Oliver Kopp, Niels Lohmann (Hrsg.)

Services und ihre Komposition

Erster zentraleuropäischer Workshop, ZEUS 2009
Stuttgart, 2.-3. März 2009

Proceedings

CEUR Workshop Proceedings Vol. 438

Herausgeber:

Oliver Kopp
Universität Stuttgart, Institut für Architektur von Anwendungssystemen
Universitätsstraße 38, 70569 Stuttgart, Deutschland
oliver.kopp@iaas.uni-stuttgart.de

Niels Lohmann
Universität Rostock, Institut für Informatik
18051 Rostock, Deutschland
niels.lohmann@informatik.uni-rostock.de

ISSN 1613-0073 (CEUR Workshop Proceedings)

Online-Proceedings verfügbar unter http://CEUR-WS.org/Vol-438/

BibTeX-Eintrag für Online-Proceedings:

```
@proceedings{zeus2009,
  editor    = {Oliver Kopp and Niels Lohmann},
  title     = {Proceedings of the 1st Central-European Workshop on
               Services and their Composition, ZEUS 2009,
               Stuttgart, Germany, March 2--3, 2009},
  booktitle = {Services und ihre Komposition},
  publisher = {CEUR-WS.org},
  series    = {CEUR Workshop Proceedings},
  volume    = {438},
  year      = {2009},
  url       = {http://CEUR-WS.org/Vol-438/}
}
```

Vorwort

Der *Zentraleuropäische Workshop über Services und ihre Komposition (ZEUS)* ist kein klassischer Workshop, bei dem fertige Forschungsergebnisse veröffentlicht und präsentiert werden. Stattdessen steht die Diskussion von Ideen, die sich noch in einem frühen Entwicklungsstadium befinden, im Vordergrund. So erhalten Teilnehmer bereits vor der Einreichung eines Konferenzpapieres erste Rückmeldungen außerhalb ihrer Gruppe. Darüber hinaus erlaubt der Workshop durch seine regionale Ausrichtung den Aufbau eines wissenschaftlichen Netzwerkes, das intensiv und zu geringen Kosten genutzt werden kann. Er richtet sich dabei insbesondere an junge deutschsprachige Forscher im Service-Bereich.

Der erste ZEUS-Workshops findet am 2. und 3. März 2009 in Stuttgart statt. Veranstalter ist das Institut für Architektur von Anwendungssystemen der Universität Stuttgart.

Es gab 17 eingereichte Beiträge, die alle nach kurzer Begutachtung in das Programm aufgenommen wurden. Weiterhin wird es einen eingeladenen Vortrag von Prof. Karsten Wolf (Universität Rostock) geben. Wir hoffen, dass die Vorträge eine gelungene Grundlage für rege Diskussionen bieten.

Stuttgart/Rostock, im Februar 2009

Oliver Kopp
Niels Lohmann

Gutachter

Katharina Görlach	Univeristät Stuttgart
Kathrin Kaschner	Universität Rostock
Oliver Kopp	Univeristät Stuttgart
Niels Lohmann	Universität Rostock
Ganna Monakova	Universität Stuttgart
Ralph Mietzner	Universität Stuttgart
Olivia Oanea	Universität Rostock
Dieter H. Roller	Universität Stuttgart
Steve Strauch	Universität Stuttgart
Tobias Unger	Univeristät Stuttgart
Sema Zor	Univeristät Stuttgart

Sponsor

Die Organisatoren danken der Firma Senacor für die finanzielle Untersützung der Ausrichtung.

Inhaltsverzeichnis

Eingeladener Vortrag

Modellierung und Spezifikation

Choreographien

Verifikation

Ausführung

A Theory of Service Behavior

Karsten Wolf

Universität Rostock, Institut für Informatik

Abstract. We outline a fundamental approach to behavioral aspects of services. In the center of this approach, we see behavioral models of services, interactions, and finite representations of sets thereof. Several operations and relations can be defined and their implementation on our representations can be studied. Finally, a number of interesting problems can be traced back to our models and operations. On the boundary of our theory, we place interfaces to other aspects of services.

1 Introduction

Services are made for being loosely coupled to larger artifacts. A reasonable coupling (i.e. interaction via message transfer) must take care of various aspects of compatibility, including:

- Semantical compatibility: what does the content of exchanged messages mean?
- non-functional compatibility: how is the exchange of a message organized?
- behavioral compatibility: in which order are messages exchanged?

We target the behavioral aspect of compatibility. The remaining aspects are taken care of by a well-defined interface which enables us to integrate any techniques, approaches, and results regarding semantics or non-functional aspects of service composition.

