the constraints of the global model through the constraints expressed in their local models.

In this paper we will study the local enforceability property of global interaction models. In particular, we will derive the formal definition of local enforceability. In order to complete this task we will first define an abstract syntax for interaction models. We will then use this syntax to reason on the enforceability property. Two approaches to automated local model composition for global model derivation will be presented, assuming synchronous and asynchronous interactions.

2. Interaction Petri Net

In this section we will present an abstract syntax suited for interaction modeling. Several interaction modeling languages already exist, among them ISDL [6], WS-CDL [3] or Let's dance [5]. However, we would like interaction modeling language to have its equivalent mathematical representation to allow formal statements about interaction model properties.

As a starting point we will informally define an abstract interaction model. An interaction model consists of a set of interrelated service interactions corresponding to message exchanges. At the lowest level of abstraction, an interaction is composed of a message sending action and a message receipt action (referred to as communication

actions). Message content is regulated by a message type. A communication action is performed by an actor playing a role.

As the next step we can extract the following choreography modeling concepts as interaction, interaction participant role, exchange message type and concept associations like interaction sender role, interaction recipient role, type of the message exchanged in a single interaction.

A Petri net [7] (also known as a place/transition net or P/T net) is the mathematical representations of discrete distributed systems. Petri nets are well suited for modeling the concurrent behavior of distributed systems, which is exactly the environment of service interactions. Thus, we would like to extend classical Petri nets in such a way to incorporate interaction modeling concepts and associations. Numerous Petri net extensions exist: colored Petri nets, time and hierarchy extension. We would like to derive our own extension – an interaction Petri net (IPN).

A classical Petri net is a quadruple (P,T,F,M), where:

- P, is a finite set of *places*;
- T, is a finite set of *transitions* $(P \cap T = \emptyset)$;
- F, is a set of arcs known as a *flow relation*. The set F is subject to the constraint that no arc may connect two places or two transitions – $F \subseteq (P \times T) \cup (T \times P)$;
- $M : P \to \mathrm{N}$, is an *initial marking*, where for each place $p \in P$, there are $n_p \in \mathrm{N}$ tokens.

With all the add-ons an interaction Petri net becomes a nonuple $(P, I, F, R, T, M, \sigma, \rho, \tau)$, where:

- P, is a finite set of *places*;
- I, is a finite set of *interaction occurrences*, analogue of classical Petri net transitions $(P \cap I = \emptyset)$;
- F, is a set of arcs known as a *flow relation* – $F \subseteq (P \times I) \cup (I \times P)$;
- R, is a finite set of participant roles;
- T, is a finite set of message types;
- $M : P \to \mathrm{N}$, is an *initial marking*;
- $\sigma : I \to R$, is a send function, which assigns each interaction occurrence $i \in I$ a sender participant role $r \in R$;
- $\rho : I \to R$, is a receive function, which assigns each interaction occurrence $i \in I$ a receiver participant role $r \in R$;
- $\tau : I \to T$, is a message type assignment function, which assigns each interaction occurrence $i \in I$ a message type $t \in T$ of the message been exchanged.

Now, in the IPN, transitions are replaced by interaction occurrences. In order to be able to define sender, receiver and message type used for every single interaction the corresponding functions σ, ρ and τ are used.

If for $(i_1, i_2) \in I \times I,\ i_1 \neq i_2$ holds that $\sigma(i_1) = \sigma(i_2)$ – sender roles are equal, $\rho(i_1) = \rho(i_2)$ – receiver roles are equal and $\tau(i_1) = \tau(i_2)$ – exchanged message types are equal, then the two interaction occurrences i_1 and i_2 belong to the same

interaction model. Two interaction instances, i.e. two concrete message exchanges, are related to an interaction model by considering the sender, receiver and message type. However, interaction instances can only be related to one out of a set of interaction occurrences by also considering the relationship to other interaction instances.

The dynamic behavior of the IPN is modeled similar as in Petri net – by tokens. Places may contain tokens, which may move by firing transitions (interactions). Interactions can only fire if they are enabled. An interaction is enabled if each of the input places contains tokens. An enabled interaction may fire (take place) and thereby consumes the tokens from the input places and produces tokens for the output places. A conversation finishes when no more enabled interactions exist.

3. Automated Composition of Local Interaction Models

IPNs suit well for defining both – local and global interaction models. In a common usage scenario each granular service possesses its own local interaction model. It is explicitly present in the service implementation and can be seen as the service interface constraint model. Further, in this paper, we will refer to a sample set of local interaction models $B = \{b_1, \ldots, b_4\}$ shown in Figure 1.

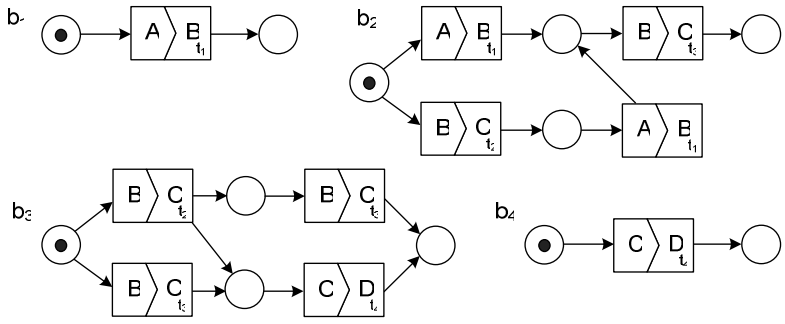

Fig. 1. Sample set of local interaction models

Here, we have introduced a graphical notation for the interaction models. Communication actions are represented by non-regular pentagons that are juxtaposed to form a rectangle denoting a single interaction. From left to right first comes the send action, then the receive action. A role of an actor performing a communication action is noted inside of a pentagon. The message type exchanged is noted in the right bottom corner of the interaction rectangle. In case of local interaction models an additional information on participant role for which this interaction model is constructed is required.

Let $b = (P_b, I_b, F_b, R_b, T_b, M_b, \sigma_b, \rho_b, \tau_b) \in B$, be a local model. Then function $\mu : B \to R$ assigns local model b a participant role $r \in R$ for which this interaction model is constructed. $\mu(b) = r \Rightarrow \forall i \in I_b : \sigma_b(i) = r \vee \rho_b(i) = r$. We will extend our

sample from Figure 1 by defining μ function for each local model from set B: $\mu(b_1) = A$, $\mu(b_2) = B$, $\mu(b_3) = C$ and $\mu(b_4) = D$.

At the same time, a global model is constructed abstracting from existing local models. It is possible to take into consideration existing local models while modeling the global model, but the primary force that drives the process is the requirement to achieve a final state. You, as a modeler of a global model, define interactions and their ordering constraints that should bring you to the final state. Let us consider a global model shown in Figure 2 that consists of interactions present in the set of local models – B.

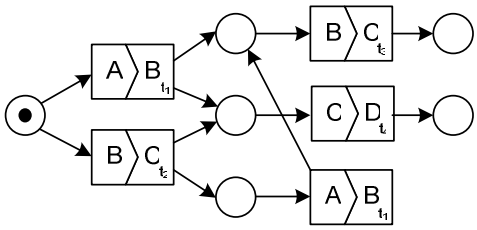

Fig. 2. Sample global interaction model

It is now a question whether this global model is enforceable. What makes this model challenging is the conversation scenario that starts with interaction between actors in roles A and B. Though, afterwards the interaction between actors in roles C and D is enabled, locally this becomes clear in the local model for role C after interaction between actors in roles B and C with the message type t_3 will happen. However, in order to conclude that this model is enforceable we should be able to map this global model on a set of local models so that all the global constraints will be locally preserved. Let us assume that such set of local models is the one presented in Figure 1. The question now becomes how to ensure that all the constraints are locally preserved in the proposed set. In order to be able to answer this question we propose to first take a look into automated local model composition. In a proposed setup it is possible to derive mechanisms for deterministic local models composition. Afterwards, the composed model can be checked to ensure that it preserves all the constraints of the global model.

Further, we will propose two approaches for automated local interaction models composition.

3.1. Global Asynchronous Interaction Model

An interaction is a kind of distributed action that occurs as two or more objects having an effect upon one another. It is therefore intuitive to model interaction as asynchronous event of sending and receiving a message action. Interactions might happen between two services of some role with matching interfaces (sender, receiver roles and message type used for interaction must match). A straight forward solution to local models composition is joining matching interfaces through an intermediate place which denotes a state of a message sent by the sender but not yet received by

the recipient. The visualization of such a composition for local models b_1 and b_2 from our sample set B is shown in Figure 3.

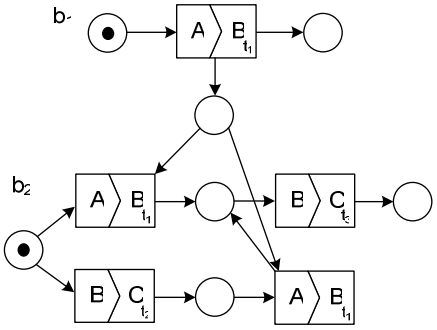

Fig. 3. Sample asynchronous local interaction models composition

Additional places should be created for each matching interface interaction occurrences. Also, additional flow relation from all communicating interaction occurrences of the sender local model role to the additional place, and from the additional place to all communicating interaction occurrences of the receiver local model role should be constructed. This procedure can be formalized and can be given in a form of an algorithm.

For $\{b_1,\ldots,b_n\}, n \in \mathrm{N}^+$, where $b_i = (P_i, I_i, F_i, R_i, T_i, M_i, \sigma_i, \rho_i, \tau_i)$ is a local IPN such that:

- $\forall i, j \in 1,\ldots,n,\ i \neq j : P_i \cap P_j = \emptyset$;
- $\forall i, j \in 1,\ldots,n,\ i \neq j : I_i \cap I_j = \emptyset$.

Then a global asynchronous interaction model $b = (P, I, F, R, T, M, \sigma, \rho, \tau)$ can be obtained by following formal steps:

- $I = \left(\bigcup_{i=1}^{n} I_i \right)$, analogously for $R, T, M, \sigma, \rho, \tau$;
- $map : P_A \to T \times R \times R$;
- for each $(t, s, r) \in T \times R \times R$ such that

$$\exists i \in I : \tau(i) = t \wedge \sigma(i) = s \wedge \rho(i) = r$$
$$p := newPlace;$$
$$P_A = P_A \cup \{p\};$$
$$map = map \cup \{(p, (t, s, r))\};$$

- $P = \left(\bigcup_{i=1}^{n} P_i \right) \cup P_A$;

- $F_A = \{(i,p) \in I \times P_A \mid \exists r \in R : i \in I_k \wedge r = \sigma(i) = \mu(b_k) \wedge$
  $\wedge map(p) = (\tau(i), \sigma(i), \rho(i))\} \cup \{(p,i) \in P_A \times I \mid \exists r \in R : i \in I_k \wedge$
  $\wedge r = \rho(i) = \mu(b_k) \wedge map(p) = (\tau(i), \sigma(i), \rho(i))\}$;
- $F = \left(\bigcup_{i=1}^{n} F_i \right) \cup F_A$.

By applying these steps one would derive a global asynchronous interaction model. Here, we have introduced the additional *map* relation in order to be able to store information required for later additional flow relation construction.

One might use a regular Petri net semantics of transitions enabling and firing (in our case interactions). However, this would mean that interactions are not atomic. It is possible that interaction send communication action has happened, but the next action would not be the corresponding receive communication action. However, for some applications this might be the desired behavior.

On the other hand, a dynamic behavior borrowed from Petri nets of such asynchronous model can be changed to ensure that interactions happen atomically. To ensure interaction atomicity the interaction enabling rule must be modified. Interaction i is enabled iff:

1. $\forall p \in {\bullet i} : M(p) > 0$;
2. $\exists i_m \in I \, \forall \, p \in {\bullet i_m} \setminus p^* : M(p) > 0, \tau(i) = \tau(i_m), \sigma(i) = \sigma(i_m), \rho(i) = \rho(i_m), p^* \in i\bullet$.

Here, ${\bullet i}$ denotes a set of input places for an interaction i, respectively $i\bullet$ is a set of an interaction output places. Interactions i and i_m can be referred to as enabled partner interactions. For enabled pair of i and i_m interactions fire in a sequence, first i, then i_m:

1. $\forall p \in {\bullet i} : M(p) = M(p) - 1 \wedge \forall p \in i\bullet : M(p) = M(p) + 1$;
2. $\forall p \in {\bullet i_m} : M(p) = M(p) - 1 \wedge \forall p \in i_m\bullet : M(p) = M(p) + 1$.

3.2. Global Synchronous Interaction Model

We have already started looking at interactions as synchronous activities. We have presented interaction enabling and interaction firing rules to ensure atomicity of single interactions, i.e. no other actions are possible between single interaction send and receive activities. A global model can be composed to assume synchronous nature of separate interactions. The visualization of such a composition for local models b_3 and b_4 from our sample set B is shown in Figure 4.

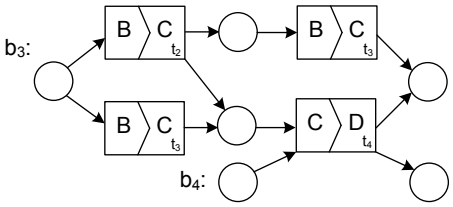

Fig. 4. Sample synchronous local interaction models composition

Corresponding interaction occurrences over local models set are merged. This procedure can be formalized and can be given in a form of an algorithm.

For $\{b_1,\ldots,b_n\}, n \in \mathrm{N}^+$, where $b_i = (P_i, I_i, F_i, R_i, T_i, M_i, \sigma_i, \rho_i, \tau_i)$ is a local IPN such that:

- $\forall i, j \in 1,\ldots,n,\ i \neq j : P_i \cap P_j = \emptyset$;
- $\forall i, j \in 1,\ldots,n,\ i \neq j : I_i \cap I_j = \emptyset$.

Then a global synchronous interaction model $b = (P, I, F, R, T, M, \sigma, \rho, \tau)$ can be obtained by following formal steps:

- $P = \left(\bigcup_{i=1}^{n} P_i \right)$, analogously for R, T, M ;
- $I = \{ i_{k,m} \mid \exists i_k \in I_K \ \exists i_m \in I_M : K \neq M, \tau(i_k) = \tau(i_m), \sigma(i_k) = \sigma(i_m), \rho(i_k) = \rho(i_m), \mu(b_K) = \sigma(i_k), \mu(b_M) = \rho(i_m) \}$;
- $\sigma = \{ (i_{k,m}, r), i_{k,m} \in I \mid \exists i_k \in I_K : \sigma(i_k) = r \vee \exists i_m \in I_M : \sigma(i_m) = r \}$, analogously for ρ, τ ;
- $F = \{ (p, i_{k,m}), i_{k,m} \in I \mid \exists (p, i_k) \in F_K \vee \exists (p, i_m) \in F_K, K \in 1,\ldots,n \} \cup \{ (i_{k,m}, p), i_{k,m} \in I \mid \exists (i_k, p) \in F_K \vee \exists (i_m, p) \in F_K, K \in 1,\ldots,n \}$.

By performing these steps one is able to derive a global synchronous interaction model. Each interaction in such a model is an indivisible concept which assumes sending and receiving a message as an atomic phenomenon.

4. Enforceability

Now, we can define enforceability, as the property of a global interaction model for which there exists a set of local models which will preserve all the constraints of a global model in the automated composition of the local models from this set.

A global interaction model $b = (P, I, F, R, T, M, \sigma, \rho, \tau)$ is locally enforceable iff there exists $B = \{b_1,\ldots,b_n\}, n \in \mathrm{N}^+$ – a set of local interaction models that includes all interactions from b, such that b and the global synchronous interaction model constructed from B are in simulation preorder relation, in the sense that b simulates

the automated composition of the local models set with a constraint that whenever automated composition reaches the final state, *b* also reaches the final state.

In theoretical computer science a simulation preorder is a relation between state transition systems associating systems which behave in the same way in the sense that one system simulates the other. Intuitively, a system simulates another system if it can match all of its moves [2].

An interaction model state transition system consists of interaction firing transitions and states that denote interaction firing sequence starting from the initial state. In the proposed definition of enforceability one might incorporate global asynchronous interaction model with state transition system assuming modified interaction firing semantics that preserves atomicity of interactions.

In Figure 5 we illustrate the global synchronous interaction model constructed from our sample local models set provided in Figure 1.

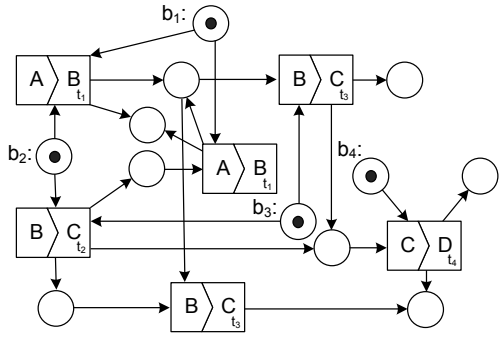

Fig. 5. Global synchronous interaction model for sample local models set from Figure 1

Here, the input places for local models are correspondently marked. Subsequently, in Figure 6 you can find state transition systems for both, the initial global model (Figure 2) and the automatically derived global model (Figure 5).