2 Representing Service Behavior

There seem to be two complementary approaches to the specification of service behavior. In the first approach, we specify the control flow of a single service (end point, peer, participant). This control flow implicitly constrains the order of messages that are transmitted via the middleware. In the second approach, we specify sequences of message transmissions which we want to see in the middleware (a choreography). A theory of service behavior should support both points of view.

For the specification of single services, we propose to use *service automata*. They offer concepts of state and transition for modeling control flow. Transitions can be labeled with primitives like sending or receiving a message thus modeling the interaction behavior of a service. Service automata may be compactly represented as Petri nets or expressions in a process algebra thus inheriting

several results from existing theories. Service automata are well linked to relevant languages like WS-BPEL or BPMN as there exist back-and-forth translations between these languages to Petri nets [1, 2], and back-and-forth translations between Petri nets and service automata.

For the specification of a choreography, we have not yet identified a canonical formalism. On one hand, a set of traces of interaction primitives (like sending or receiving a single message) seems to be a more reasonable starting point than process description formalisms like the pi-calculus. The reason is that the middleware that connects services is not an actor which deliberately takes decisions. It is rather a medium that records the effect of decisions taken elsewhere. On the other hand, some authors insist that it might be essential for a choreography description to record who is in charge for selecting a particular sequence from the space of opportunities given by a set of traces. We conclude that the selection of an appropriate formalism for a theoretical study of choreographies is still a future work task.

In addition to the specification of a single behavior, we consider specifications of sets of behaviors. A meaningful example in the case of single services is an operating guideline, i.e. a finite representation of the set of all compatible partner services to a given service. We believe that many interesting problems can be traced back to simple questions concerning single behaviors or sets of behaviors.

3 Operations and Relations on Service Behaviors

Among the important operations on behavior, there are of course a few trivial ones. These include, for example, the composition of services to larger ones or ecxtracting the set of traces in the middleware that can be realized by a given composition of services.

A next class of operations concerns the synthesis of missing components to an incomplete specification. We already have algorithms for the synthesis of compatible partners to a given services [3], or for synthesizing an adapter to a set of incompatible services [4]. Similar techniques should work for synthesizing end points to a given choreography. In many known scenarios, a synthesis algorithm that calculates a single fitting behavior can be extended to an algorithm that computes a finite representation of all fitting behaviors of some kind [5, 6].

In a third class of algorithms, we investigate standard operations on our core objects. As a starting point, we look for realizations of standard set operations (intersection, union, complement, projections) to representations of sets of service behaviors. The actual challenge is that we start with, and want to arrive at, finite representations of infinite sets (where each element is as complex as the control flow of a service). We have some indication, that we will succeed with a minor extension of the structures used for operating guidelines (characterization of the set of all compatible partners of a given service). Further down in this article, we sketch some useful applications.

In a fourth class of operations, we would transform given service behaviors. Potential applications include the repair of malfunctioning compositions [7], the generation of public views out of private views, or vice versa [8, 9].

Service behaviors need to be compared with each other. A core concept in this regard is the one of equivalence. Several equivalence notions for services have already been proposed. Not all of these notions are very well motivated. For instance, substituting a service with a trace equivalent one may not preserve compatibility with other services. On the other hand, requiring bisimulation equivalence is often too strong a requirement that prevents harmless substitutions.

We believe that reasonable equivalence notions need to be derived from application scenarios. As an example, study a scenario where a service is substituted by another one such that all compatible partners of the old service remain compatible with the new one. The corresponding equivalence holds between all services with the same set of compatibvle partners. It turns out that this equivalence is strongly tied to the process algebraic notion of a should testing equivalence which is a non-trivial entry in the huge zoo of equivalences proposed in process algebra. In future work, we want to identify other application scenarios which may call for different notions of equivalence.

4 Targeted Problems

Of course, the objective of our theory is to provide useful solutions to interesting problems. In this section, we demonstrate that the outlined theory of service behavior yields approaches to a number of interesting problems.

Verify and validate a service

Through the synthesis of a compatible partner [3], we may prove the principal wellformedness of a service. Some initial approaches suggest that even the construction of diagnosis information for a given malfunctioning service involves operating guidelines, i.e. a core element of our theory [10]. Using the set of all compatible services [5], we may compare the external effect of a service to a specification. In particular, we may verify whether or not certain targeted partners are among the compatible ones. The characterization of all partners may yield useful (positive) test cases in some scenarios [11] while the complement of that characterization might include negative test cases.