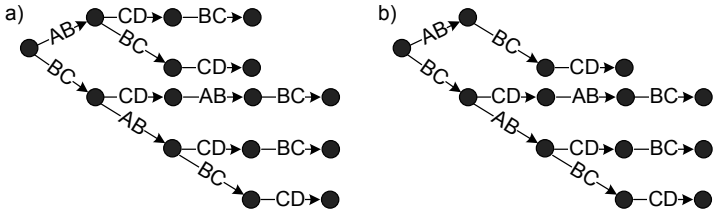

Fig. 6. State transition systems: a) for the global model – Figure 2, b) for the global synchronous interaction model – Figure 5

In Figure 6 it is noticeable that system *b* is more restrictive as compared to system *a*. The additional restriction comes from the local model for actor in role C not knowing that interaction between actors in roles C and D is enabled before the interaction between actors in roles B and C has happened. However, system *a* simulates system *b* with additional constraint that once system *b* reaches its final state, system *a* is also in its final state. Finally, system *b* contains all the interactions that

appear in *a*. Thus, we conclude that the initial global model proposed in Figure 2 is enforceable.

5. Conclusions

In this paper we have done a survey on the enforceability property of global interaction models. We have provided the extended Petri net notation (IPN) suited for choreography modeling. We have developed two approaches for automated local interaction models composition. Finally, we have proposed the formal definition of the enforceability property for global interaction models.

The fact that we have used Petri net as the basis for interaction modeling allowed us to reuse the formal mathematical model of Petri nets and to perform further reasoning on it. Also, it is possible to perform analysis of Petri nets properties on interaction Petri nets. These are reachability, liveness and boundedness properties.

Enforceability was already studied in [1]. In this work the authors define an algorithm for determining if a global model is locally enforceable and an algorithm for generating local models from global ones. However, this work is based on the verbal definition of enforceability. Now, we have closed this gap.

As a complement to the work reported in this paper a formal definition of the simulation, with the constraints applied while enforceability definition, might be derived. Probably, a simulation variant might be defined to suit our needs. Also, an investigation of IPN ability to represent common interaction patterns [4] is of great interest.

6. References

1. J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker: Service Interaction Modeling: Bridging Global and Local Views. In Proceedings of the 10th International EDOC Conference, Hong Kong, 2006
2. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM, 43(3):555-600, May 1996
3. N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, November 2005. http://www.w3.org/TR/ws-cdl-10
4. A. Barros, M. Dumas, and A. H.M. ter Hofstede: Service Interactions Patterns. In Proceedings of the 3rd International Conference on Business Process Management (BPM), Nancy, France, September 2005. Springer Verlag, pp. 302-218
5. J.M. Zaha, A. Barros, M. Dumas, A. ter Hofstede: Let's Dance: A Language for Service Behavior Modeling. Technical Report FIT-2006, Faculty of IT, Queensland University of Technology, 2006. http://eprints.qut.edu.au/archive/00004468/
6. D.A.C. Quartel, R.M. Dijkman, M. van Sinderen: Methodological support for service-oriented design with ISDL. In Proceedings of the 2nd Intenational Conference on Service-Oriented Computing (ICSOC), New York NY, USA, November 2004, pp 1–10, Springer Verlag
7. C.A. Petri. Kommunikation mit Automaten. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962
8. W3C Web Services Glossary, 2004, http://www.w3.org/TR/ws-gloss/

A Compatibility Notion based on Desired Interactions

Matthias Weidlich

Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
matthias.weidlich@hpi.uni-potsdam.de

Abstract. Current approaches used to determine compatibility for compositions of business processes neglect the aspect, whether the interconnected processes are able to potentially attain their goals. Based on the assumption, that these goals are related to some interactions, we provide a new compatibility notion. Therefore this paper introduces *desired interactions compatibility* which adds reachability of certain interactions to interaction soundness. In addition to the definition of our compatibility notion in π-calculus, we discuss its application to operating guidelines.

1 Introduction

Together with the increasing influence of the service-oriented architecture (SOA), a growing demand for compatibility notions can be determined, as any process integration relies on them. Applied during design time, these notions ensure the correctness of process compositions. Additionally, the question whether service providers and service requesters can interact successfully has high relevance for service discovery. For a specific service request, the service broker has to select these services from a repository that are guaranteed to interact properly with the requester.

Various approaches have been presented to check structural and behavioral compatibility. Nevertheless, criteria for the evaluation of the success of service interactions have often been limited to the freedom of live- and deadlocks or the prevention of unanticipated messages. This paper argues, that this is not sufficient as every interaction is executed to attain a certain goal. This requirement derives from the business process definition as *"a set of one or more linked procedures or activities which collectively realize a business objective or policy goal..."*[1]. Hence, a compatibility notion should also take the potential achievement of this goal into consideration. Therefore, we provide a new compatibility notion, ensuring that interconnected processes can achieve a certain goal.

A main question is how goals can be formalized. One possibility would be the definition of global desired termination states. In these states all processes of the composition have to be in one of their local desired termination states. This idea raises the question how services without explicit end states should be

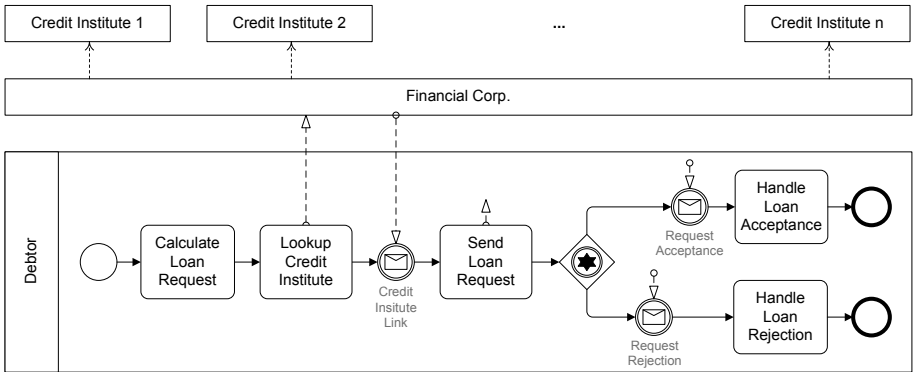

Fig. 1. Loan request scenario

treated (for instance a service provider is reset to an inital state after responding a certain request). However, this paper focus on another possibility to define goals, namely desired interactions. Thus, we assume that the goal of a process in a composition is always directly related to some interactions. Consequently, processes are compatible in regard to their goals, if all desired interactions can potentially occur.

The next section discusses the context of interconnected processes by means of an example and section 3 summarizes related work. In section 4 we define our compatibility notion using the π-calculus, while we propagate our ideas to operating guidelines in section 5. The paper concludes with a discussion of the achievements.

2 Interconnected Processes

To introduce the topic, we present an example from the financial domain, namely a loan request. Figure 1 illustrates the example using a slightly extended variant of the Business Process Modeling Notation (BPMN). At first the debtor calculates the loan amount and sends this information to a financial corporation. This corporation determines the credit institute offering the best conditions and sends a link to it back to the debtor. Subsequently, the debtor sends a loan request to the selected credit institute and receives either an acceptance or a rejection of his request. This example comprises five interactions: credit institute lookup, credit institute link transmission, loan request, rejection response and acceptance response, whereby the latter is the desired response from the debtor's view. Thus the exchange of the acceptance message is the desired interaction in our example. Hence any service implementation of the credit institute, which always rejects loan requests (e.g., because it does not offer any loan transactions), must not be compatible with a debtor service implementation. In contrast, a credit institute implementation, at least sometimes sending acceptance messages or perhaps not

even capable to send rejection messages, is compatible, as the debtor achieves his goal and takes out a loan. Therefore, an implementation of the credit institute must not be able to send acceptance as well as rejection messages.

Service compositions respecting their desired interactions can contain deadlocks, livelocks and dead activities. Although there may be some reasons to allow these effects (e.g. a deadlock of the credit institute service is solved at some higher abstraction level and thus is irrelevant, if the desired interaction already occurred), we want to prohibit them. The motivation behind is driven by the interpretation of non-desired interactions as treatement of negative responses (as in the introduced example), fall-back or error handling processes. Although their occurrence is not necessary to attain a certain goal, they should lead to valid process states. This is ensured by applying common compatiblity notions, that prove the subprocesses in the composition to be free of deadlocks and livelocks.

3 Related Work

Related work comprises various compatibility notions published in recent years. Martens [2] defined *weak soundness* for Petri nets ensuring that the interconnected processes are free of deadlocks and livelocks. Canal et al. [3] introduced another compatibility notion for π-processes, overcoming the limitations of Petri nets regarding dynamic binding. Nevertheless, this notion is specified for bilateral communication only. Puhlmann et al. [4] introduced *interaction soundness* for the π-calculus. It is based on *lazy soundness* and proves processes to be free of deadlocks and livelocks, while allowing lazy activities. However, process goals are not considered. A remarkable approach has been published by Dijkman and Dumas attempting to overcome goal incompatibilities [5]. Nonetheless, they require a comprehensive choreography description and the procedure is also limited to Petri nets. Among other types of structural compatibility, Decker presents *minimal structural compatibility* in [6], even though a formal definition is missing. The operating guideline approach [7–9] suggests a different point of view, since all valid interaction behavior respecting the *well-communicating* criterion is specified. Currently, this approach is limited to acyclic Petri nets.