Construct a service

Due to an already existing link from service automata via Petri nets to abstract WS-BPEL, we may support the automatic generation of services for various purposes. These services are compatible by construction. Our theory enables a flexible selection of services to be generated [12]. We may, for instance, translate various requirements into finite representations of sets of services and then use intersection as an instrument for filtering some desired behavior out of the set of all compatible ones.

Compose services

Composition may be supported for instance by synthesizing missing components (adapters) [4], by transforming participants of a malfunctioning composition [7], by exchanging components (for instance, public views with private views in contract scenarios) [8]. We may support the selection of services from repositories. For finding a compatible partner of a given service R in a repository of services $P_1, \ldots, P_n$, we may check containment of R in one of the sets OG_i $(1 \leq i \leq n)$ where OG_i is the (finite representation of the) set of compatible partners of P_i. Even more efficiently, we may precompute the unions $OG_1 \cup \ldots \cup OG_{ndiv2}$ and $OG_{ndiv2+1} \cup \ldots \cup OG_n$ to more quickly reduce the search space. Using this idea, we may select a suitable P_i with $\log n$ containment checks instead of n such checks.

Replace a service

Using the right equivalence notions, replacement of services can be done without harming compatibility. Our existing substitutability notion can be traced back to basic set operations as follows. Let OG_X be the set of compatible partners of service X. Then P can be exchanged with R iff $OG_P \subseteq OG_R$ which is equivalent to $OG_P \cap \overline{OG_R} = \emptyset$. The remaining emptyness check should be easy. Moreover, a nonempty $OG_P \cap \overline{OG_R}$ canonically provides examples which prove non-substitutability. Such an example may help in providing diagnostic information and is not available in existing approaches to checking substitutability.

Verify and validate a choreography

There exist notions of realizability of a choreography. We believe that it is possible to translate the realizability problem into a partner synthesis problem. In the future, we may see more interesting problems concerning choreographies.

5 Problem Parameters

Most problems and solutions can be formulated in several settings. So far, we have identified the following parameters which more or less influence all approaches stated so far.

Compatibility notion

There are several reasonable notions of compatibility. The one occurring most frequently is deadlock freedom in the composed system. One may also require that it should always be possible to reach a designated terminal state, or to have a composed system that is sound (as defined for workflow models). Instead of possible termination one can also require eventual termination. The latter notion requires that the model contains information about fairness of decisions in the control flow. Additional user definable constraints may parametrize compatibility.

Nature of message passing

In a canonical setting, message passing is thought of being asynchronous. Existing approaches do or do not allow overtaking of messages. In the literature, synchronous communication is frequently studied, too. Further, we may or may not consider constraints that are implied by the semantics of messages. For instance, semantics may identify message type a as "empty form" and b as "filled form" which implies that it would not make sense to send b before having received a. Further down, semantical constraints are discussed in more detail.

Distribution of partner

Many services have interfaces to more than one partner. For synthesizing partners, we may or may not assume the capability of those partners to be coordinated during run-time or during build-time. This leads to different results concerning well-formedness [13].

6 Interface to Other Aspects

For being applicable, our solutions must be in line with the remaining aspects of service compatibility. As an example, we sketch an interface to semantics which we already found useful in the context of adapter synthesis.

We already mentioned that the semantics of messages may imply constraints on the behavior of a synthesized service. Examples of such constraints include

- Do not send a filled form before having received an empty one (while the order of unrelated messages does not matter)
- Do not send a message containing somebody else's password without having received it in another message (wihle you may send your own password without having received any message)

We claim that most relevant semantical constraints can be expressed in terms of transformation rules such as empty_form $\rightarrow$ filled_form, $\rightarrow$ own_password, meter $\rightarrow$ feet, or street + zipcode + name $\rightarrow$ address. The rules specify the semantically implied effect of message contents on the behavior, without implying any particular approach to represent or discover semantics as such. In fact, the rules may be specified manually, synthesized from semantic web approaches, etc. We already showed that it is possible to trace back synthesis problems in presence of semantical constraints to plain synthesis problems [4, 14].

7 Tools

There is already a family of tools which provide some of the discussed functionality:

- LoLA for the exploration of state spaces and thus for the investigation of complete compositions;

- Fiona for the synthesis of compatible partners and partner sets as well as for the synthesis of adapters, checking compatibility and a few other applications;
- BPEL2oWFN and oWFN2BPEL for the translation between WS-BPEL and Petrio nets;
- Rachel for a repair of a malfunctioning choreographies
- and certainly a number of tools developed in other groups.

They prove that, to the degree innplemented, operations can indeed be applied to realistic service specifications.

8 Conclusion

We propose a reasonable set of objects and operations to constitute a theory of service behavior. We have already identified a number of interesting problems which all can be traced back to a small number of operations on recurring kinds of objects like sets of service behaviors (also known as operating guidelines). We believe that a further consolidation of the theory would yield additional insights into the nature of services and their composition. Moreover, tracing back many interesting problems to a few operations helps us to strengthen the tool support for a large variety of problem settings.

References

1. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In Dumas, M., Heckel, R., eds.: Web Services and Formal Methods, Forth International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007, Proceedings. Volume 4937 of Lecture Notes in Computer Science., Springer-Verlag (2008) 77–91
2. Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In Kühne, T., Reisig, W., Steimann, F., eds.: Modellierung 2008, 12.-14. März 2008, Berlin, Proceedings. Volume P-127 of Lecture Notes in Informatics (LNI)., GI (2008) 57–72
3. Wolf, K.: Does my service have partners? LNCS ToPNoC **5460**(II) (2009) 152–171 Special Issue on Concurrency in Process-Aware Information Systems.
4. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (2008)
5. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25–29, 2007, Proceedings. Volume 4546 of Lecture Notes in Computer Science., Springer-Verlag (2007) 321–341
6. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. (2008) (Accepted for publication in December 2008).
7. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In Dumas, M., Reichert, M., Shan, M.C., eds.: Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 1–4, 2008, Proceedings. Volume 5240 of Lecture Notes in Computer Science., Springer-Verlag (2008) 132–147

8. Aalst, W.M.P.v.d., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. (2008)
9. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract WS-BPEL processes. In Ma, W.Y., Tomkins, A., Zhang, X., eds.: Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21–25, 2008, ACM (2008) 785–794
10. Lohmann, N.: Why does my service have no partners? In Bruni, R., Wolf, K., eds.: Web Services and Formal Methods, Fifth International Workshop, WS-FM 2008, Milan, Italy, September 4–5, 2008, Proceedings. Lecture Notes in Computer Science, Springer-Verlag (2008)
11. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In Feuerlicht, G., Lamersdorf, W., eds.: Service-Oriented Computing – ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Workshops Proceedings. Lecture Notes in Computer Science, Springer-Verlag (2008) (to appear).
12. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In Alonso, G., Dadam, P., Rosemann, M., eds.: Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24–28, 2007, Proceedings. Volume 4714 of Lecture Notes in Computer Science., Springer-Verlag (2007) 271–287
13. Wolf, K.: Does my service have partners? LNCS ToPNoC **5460**(II) (2009) 152–171 Special Issue on Concurrency in Process-Aware Information Systems.
14. Wolf, K.: On synthesizing behavior that is aware of semantical constraints. In Lohmann, N., Wolf, K., eds.: 15th German Workshop on Algorithms and Tools for Petri Nets, AWPN 2008, Rostock, Germany, September 26–27, 2008, Proceedings. Volume 380 of CEUR Workshop Proceedings., CEUR-WS.org (2008) 49–54

A scenario is a behavioral view – Orchestrating services by scenario integration

Dirk Fahland

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
fahland@informatik.hu-berlin.de

Abstract. The construction of a complex service orchestration is a tedious and error-prone tasks as multiple service interactions with a single orchestrating service must be specified and combined. We suggest to specify a service orchestration in terms of behavioral scenarios that capture a specific aspect of service interaction, a *behavioral view* in isolation. By synchronizing the different scenarios, the views get integrated and define the behavior of a complex service orchestration. Our formal model for scenarios and their integration is a class of Petri nets called oclets.

Keywords: service choreography, view, scenario, Petri nets

1 Behavioral views for service orchestration

In service-oriented computing [1], services serve as building blocks for complex, distributed systems. A service orchestration is a means to coordinate n services by a $(n+1)$st service, called *orchestrator* that communicates with each of the n given services directly and coordinates the overall exchange by its internal logic [2]. Process modeling languages like BPEL combine workflow modeling with the SoC paradigm for specifying and executing orchestrator services.

While specifying the orchestrator's interaction with a specific, individual service is usually straight forward, the coordination of all interactions with all partner services remains a challenge. If service interactions are sufficiently complex and depend on each other, the resulting orchestrator logic will be complex, and so will be the orchestrator model. Constructing a comprehensible orchestrator model with appropriate sub-processes etc. is a tedious task. The sub-process hierarchy usually needs to be remodeled if another service interaction, that cuts across the present hierarchy, is to be integrated into the model.