4 Formalization of the Compatibility Notion in π-Calculus

This section describes our compatibility notion based on desired interactions. At first, the π-calculus, which can be used for description and analysis of interacting processes, is introduced. Due to its link passing capability, the π-calculus is predestined to model dynamic binding in the SOA domain (an extended motivation can be found in [10]). Besides, the potential utilization of simulation techniques to prove compatibility, argues for the application of π-calculus as our formal foundation. The second part summarizes process requirements related to

structural compatibility and interaction soundness. In the third part, we use the π-calculus to formally define the notion in respect to desired interactions.

4.1 Prerequisits: The π-Calculus

The π-calculus is a process algebra developed to describe and analyze concurrent, interacting processes in a formal way. It is based on names representing the communication channels as well as the messages sent over them. Hence communication channels can be passed to other processes to support link passing mobility, the π-calculus is particularly suited to describe systems with dynamic or evolving structures.

The grammar of the π-calculus is defined in the Backus normal form as follows:

$$\begin{aligned} P &::= M \mid P|P' \mid \mathbf{v}zP \mid !P \mid K(y_1, \dots y_n) \\ M &::= \mathbf{0} \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi. \end{aligned}$$

Informally spoken, the process semantic can be characterized as follows: The concurrent execution of two processes P and P' is denoted by $P|P'$. $C = \prod_{i=1}^{n} P_i$ is the short form for the definition of the composition C as the parallel execution of n processes P_i and $P \in C$ denotes that P is a concurrent subprocess of C. In the process $\mathbf{v}zP$, the operator $\mathbf{v}$ restricts the name z to P. The meaning of the process replication, given by $!P$, is that an infinite number of replicated process instances are acting in parallel. Recursion for P with the parameters $y_1, \dots y_n$ is expressed via $P(y_1, \dots y_n)$. The inaction $\mathbf{0}$ represents empty or inactive behavior. Further on, the summation operator specifies alternatives, as $M + M'$ evolves to M or M'. All actions a process can do, are given by π. To model interactions, input and output prefixes are used. The output prefix $\overline{x}\langle\tilde{y}\rangle.P$ evolves to P after sending a sequence of names $\tilde{y}$ over the channel identified with x. The corresponding action is the input prefix $x(\tilde{z}).P$. This process receives a sequence of names and continues as $P\{^{\tilde{m}}/_{\tilde{z}}\}$ with all occurrences of $\tilde{z}$ replaced by the received names. The corresponding input action to the output action $\overline{x}$ is also denoted as $\overline{\overline{x}}$, which is semantically identical to x. Additionally, τ, the silent transition, models an internal action, which cannot be observed at all. The match prefix $[x = y]\pi.P$ is defined in the expected way: the process behaves as $\pi.P$, iff x and y are identical, and like $\mathbf{0}$ otherwise.

A complete formal definition of the π-calculus semantics based on a labeled transition system can be found in [11]. Nevertheless we introduce some of the reasoning concepts, as they are of high relevance for this paper.

In the π-calculus the name x is *bound* inside a process P by binding operators, either $y(x)$ or $\mathbf{v}xP$. Names that are not bound at all, are in the set of *free* names denoted by $fn(P)$. In contrast to bound names, free names can be accessed from processes outside of P. On account of this, they are used to model interactions and can be observed for simulation-based reasoning. Furthermore a *transition sequence* $P \xrightarrow{\alpha} P'$ is a sequence of interactions or unobservable actions, with α

describing the actions to transform P to P'. An action x of the process P is reachable, if there exists a transition sequence $P \xrightarrow{\alpha} P' \xrightarrow{x} P''$.

Based thereon, *simulation* is defined as a binary relation $\mathcal{R}$ on two processes: $P\mathcal{R}Q \wedge P \xrightarrow{\alpha} P' \Rightarrow \exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$. Informally, *weak simulation* is a simulation focusing the observable behavior regarding free names while abstracting from internal actions. Furthermore, an *open simulation* is a simulation, that is preserved by all name substitutions (please refer to [11–13] for details).

4.2 Structural Compatibility and Interaction Soundness

As already mentioned, we require processes not to be able to receive all messages potentially sent. Following the arguementation in [6], we demand at least one potential interaction between two processes. Thus, every process P_i participates in at least one potential interaction with one of the other processes of $SYS = \prod_{i=1}^{n} P_i$. In the following, we require at least one static link between two processes. This is the case, when one name exists in the set of actions, for which a corresponding action is in the set of actions of another process of SYS. With A_{P_i} as the set of actions for a process P_i, which can comprise any name x of a channel that is used in an input (output) prefix $x(\tilde{y})$ $(\overline{x}\langle\tilde{y}\rangle)$, we formalize minimal structural compatibility for $SYS = \prod_{i=1}^{n} P_i$ as follows (please note, that $x \in A_{P_i}$ identifies one element of the set and not necessarily an input action):

$$\forall P_i \in SYS : \exists x \in A_{P_i} : \overline{x} \in A_{P_j} \wedge i \neq j.$$

Due to the link passing capability of the π-calculus, we can imagine a process composition $SYS = (A|B|C) = (\overline{a}\langle c\rangle.\mathbf{0}|a(b).\overline{b}\langle\rangle.\mathbf{0}|c().\mathbf{0})$, in which every process communicates with another process, but the introduced criterion is not fulfilled. That derives from the dynamic binding, as a link to process C is at first passed from A to B, which interacts with C afterwards. Therefore, a formalization of minimal structural compatibility respecting potential dynamic binding in the π-calculus would have to take two aspects into account: on the one hand, one has to regard the knowledge about free names of input actions of processes without static link to another process (in the introduced example A has this knowledge about C in form of the name c). On the other hand, the potential propagation of this knowledge to processes that can send on bound names (in the example A propagates the knowledge to B, which can send on a bound name) has to be considered.

It was already mentioned, that the reachablility of desired interactions does not ensure the correctness of process compositions. To prove the processes to be free of deadlock and livelocks, a common compatiblity notion has to be applied. Therefore, we prove all subprocesses of the composition to be interaction sound. Every subprocess is treated separately and its interaction soundness is decided regarding an environment built of all remaining subprocesses of the composition. Interaction soundness guarantees the subprocesses to be free of deadlocks in the context of the process composition. Interaction soundness has been chosen because it is grounded on lazy soundness, and thus allows activities to be become

active after the final activity has been reached. Interaction soundness is defined for a process P in an environment E. It requires the unification, denoted as $P \uplus E$, to be lazy sound. Thus the final activity of $P \uplus E$ is semantically reachable from every activity reachable from the initial activity, until the final activity has been reached for the first time. In addition, it is required that the final activity is reached exactly once.

In regard to the process composition $SYS = \prod_{j=1}^{n} P_j$, we have to decide interaction soundness for every subprocess P_j. The environment E_{P_k} for a certain subprocess P_k emerges from SYS by removing P_k, resulting in $E_{P_k} = (\prod_{j=1}^{k-1} P_j | \prod_{j=k+1}^{n} P_j)$. The system consisting of P_k and E_{P_k}, denoted as $SYS_{P_k} = (P_k | E_{P_k})$, is enhanced with the free name i for the initial activity and the free name $\overline{o}$ for the final activity. While the enchancement with the name i can be done straightforward, the enhancement of the final activity might require an extensive restructuring of the process. This task has to be done potentially in a manual way, depending on the algorithms used to derive the π-calculus representation of the process. The derived annotated system, denoted as $ASYS_{P_k}$, is then checked for weak open bisimulation equivalence with $S_{LAZY} = i.\tau.\overline{o}.\mathbf{0}$. We formalize the fact, that all subprocesses of the composition SYS have to be interaction sound, as follows:

$$\forall P_j \in SYS: \; ASYS_{P_j} \approx^{O}_{i,o} S_{LAZY}.$$

4.3 Desired Interactions in π-Calculus Processes

Based on the above-named requirements, we extend the known compatibility notions by taking desired interactions into consideration. In this context desired interactions are interactions between two or more processes, of which the potential reachability should be guaranteed. They are needed for some processes to attain their policy goals, which are defined independently of the process end states. Thus, the set of desired actions DA_P for a single process P, can comprise any *free* name x of a channel that is used in an input (output) prefix $x(\tilde{y})$ $(\overline{x}\langle\tilde{y}\rangle)$ contained in P. In other words, any send (receive) action on a channel with the *free* name $x \in DA_P$ is a desired action of P. The requirement for *free* names originates from the essential observability of the according action. Without this demand we could consider an example process $P = (lookup(z).\overline{z}\langle\rangle.\mathbf{0})$ with $DA_P = \{z\}$, which leads to the following problem: the desired action cannot be identified due to the dynamic binding of the name of the corresponding channel.