The problem itself is not new and relates to the problem of *process integration* for classical (non-communicating) process models [3]. A recurring solution approach for behavioral integration takes inspiration from databases where a complex relational database is constructed by integrating the *views* of the various users on the database [4]. A database view is, in principle, a projection of a complete database onto a few specific somehow connected objects of interest. Such a view has a valid interpretation in isolation. Conversely, a "complete"

set of views describes the entire database; hence it can be constructed from its views. If the set of views is not complete or do not fit, they have to be adjusted, viz. *integrated*.

In this paper, we propose the concept of a *behavioral view* for the construction of behavioral models. In analogy to databases, a behavioral view is, in principle, a projection of a complete behavioral model onto a specific behavior, i.e. a partial process execution also called *scenario*. Conversely, a "complete" set of behavioral views describes the entire behavior. A behavioral view can have arbitrary structure (as longs as it is a connected partial execution); it may therefore cut hierarchies and hence is a means to express cross-cutting concerns in process models. While the definition of a behavioral view is straight forward, the converse, their integration to form a complete behavioral model is non-trivial.

We argue that a well-founded approach for behavior modeling by view integration requires an appropriate formal model. In [5] we proposed the Petri net class of *oclets* as a formal model for scenarios (or behavioral views) on the basis of a formal, operational semantics [6]. Intuitively, this formal model constructs complex behavior by concatenating and merging "fitting" scenarios. This reduces the problem of behavioral integration to make a given set of scenarios "fitting" to each other. In general, the solution is to unify the tasks and resources occurring in the scenarios appropriately.

In the remainder of this paper, we first substantiate the concept of a behavioral view in Sect. 2 as we introduce oclets, and their semantics at an intuitive level. We subsequently explain the problem of behavioral integration and suggest a procedure for process integration by the help of an example. We conclude the paper with a discussion of related work and a presentation of open research problems in Section 4.

2 Scenario-based service modeling with oclets

For formally modeling services, the Petri net class of *open nets* has been established. Open nets allow for a rigorous analysis of behavioral properties of services while industrial service modeling languages can be translated to open nets [7], and vice versa [8]. The behavior of an open net is defined by standard Petri net semantics; this operational semantics defines the (partially-ordered) runs of the modeled service. As every service model is finite, these runs are not arbitrary, but exhibit a certain structure, e.g. transitions always fire in a specific order.

Scenario-based models exploit this regularity of the runs: One can identify various repeating patterns (*scenarios*) that are partial executions of the service. The entire service behavior is composed of these scenarios. A scenario-based model makes a scenario a modeling artifact. The entire service behavior is expressed as a set of scenario; a corresponding formal semantics describes how scenarios compose to runs.

In [5], we propose the Petri net class of oclets for formally describing scenarios with an operational formal semantics. An *oclet* is an acyclic Petri net with a local precondition describing which requirements must be satisfied in order

to execute the subsequent scenario. We denote service communication by annotating transitions by incoming arrows (receive a message) and outgoing arrows (send a message).

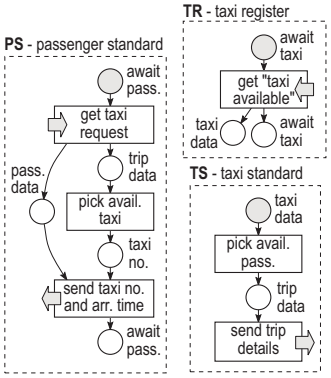


Fig. 1. Standard service interaction with passengers and taxis.

Figure 1 depicts some oclets; the minimal (no predecessor) grey-shaded places denote the precondition of the oclet. Our running example is a passenger-taxi coordination service which is specified in two views:

(1) The first view (oclet PS) specifies that a passenger can send a pickup request to the service, which it processes by picking an available taxi and returning taxi number and arrival time.
(2) The second view (oclets TR and TS) specifies that an available taxi can register at the service at any time. If taxi data is registered at the service, the service will pick a matching passenger and send the corresponding trip details to the taxi.

Figure 3 depicts a partially ordered run of the oclets PS, TR, TS as an acylic Petri net. The run is constructed by merging (copies of) oclets at equally labeled nodes in the obvious and intuitive manner. The run of Fig. 3 shows explicitly that the passenger view and the taxi view are not related to each other, while each makes sense on its own.

We can easily extend our view based model with another view, see Fig. 2: A passenger may cancel a request at any time (oclet PC); the service resets its processing subsequently (PR). This allows to construct the run depicted in Fig. 4. Thereby oclet PS of Fig. 1 is not executed completely (transition send taxi no is not enabled because get cancel request of PC occurred. Instead, oclet PR is appended by merging transitions pick avail. taxi of PS and PR.

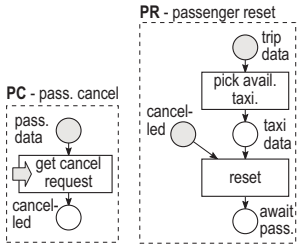


Fig. 2. Service cancelation by passenger and service reset.

The run of Fig. 4 shows again that the given behavioral views do not fit to each other; the taxi still gets notified about the passenger although the request has been canceled. The views must be integrated.

3 View integration with scenarios

We just have introduced oclets as a modeling language for behavioral views and show that if behavioral views do not fit to each other, they cannot be composed to a run. In this section, we sketch how behavioral views of services can be integrated to define a consistent orchestrator service.

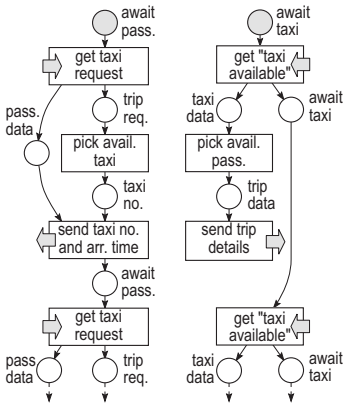


Fig. 3. A standard run constructed from scenarios ps, tr, ts.

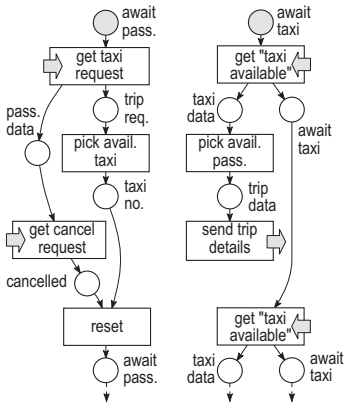


Fig. 4. A run with service cancellation constructed from scenarios ps, tr, ts, pc, tc.

We suggest the following *integration procedure* depicted in Fig. 5. The orchestrator's interaction with each of its partner services is specified in a behavioral view. To integrate the different views, (1) all views are refined to the same granularity of resources and tasks; (2) a modeler identifies points of synchronization (specific resources or tasks) and defines integration options. Finally, (3) synchronization is made explicit in each scenario by applying the integration options. This adjusts the different scenarios to each other s.t. the formal semantics of oclets constructs the orchestrator's behavior by concatenating and merging the now fitting scenarios. Possibly, some integration issues are not visible after the first integration step, so steps (2) and (3) are iterated until satisfaction. The grey tasks of Fig. 5 require interaction with a human modeler.

As all oclets of our example process already have equal granularity, the first step changes nothing. We now have to define integration options for our scenarios in order to unify the given oclets accordingly. An *integration option* maps a set of transitions of the oclets to be integrated to a (possibly new) transition. For some kinds of integrations like *parallel synchronization* the resulting integrated transition is a function of the integration inputs; see [3].

In our example, the standard behavior in Fig. 3 suggests to integrate the views via transitions pick avail. taxi and pick avail. pass by the parallel composition depicted in Fig 6. Further, the reset oclet pr of Fig. 2 must be extended to properly handle cancelation also in the service's interaction with the taxi. Applying this integration on all oclets in the subsequent unification step re-

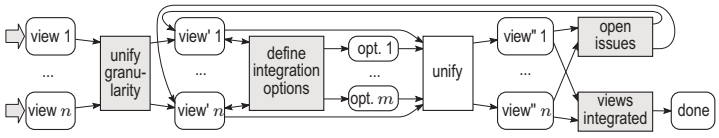


Fig. 5. Procedure for integrating behavioral views.

places transitions pick avail. taxi and pick avail. pass each by the integration result match request; see oclets PS2, TS2, and TCS in Fig. 7.

One can quickly see that this integration option alone is insufficient: The reset transition of TCS only resets the passenger thread of the process while the taxi thread is unmodified. A human modeler who inspects the integration result TCS can detect this problem. Hence, another (obvious) integration option that extends transition reset of Fig. 7 by the dashed dependencies must be specified. After this integration step, the passenger view and the taxi view are integrated, but still exist in isolation.