The set of desired actions of a process composition $SYS = \prod_{i=1}^{n} P_i$, comprising n processes P_i, is denoted by DA_{SYS}. It is defined as the union of sets of desired actions DA_{P_i} regarding all subprocesses. Hence, DA_{SYS} is not a multiset, every element is unique. Subsequently, these desired actions identify the set of desired interactions DI_{SYS}, that contains all desired actions and their corresponding actions. Thus, a desired interaction is a pair of an input and an output action:

$$x \in DA_{SYS} \Rightarrow x \in DI_{SYS} \wedge \overline{x} \in DI_{SYS}.$$

The reachability of these desired interactions extends the existing compatibility notions and leads to the following definition of desired interaction compatibility. The processes in a process composition $SYS = \prod_{i=1}^{n} P_i$ are *desired interaction compatible*, if and only if:

1. the proces composition SYS is minimal structural compatible,
2. every process P_i is interaction sound (in regard to the other processes acting as an environment) and
3. all desired interactions DI_{SYS} are reachable.

Algorithms have been presented to prove the first and the second criterion, consequently we focus on the third constraint. The reachability of the desired interactions is decided using simulation techniques. The idea behind is to observe whether a desired interaction occurs during the execution of the process composition. Therefore, we define a process $I_x = (x|\overline{x})$ executing the two actions x and $\overline{x}$, which compose the according interaction, in parallel. The interaction is reachable during the process evolvement, iff the process composition weakly simulates the process I_x provided that all other free names unequal to x have been restricted to SYS. We apply weak simulation due to the abstraction of all internal communication (evolving to silent transitions) within the composition. Hence, all free names, which should not be observed, are restricted to the process composition. Consequently, their related actions are treated as internal communication. Thus, we formalize this requirement for all desired interactions DI_{SYS} of a process composition $SYS = \prod_{i=1}^{n} P_i$ as follows:

$$\forall x \in DI_{SYS} : \ (\mathbf{v}y|y \in fn(SYS) \setminus \{x, \overline{x}\})(SYS) \precsim^{O} I_x \quad \text{with} \quad I_x = (x|\overline{x}).$$

The necessity to consider each interaction separately originates from the arbitrary order of the potential interactions. Thus, we prove reachability for a single interaction and not for all conceivable combinations of desired interactions. Consider an example composition consisting of the three processes $A = s.\tau.e.\mathbf{0}$ and $B1 = B2 = \overline{s}.\mathbf{0} + \overline{e}.\mathbf{0}$ with $DI_{(A|B1|B2)} = \{s, \overline{s}, e, \overline{e}\}$. The processes in this composition interact successfully, if we can observe the interactions on the two channels s and e. However, we do not require the processes to participate in these interactions in every conceivable combination, as the order is determined by the process A, which receives e *after* s. Consequently, this process composition is not able to simulate $I_{s,e} = (s|\overline{s}|e|\overline{e})$, in which the actions are not restricted to any order. In contrast, the composition simulates both $I_s = (s|\overline{s})$ and $I_e = (e|\overline{e})$, so that the order of these interactions can be restricted in the composition. Additionally, $I_{s,e}$ would require that both s and e occur, which might not always be the case. Imagine $A = s.\mathbf{0} + e.\mathbf{0}$ as a new definition of the process A. The desired interactions are mutually exclusive, hence the system containing the new definition of A cannot simulate $I_{s,e}$, which requires the occurence of both actions s and e.

Finally, we want to apply desired interaction compatibility to the example process illustrated in figure 1. Therefore, we define the debtor ($DEBT$), the

financial corporation ($FICORP$) and two credit institutes (CRE_1 and CRE_2) as follows:

$$\begin{aligned} DEBT(lu, acc, rej) &= (\mathbf{v}c)(\tau.\overline{lu}\langle c\rangle.c(ci).\overline{ci}\langle acc, rej\rangle.(acc.\tau.\mathbf{0} + rej.\tau.\mathbf{0})) \\ FICORP(lu, cri) &= (\mathbf{v}x)(lu(x).\overline{x}\langle cri\rangle.FICORP(lu, cri)) \\ CRE_1(x) &= (\mathbf{v}a, r)(x(a, r).(\overline{a}.CRE_1(x) + \overline{r}.CRE_1(x))) \\ CRE_2(x) &= (\mathbf{v}a, r)(x(a, r).\overline{r}.CRE_2(x)) \end{aligned}$$

Furthermore we define two process compositions:

$$\begin{aligned} SYS_1(acc, rej) &= (\mathbf{v}l)(DEBT(l, acc, rej)|(\mathbf{v}c)(FICORP(l, c)|CRE_1(c))) \\ SYS_2(acc, rej) &= (\mathbf{v}l)(DEBT(l, acc, rej)|(\mathbf{v}c)(FICORP(l, c)|CRE_2(c))) \end{aligned}$$

Thus, the debtor in the first composition (SYS_1) eventually sends the loan request to a credit institute, which either accepts or rejects the request. In the second composition (SYS_2) the loan request is treated by the other credit institute, which always rejects the request. Obviously an accepted loan request is part of the goal definition for the debtor, therefore we define $DI_{SYS_1} = DI_{SYS_2} = \{acc, \overline{acc}\}$. As all above mentioned criteria concerning the participation of every process and interaction soundness are fulfilled, we can decide desired interaction compatibility. Hence, SYS_1 simulates $I_{acc}(acc) = (acc|\overline{acc})$, the processes in SYS_1 are desired interaction compatible. In contrast SYS_2 fails to simulate I_{acc}, thus the compatibility criteria are not fulfilled.

5 Desired Interactions with Operating Guidelines

As operating guidelines are a promising technique to specify potential interaction behavior, we also relate our ideas to this approach. The first part of this section introduces the main concepts of the operating guidelines approach. The second part extends the *well-communicating* criterion, in regard to desired interations. The discussion in this paper is restricted to bilateral process communication of a service provider and a service requester. Although operating guidelines are not restricted to bilateral scenarios, a discussion of multilateral process communication is beyond the scope of this paper.

5.1 Prerequisits: Operating Guidelines

Operating guidelines require processes to be defined as open workflow nets (oWFN) according to the formal model presented in [7]. These nets are workflow nets [14], enriched with places for asynchronous communication. Thus, an open workflow net is a Petri net $N = (P, T, F, m_0, \omega, in, out)$, specified by a set of places P, a set of transitions T, a flow relation $F \subseteq (S \times T) \cup (T \times S)$, an initial marking m_0, a set Ω of final markings and two sets $in, out \subseteq P$ containing the input and output places, respectively. Further, for all transitions $t \in T$: $card(\{p \in in|(p, t) \in F\} \cup \{p \in out|(t, p) \in F\}) \leq 1$ and if $p \in in (p \in out)$

then $(t,p) \notin F((p,t) \notin F)$. The inner of N is obtained by removing all interface places $p_i \in in \cup out$, together with their adjacent arcs. The behavior B^N of N is the reachability tree of the inner of N, whose edges are annotated with $!x$ ($?x$), if the corresponding transition is connected to an output (input) place x, and with τ otherwise. Therefore $B^N = (\sum, S, s_0, T, F)$ is defined via an alphabet $\sum = in \cup out$, a set S of states, an initial state $s_0 \in S$, a set of transitions $T \subseteq S \times L \times S$ with $L = \{?x | x \in in\} \cup \{!x | x \in out\}$ as the set of transition labels and a set $F \subset S$ of final states.

An operating guideline is an annotated automaton, which characterizes the set of all possible interaction behavior. It is defined as $OG_N = (\sum, S, s_0, T, F, \phi)$ for a Petri net N with the above mentioned definitions and an annotation function $\phi(s)$ mapping every state $s \in S$ to a Boolean formula with all outgoing transition labels of s as propositions. OG_N for an acyclic, deterministic (the non-deterministic case is described in [15], cyclic nets are still to be treated) Petri net N is constructed out of the complete behavior B^R of some partner oWFN R with an outgoing edge labeled with $!x$ ($?x$) for every input (output) place x of N. In an iterative process, all nodes representing undesired situations are removed from B^R. The resulting automaton B^* is the most permissive behavior. The operating guideline OG_N is constructed out of B^* by defining the annotation function ϕ. This function annotates every node with a Boolean formula over the labels of the outgoing edges. Consequently, the transitions required for proper interaction are identified. Please refer to [9] for any details.

5.2 Compatiblity Criterion based on Desired Interactions

Referring to [7] two processes are *well-communicating*, iff the behavior of one process (the requester) is an isomorphic subtree of the operating guideline of the other process (the provider) and all annotations are satisfied. Taking desired interactions into consideration we extend these criteria. Consequently, we require the behavior of the process acting as service requester to include the corresponding actions to all desired actions of the service provider. In this context every input (output) transition label $?x \in L$ ($!x \in L$) can identify a desired action, so that $DA \subseteq L$. Following the definition introduced in section 4.3, the set of desired interactions DI contains all desired actions and their corresponding actions.

We formalize the desired interaction criteria for an oWFN R (the requester) with its behavior B^R and an operating guideline OG_P for a service provider P in the following way:

$$\forall\ ?x, !x \in DI_{(R|P)} :\ (?x \in DA_P\ \Rightarrow !x \in B^R)\ \wedge\ (!x \in DA_P\ \Rightarrow ?x \in B^R)$$

Figure 2 illustrates the open workflow nets for the introduced example. Hence, we abstract from the dynamic binding of the debtor and a certain credit institute, the financial corporation is not presented. Below the oWFNs the operating guideline OG_{DE} for the debtor and the behavior automata of the two credit institutes are visualized. Again the exchange of an acceptance message

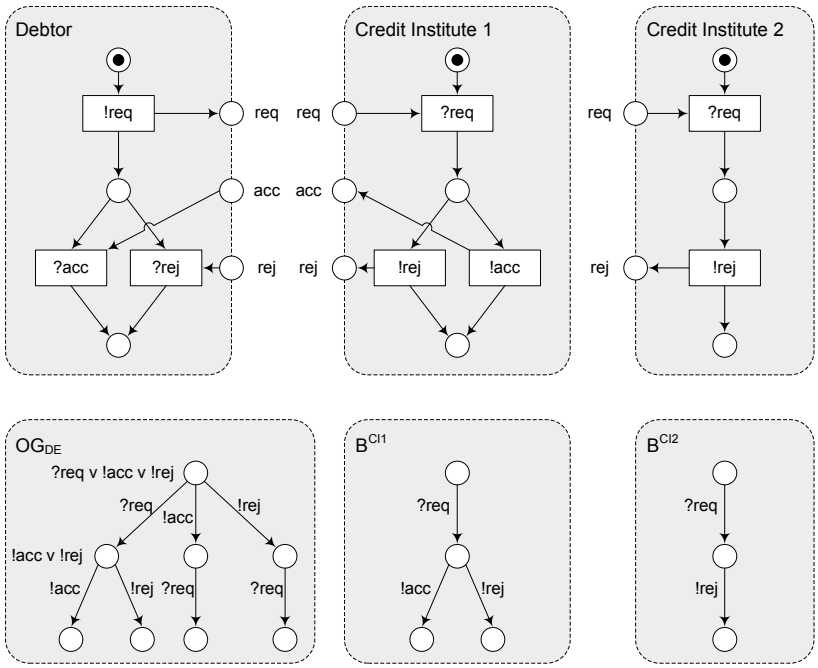

Fig. 2. Debtor with operating guideline OG_{DE} and two credit institutes and the related behaviors B^{CI1} and B^{CI2}

is a desired interaction, thus $DI_{(DE|CI1)} = DI_{(DE|CI2)} = \{?acc, !acc\}$. The two conceivable systems including the debtor and one of the credit institutes, are *well-communicating* as the behaviors are subtrees of the operating guideline and the annotations are fulfilled. Nevertheless only the composition including the credit institute $CI1$ is also desired interaction compatible, due to the absence of the transition label $!acc$ in B^{CI2}.

6 Conclusion

This paper motivates the need to consider the potential achievement of process goals in process integration scenarios. To put our ideas into perspective, we shortly discussed goal formalization, minimal structural compatibility and interaction soundness as related concepts. To address the demand for potentially successful interaction, a new compatibility notion based on desired interactions has been introduced. The approach presented extends interaction soundness, consequently processes possibly containing lazy activities are proved to be free of deadlocks and livelocks.

To ensure the reachability of certain interactions, we firstly defined desired actions and interactions using π-calculus as formal foundation. The π-calculus algebra has been chosen as it allows reasoning for business processes with dynamic name binding. Based thereon, we specified *desired interactions compatibility* and showed how simulation techniques can be used to decide it. We validated our findings using the Advanced Bisimulation Checker (ABC) [16]. Further on, we discussed desired interactions in the field of operating guidelines and presented an extended correctness criterion.

Future work remains to propagate the presented approach to operating guidelines for multiple partners [9]. In addition, our formalization of minimal structural compatibility requires static links between processes of a process composition. Thus, a formalization of this criterion, which takes dynamic binding into account is to be presented. Further work also has to focus on desired end states as a potential alternative to desired interactions, when considering goals in a compatibility notion.

References

1. Workflow Management Coalition: Terminology & Glossary (1999)
2. Martens, A.: On Compatibility of Web Services. Petri Net Newsletter **65** (2003) 12–20
3. Canal, C., Pimentel, E., Troya, J.M.: Compatibility and inheritance in software architectures. Sci. Comput. Program. **41**(2) (2001) 105–138
4. Puhlmann, F., Weske, M.: Interaction Soundness for Service Orchestrations. In Dan, A., Lamersdorf, W., eds.: Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC 2006). Volume 4294 of LNCS., Springer Verlag (December 2006) 302–313
5. Dijkman, R.M., Dumas, M.: Service-Oriented Design: A Multi-Viewpoint Approach. Int. J. Cooperative Inf. Syst. **13**(4) (2004) 337–368
6. Decker, G., Weske, M.: Behavioral Consistency for B2B Process Integration. In: Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAISE 2007), Trondheim, Norway. (2007)
7. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
8. Massuthe, P., Schmidt, K.: Operating Guidelines - an Automata-Theoretic Foundation for the Service-Oriented Architecture. In: Proceedings Fifth International Conference on Quality Software (QSIC 2005), Washington, DC, USA, IEEE Computer Society (2005) 452–457
9. Massuthe, P., Schmidt, K.: Operating Guidelines - an Alternative to Public View. Informatik-Berichte 189, Humboldt-Universität zu Berlin (2005)
10. Puhlmann, F.: Why do we actually need the pi-calculus for business process management? In Abramowicz, W., Mayr, H.C., eds.: BIS. Volume 85 of LNI., GI (2006) 77–89
11. Sangiorgi, D.: A Theory of Bisimulation for the pi-Calculus. Acta Informatica **16**(33) (1996) 69–97
12. Parrow, J. In: An Introduction to the π-Calculus. Elsevier (2001) 479–543
13. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)

14. Aalst, W.: The Application of Petri Nets to Workflow Management. (2000)
15. Massuthe, P., Wolf, K.: An algorithm for matching nondeterministic services with operating guidelines (January 01 2006)
16. Briais, S.: Advanced Bisimulation Checker (ABC) http://lamp.epfl.ch/ sbriais/abc/abc.html (2007)

Implementing Service Orchestrations using π-Calculus

Olaf Märker

Hasso Plattner Institute at the University of Potsdam

Abstract. This paper presents approaches to implement service orchestrations with the π-calculus. It describes how an execution engine can enable π-processes to invoke web services and how the invocations can be represented in π. Furthermore, a solution to model error handling on process level is introduced.

1 Introduction

The π-calculus, introduced by Milner, Parrow and Walker [5], is a calculus for communicating systems that can express processes with changing structure. It is used to model processes and to reason on processes. The π-calculus gives also the base for programming languages like Pict [6].

Furthermore, the π-calculus is discussed as candidate for modelling business processes. Theoretical work in this area was already done by the team of Mathias Weske at the Hasso Plattner Institute at the University of Potsdam. They have shown, for example, that it is possible to express the workflow patterns in π-calculus [7].

When business processes can be modeled with the π-calculus to reason on them then it should also be feasible to use these formal process descriptions in π as basis for automated process execution. Therefore, a execution engine comparable to BPEL engines should be created that uses π-processes as input. The benefit of such an engine would be the possibility to check the processes for soundness, e.g. the absence of dead locks, before they will be executed and the proof that the theoretical concepts for the π-calculus for business processes will work in a real world scenario. And of course the automatisation of business processes.

One common way of implementing business processes is the composition of web services. Such a loosely coupling of several service is also called service orchestration. An execution engine that should support service orchestration has at least to be able to call web services. It has to be discussed how the theoretical approaches of the π-calculus can be matched to service invocation in practise using existing technologies. Therefore, a lot of questions have to be solved to come to a solid foundation for an execution engine. This paper should give a starting point with a solution for the first three challenges:

1. How can π-processes communicate over the internet?
2. How can requests and responses be created in a π-process?
3. How can errors be handled?

First, an overview of the desired architecture will be given followed by an introduction to the π-calculus. Afterwards, each of the three questions should be discussed in an own section. At the end, a conclusion will summarise the presented approaches and the next steps toward an execution engine will be sketched.

1.1 Related works

In her master thesis, Anja Bog has developed a simulator that interprets π-processes. She describes how a process can be represented in a tree structure and how the reduction rules can be applied automatically [1]. This will give a solid foundation for the process handling of an execution engine.

The interpretation of π-processes is also discussed for programming languages based on π. There, the main concepts focus on objects and interactions between objects in terms of object oriented programming. Communication get the semantic of invoking methods of objects which fits only partly to the communication of entities over the internet [6][2].