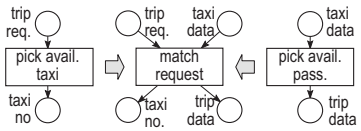


Fig. 6. View integration option

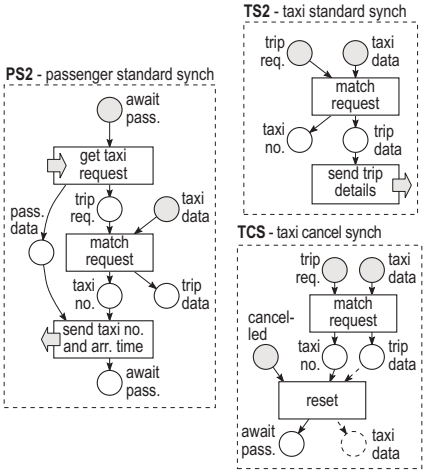


Fig. 7. Integrated oclets with unified transitions and enabling conditions.

The integrated service model is now given by oclets PS2, TR, TS2, PC, and TCS. Figure 8 depicts a standard run of the integrated service while Fig. 9 depicts a run with cancelation. Thereby, the match request transitions of the various oclets are merged upon construction of the runs as they are now pairwise compatible in term of enabling condition and effect. The integration of the different scenarios on common transitions is a consequence of our formal model [6].

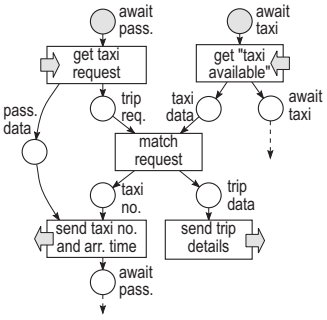


Fig. 8. A standard run of the integrated scenarios ps2, tr, ts2.

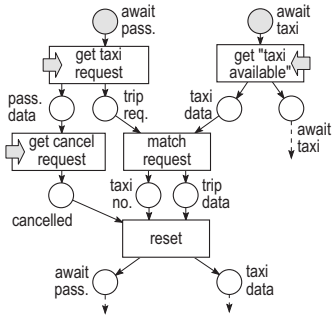


Fig. 9. A run with service cancelation of the integrated scenarios ps2, tr, ts2, pc, tcs.

4 Conclusion

We proposed scenarios as behavioral views for modeling complex service orchestrations. The interaction of an orchestrating service with each of its partner services is modeled separately in terms of their interaction scenarios. In a subsequent integration step, the different views are first unified in granularity; then integration options for synchronizing different views are defined by the modeler; finally, the views are unified according to the integration options. We proposed the Petri net class of oclets as a formal model for scenarios; their operational semantics allow to construct the behavior of the orchestrator directly from the unified scenarios.

Our proposition is a contribution to the relatively novel field of service (or process construction) by view integration [9]. Currently, there exists no systematic solution for behavioral process integration. Initial works like [4, 10] consider the problem from an information system perspective where database schema integration is extended by considering behavior of data processing as well. The problem of behavioral integration is now also being research in isolation. The general line of research aims on constructing complex, integrated process models by merging smaller process models (the views) on synchronization points. In [11, 12] different behavioral process integration options are considered for constructing such merged process models. These integration options can be applied when merging UML activity diagrams for constructing complex process models [3]. Similar results are available for Petri net based models [10] and EPCs [13].

One markable observation in all these approaches is that prior to integration, the process models have to be flattened in order to define and apply the integration options. While some sort of task grouping can still be applied [10], the modeler essentially works on an ever growing complex model. Because process integration involves frequent human interaction (see Sect. 3), this complexity becomes a problem. One of the main problems appears to be that in classical models, behavioral integration is achieved only by model composition. The notion of behavioral view must essentially be given up in order to integrate. Because, at the same time, the model must be flattened, there are no abstraction techniques available to support the modeler in reducing the complexity.

We argue that a scenario-based service model, as the one suggested on this paper, preserves the advantages of view-based service definitions. We have that different behavioral views can be unified in a way that they yields behavioral integration of the views while *preserving* each view. Only local changes must be made to achieve integration. Thus original view, and integrated views can still be related to each other. This provides smaller modeling artifacts, and hence an abstraction technique that suits the problem of process integration. The formal semantics of oclets yields a precise behavioral definition.

While the formalization of a scenario as an oclet and its formal semantics are available, the following questions remain to be answered for actually applying oclets for service integration: If two oclets specify behavior at different levels of granularity, how can a common granularity be achieved? What are the available options to integrate two given oclets of same granularity? Can these options

be computed automatically? How does the integration of two oclets affect the integration of other oclets? Given a set of integration options, are the unified oclets that realize the integration canonical? Are the integrated oclets consistent? Can inconsistencies be computed automatically?

We do not claim this list to be complete, but we consider answers to these questions to be fundamental for a systematic solution for process integration, whether using oclets or any other approach.