1.2 Architectural overview

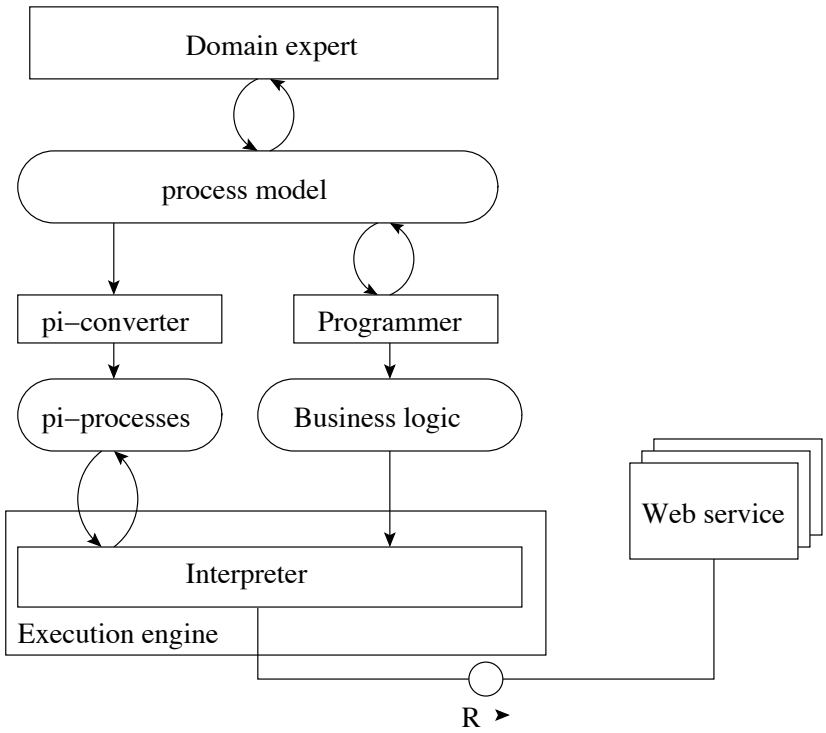

Fig. 1. System architecture as FMC compositional structures diagram

The scenario of application of the execution engine is displayed in figure 1. The goal is the automatisation of a business process that orchestrate web services. For that a domain expert can design a business process in a graphical editor. The process model can be converted to π-calculus processes. The π-processes can then be checked for soundness.

Furthermore, a programmer has to provide or implement all non-process functionality, for example the business logic. The resulting routines and the processes have to be deployed to the execution engine. Within the execution engine a π-interpreter operates on the processes and call business routines and external web services as is it defined in the processes. This π-interpreter will also be called `interpreter' in this paper.

1.3 The π-calculus

Table 1. The grammar of the π-calculus

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

The π-calculus is a process algebra with a small grammar that is displayed in table 1. The basics elements are process identifier symbolised by upper case letters, interactions represented as lower case letters, the inaction symbol **0**, and the Greek letter τ. A process will end with either a process identifier, which means that the associated process will be started, or the inaction symbol, that will terminate this process. A process identifier and an inaction can be prefixed with one or more interaction or τ's, each separated with a dot. The inaction symbol at the end of a process will often be omitted.

There are two types of interactions, a reading or input action $x(a)$ where x is the channel and a is a place holder for the transmitted name, and a complementary writing or output action $\overline{x}\langle b\rangle$ where b is the name that has to be transmitted over the channel x. τ stands for an unobservable action or communication.

Processes can be executed in parallel. This will be denoted as $A \mid B$ for the processes A and B. Furthermore, there can be an exclusive choice between two processes but both processes must have at least one prefix as in $x(a).A+y(b).B$. Only one side of the choice can be executed, usually this will be decided by the prefixes. In the example they are both reading actions. The first reading action, that receive a name will decide the choice for its side.

The handling of a process will be realised with the reduction semantics. The principle is that each action will be processed once and then vanish, so the process will be reduced. An action can only be reduced when it is not prefixed by another one. For that reason, only processes that are in parallel can communicate with each other. The communication in the π-calculus is synchronous, an output action on a channel can only be processed when there is an unprefixed input action in a parallel process that reads from the same channel. Both, the output and the input action will be reduced at the same time. For instance, the process $x(a).\overline{a}\langle y\rangle.A \mid \overline{x}\langle b\rangle.B$ can be reduced to $\overline{b}\langle y\rangle.A\{^{b}/_{a}\} \mid B$. As the a in the input action is only a place holder it has to be replaced with the name read from the channel for the rest of the process. So, the second interaction on the left side

will change from $\overline{a}\langle y\rangle$ to $\overline{b}\langle y\rangle$ because b is read in the former communication. The notion $A\{^b/_a\}$ means that each a in A will be replaced by b. Furthermore, it has to be mentioned that each name in the π-calculus can be used as channel and parameter. Processes like $\overline{a}\langle a\rangle.\mathbf{0}$ are also possible.

The scope of names can be restricted with the scope operator, the bold-face v. The name behind this operator will be distinct from all other names in the environment. The consequence is that no other process can write to or listen on this name as it is only be known by the creator process. But, the scope can be extruded by sending the name as parameter of an interaction to other processes.

There exist furthermore an operator for condition that is denoted $[x = y]$. In the basic π-calculus only the test for name equivalence is possible. When a condition will be processed then either the names are equal and the following prefix can be handled or they are unequal and so the rest of the sequence can not be executed anymore. An example for the condition operator is $x(y).([y = a]\overline{s}\langle y\rangle + [y = b]\overline{t}\langle y\rangle)$. When the name a will be read from the channel x, then all y have to be replaced by a. The result will be $[a = a]\overline{s}\langle a\rangle + [a = b]\overline{t}\langle a\rangle$. Here only the first condition match and the left side of the exclusive choice wins. When b was read instead then the right side wins.

The last π-construct is the replication operator, the exclamation mark. Each replicated process exists so many times as it is needed. For instance the process $!\mathbf{v}x\ \overline{gen}\langle x\rangle$ will so often generate a fresh name and send it on the channel gen as other processes read from the channel gen.

2 Approaches

In the next sections the three raised questions will be discussed, each in an own subsection, and solutions will be presented. It will start with the question: How can π-processes communicate over the internet?

2.1 Communication

To enable the interpreter to communicate with existing web services it has to use the same communication protocol. As many web services, specially SOAP-based services or services in a REST architecture, use the Hypertext Transfer Protocol (HTTP) [3] the interpreter should support this standard communication protocol, too.

HTTP (version 1.1 is the most common at the time of writing this paper) is a protocol that works with request/response pairs. Both, the request and the response will be send over the same connection. When a server receive a request then it will generate a response and send it immediately back to the requester. But this conflicts with the basics of the π-calculus where input and output actions are separated.

So, the question arises how the concept of request and response pairs could be represented in the π-calculus? For this question it is worth while to look at a possible implementation of the interpreter. When the interpreter has to process an output prefix it has to create a connection to the server and send the request over this connection. Over the same connection the interpreter receive the response from the server, regardless to the

current process execution. Web services usually do not respect the peculiarities of the π-calculus and do not wait for sending the response until the client-process listen for them. The interpreter has to buffer the response till the process itself read it with an input prefix. But the challenge is to match this buffered response to an input prefix from the process. As input and output interactions are separated in the π-calculus, processes can send a request, then doing something else like interact with other processes, and afterwards receiving the response. A solution for this challenge has to operate on the π-level. In the process model the matching from responses to requests should be non-ambiguously.

There are some approaches to represent requests/responses pairs in π. For example one can model the connection within the process that has to be opened and on which the request and the response would be sent. A similar approach is often used in the theory to transfer more then one message between two processes without the intervention of other processes, for example to encode the polyadic π-calculus in the monadic (standard) π-calculus [4]. But that leads to unwanted complexity within the process and clashes with the idea of the layered network protocol stack. The processes should use HTTP as a service that is provided by the interpreter.

A better approach will be the next one. The base theory of the π-calculus is about names that can be used as channels for communication and in the same way as values that should be transmitted while communicating. So a request can be sent to a service over a channel and then the service can use the request as channel to send the response back to the client.

$$Client \stackrel{def}{=} \mathbf{v}req\ \tau.\overline{x}\langle req\rangle.req(resp).\tau.P \tag{1}$$

$$Service \stackrel{def}{=} !x(req).\mathbf{v}resp\ \tau.\overline{req}\langle resp\rangle \tag{2}$$

The client in process 1 creates the request req and send it to the service via the channel x. The service creates the response $resp$ and send it to the client using req as channel. The name req is only known by the client and the service and no other process can disturb the communication between both processes. The client then read the response from the channel req and continue. The interpreter can now assign a buffered response exactly to an input prefix within the process as it is clear to which request the response belongs.

The advantages of the proposed representation are the explicit visualisation of the correlation between request and response and the simplicity of the modelling without the overhead of an additional connection. But, from the view of a modeller it can be strange to use a request, that is a document, as a communication channel. It can also become difficult when developing a type system for the π-calculus for service orchestration to distinguish between channels, documents like requests and responses and request which can be used as channel in some cases. To make it clear for the modeller the construct $\overline{req}\langle resp\rangle$ should have the semantic of 'answering a request with a response'.

In the pi-calculus the communication is synchronous so that one communication partner can first send its message when the counter part is listen on the same channel. But how could the interpreter recognise, that a service is not yet listening? This should be easy if the server does not response, that means a connection timeout occur while trying to connect to the server. But what when the server response with an error message. There

are several error codes and theire semantic defined in the HTTP specification. Messages like `404 File not found' can be interpreted as a not listening service whereas messages like `500 Internal server error' marks errors that should be handled. So, the interpreter has to divide between different error codes and has to adjust its behaviour.

When a service does not response or the error message indicates that the service is not available yet, the only way to implement the blocking behaviour of the π-calculus is to poll the service periodical. There exist no mechanism for web services to tell the clients when it is online.

2.2 Functional aspects

In the last section an approach was introduced that allows π-processes to communicate over HTTP with services and even with other processes. Requests and responses where created and transmitted between processes. But requests and responses are documents, in the meaning of HTTP messages with a head for meta information and a body with the user content. The theory of the π-calculus can only create names but no content.

In this section the second question should be discussed: How can requests and responses be handled within a π-process? A general approach will be introduced for data handling.

To enable π-processes to call web services there is the requirement to implement certain functionality. For example, to think of a SOAP-based service, the actual service call has to be put into a SOAP-envelope, that is a XML-document, and also the response from the service will be boxed into a SOAP-envelope. The client has to parse this document to use the response. But where should such functionality be implemented?

The π-calculus is of course a powerful and expressive system. So it was allready shown that even numbers can be encoded as process in π, and there are also object-oriented programming languages which are based on π. But the π-calculus was primarily designed to describe communicating systems with concurrent processes, and there are its strengths. Implementing data handling within π seems not to be a trivial task.

For the sake of simplicity the solutions presented in this paper will use the π-calculus only for communication and process handling. All functional parts, especially the creation and handling of requests and responses, should be implemented in other programming languages where well-engineered frameworks will support the developing. The interpreter will enable π-processes to call functions in other languages.

With this premise it should be easier to create a working execution engine for the π-calculus and it gives a lot of flexibility. Each kind of service that is using HTTP as communication protocol can be called from a π-process as the actual service protocol is independent from the interpreter. In later extension it should be possible to handle some functionality within the process. Therefore, the stated premise should not be understood as dogma, it will allow to focus the development of the interpreter to create a solid basis, and should be reevaluated en route.

But how can functions implemented in other languages be called from π-processes. As a function call is a internal construct and do not imply communication with other processes, it can be represented with the τ prefix. τ stands for an unobservable action or communication, where unobservable mean that it is no inter-processes communication. The τ has no visible influence on the process.

Now, the processes can be extended with a τ at each position, where requests and responses have to be created or other data handling will be required. That was allready done in the processes 1 and 2. As it can be seen there will be needed more than one function. To distinguish different functions the τ has to be annotated with an identifier for example with a function name so that the interpreter knows which function should be called. This could be done for the service and client process as follow:

$$Client \stackrel{def}{=} \mathbf{v}req\ \tau_{createRequest}.\overline{x}\langle req\rangle.req(resp).\tau_{handleResponse}.P \tag{3}$$

$$Service \stackrel{def}{=} x(req).\mathbf{v}resp\ \tau_{handleRequest}.\overline{req}\langle resp\rangle \tag{4}$$

At the beginning the client creates the request. Therefore, the π-name req will be introduced and the function $createRequest$ has to be called. In this function the request will be filled with content. After receiving the request the service has to interpret it, carry out the requested service, and create the response. This will be done in the function $handleRequest$. When the response was send back to the client, it has to interpret the results within the function $handleResponse$.

2.3 Error handling

Up to now, the interpreter is able to allow communication with HTTP over the internet and requests and responses can be created and handled. This should be enough to orchestrate services within a process. But, as the goal is to call services in the real world where a lot of errors can occur it is requisite to cope with them.

In the theory of the π-calculus the term 'error' is mostly used for wrong defined processes that will cause problems on the process level when the system evolves. One well-known problem arise from the polyadic π-calculus where more than one name can be send over a channel at the same time. At this point runtime-errors can occur when the count of names that will be sent is unequal to the count of read names from the same channel. Some times this is also called 'communication error' [9].

As this paper focuses on the implemention of an execution engine the term 'error' will not be used in the meaning of wrong process definitions, but as failure or exception in connected subsystems that prevents the application of the reduction rules with theire semantic.

Errors can occur at each interaction and in function for creating and handle request and responses. The former are communication error, which can be the consequence of a disturbed or congested network connection, and the latter will results from bugs and unhandled exceptions in the functions. In the theory of the π-calculus interactions are atomic actions that can take place when there exist an unprefixed output action and an unprefixed input action on the same channel. In the monadic π-calculus where only one name can be transmitted per interaction the interaction are assumed to succeed. This is also true for τ.

In this section a solution will be presented, that allow to handle errors in a generic way and that is conform to the reduction rules of the π-calculus. For that solutions two assumptions have to be made.

Assumtion. Each interaction will first be reduced in the process when the underlying communication ended successfully and each τ will first be reduced when the invoked function call finished without unhandled exceptions and errors.

Assumtion. The interpreter will send a message to the π-channel e when an error occur while communicating or calling a function.

These both assumptions allow to introduce the process $e(message).E(message)$ that will start an error handling process E when the interpreter sends a message to the channel e. This process can now be added with an exclusive choice to all critical actions like in the following example: The process $\overline{x}\langle a\rangle.P$ can be extended to $(\overline{x}\langle a\rangle.P + e(message).E(message))$. In this situation two things can happen. Either the communication over the channel x finish successfully, then the output action can be reduced and the left side of the choice is chosen, or an error occur and the interpreter sends a message over the channel e and the right side of the exclusive choice will be executed.

This concept applied to the client process 3 will lead to:

$$\begin{aligned} Client \stackrel{def}{=} \mathbf{v}req\ \tau_{createReq}.(\overline{x}\langle req\rangle.(req(resp).(\tau_{handleResponse}.P + e(m).E(m)) \\ + e(m).E(m)) + e(m).E(m)) + e(m).E(m) \end{aligned} \tag{5}$$

Of course, this will increase the size of each process but it will only have linear influence to the runtime complexity. As this approach is generic, it is possible to extend all processes automatically with this standard error handling. Furthermore, the process E can be substituted with more specific processes when needed.

One error case need special handling. When a service receive a request and afterwards an error occur at the server, the client has to be informed as it wait for a response. The service process could be defined as follow:

$$\begin{aligned} Service \stackrel{def}{=} x(req).(\mathbf{v}resp\ \tau_{handleRequest}.(\overline{\overline{req}}\langle resp\rangle + e(m).E(m)) + \\ e(m).\mathbf{v}error\ \tau_{error}.\overline{\overline{req}}\langle error\rangle.E(m)) + e(m).E(m) \end{aligned} \tag{6}$$

When an error occur in the function $handleRequest$, instead of the response an error message will be created and sent to the client. This could be for example an HTTP-message with the HTTP error code 500 'Internal server error'.

3 Conclusion

In this paper an approach was introduced that describe how an orchestration of web services could be implemented using the π-calculus. The orchestration is modeled as process with respect to the request/response characteristic of the used communication protocol HTTP. An execution engine can interpret and process the π-models. The handling of data is supposed to be realised in other programming languages. Therefore, it is possible to call functions in other languages from a π-process which is denoted by a τ annotated with the function name. Errors can be handled with a generic error handling block. Processes can automatically be extended with these blocks.

With the presented results it should be possible to create an execution engine for the π-calculus for service orchestrations. But, further questions arise and should be answered when developing the execution engine. The theory of the π-calculus deals with

names, the interpreter with URLs, requests and responses. Requests could only be send to URLs and each request has to be matched by one response. To enforce the correct use of the elements a suitable type system is required.

The examples in this paper were easy ones without a complex workflow. Thus it has to be shown that the introduced approaches harmonise with complexer constructs, in particular with the workflow patterns. Moreover, the compatibility to existing soundness checking algorithm has to be proven.

These questions should be solved in my master thesis which will follow. There an execution engine for the π-calculus for service orchestrations should be designed.

References

1. Anja Bog. A visual environment for the simulation of business processes based on the pi-calculus. Master's thesis, Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam, October 2006.
2. Silvano Dal-Zilio. An interpretation of typed concurrent objects in the blue calculus. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 409--424. Springer, 2000.
3. Roy T. Fielding et al. Hypertext transfer protocol - http/1.1. RFC 2616, W3C, June 1999.
4. Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.
5. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1--77, September 1992.
6. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455--494. MIT Press, 2000.
7. Frank Puhlmann and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In W.M.P. van der Aalst et al., editor, *Business Process Management*, volume 3649 of *LNCS*, pages 153--168. Springer-Verlag, Berlin, December 2005.
8. Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
9. Vasco Thudichum Vasconcelos and Antonio Ravara. Comunication errors in the π-calculus are undecidable. *Information Processing Letters*, 71:229--233, 1999.