Acknowledgements We would like to thank Jan Mendling for bringing this topic to his attention and for the reviewers comments and suggestions for improving this paper. D. Fahland is funded by the DFG-Graduiertenkolleg 1324 "METRIK".

References

1. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Commun. ACM **44**(4) (2001) 71–77
2. Dijkman, R.M., Dumas, M.: Service-oriented design: A multi-viewpoint approach. Int. J. Cooperative Inf. Syst. **13**(4) (2004) 337–368
3. Grossmann, G., Ren, Y., Schrefl, M., Stumptner, M.: Behavior based integration of composite business processes. In: BPM'05. Volume 3649 of LNCS. (2005) 186–204
4. Schmitt, I., Saake, G.: A comprehensive database schema integration method based on the theory of formal concepts. Acta Inf. **41**(7-8) (2005) 475–524
5. Fahland, D., Woith, H.: Towards process models for disaster response. In: Workshops of the BPM'08, Milan, Italy (September 2008) LNBIP to appear.
6. Fahland, D.: Oclets - a formal approach to adaptive systems using scenario-based concepts. Informatik-Berichte 223, Humboldt-Universität zu Berlin (2008)
7. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data Knowl. Eng. **64**(1) (January 2008) 38–54
8. Lohmann, N., Kleine, J.: Fully-automatic Translation of Open Workflow Net Models into Human-readable Abstract BPEL Processes. In: Modellierung 2008. Volume P-127 of LNI., GI (March 2008) 57–72
9. ACM, C.: Special issue: Developing and integrating enterprise components and services. Commun. ACM **45**(10) (Oct. 2002)
10. Preuner, G., Conrad, S., Schrefl, M.: View integration of behavior in object-oriented design. Data Knowl. Eng. **36** (2001) 153–183
11. Shen, J., Grossmann, G., Yang, Y., Stumptner, M., Schrefl, M., Reiter, T.: Analysis of business process integration in web service context. Future Generation Comp. Syst. **23**(3) (2007) 283–294
12. Grossmann, G., Schrefl, M., Stumptner, M.: Classification of business process correspondences and associated integration operators. In: ER (Workshops). (2004) 653–666
13. Mendling, J., Simon, C.: Business process design by view integration. In: Business Process Management Workshops. (2006) 55–64

Does my service have unspecified behavior?

Kathrin Kaschner and Niels Lohmann

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{kathrin.kaschner, niels.lohmann}@uni-rostock.de

Abstract. Services are loosely coupled interacting software components. Since two or more services are usually composed to one software system, the behavior of an implemented service should not differ to its specification. Therefore we propose an approach to test, if the implementation contains unspecified behavior. Due to the interacting nature of services this is a nontrivial task.

1 Introduction

In the paradigm of service-oriented computing (SOC) [1], *services* are encapsulated, self-contained functionalities. Usually, they are not executed in isolation, but interact with each other via a well-defined interface. Thereby, the interaction between services follows complex *protocols* and goes far beyond simple remote-procedure calls (i. e., request/response operations). Arbitrary complex and possibly stateful interactions on top of asynchronous message exchange are common. Finally, a system can be *composed* by a set of services.

Best practices propose that systems should be *specified* prior to their implementation. A specification of a service (e. g., an abstract WS-BPEL process) describes its interface and the possible interactions with other services. Furthermore it already contains all relevant internal decisions of the control flow (e. g., the criteria whether or not a credit should be approved) and data about non-functional properties. In contrast, implementation-specific details (e. g., whether amounts are represented as integers or floats) are usually not yet defined.

Ideally, an implementation should *conform* to its specification. This means, all specified properties should be implemented. But it is obviously not always possible to verify the conformance in a formal proof. Furthermore verification is excessively elaborate and therewith often too expensive. Thus, it is customary to *test* the implementation.

As Dijkstra stated, program testing can be a very effective way to show the *presence* of bugs, but is hopelessly inadequate for showing their *absence*. To nevertheless be able to make a statement about the correctness of an implementation, a test suite with a significant number of test cases is necessary.

Testing services received much attention both in academia and industry. However, the communicating nature of services is often neglected. Instead of taking the new characteristics of SOC into account, existing work on testing classical software systems was slightly adapted to cope with the new languages used to implement services. Consequently, these approaches only take the internal

behavior of a service into account, but ignore the interactions with other services. Others in turn are restricted to stateless remote-procedure calls. Both might lead to undetected errors.

However, communication is an essential part of the specification of interacting services and consequently it is crucial to check it during conformance testing. That means, any interaction derived from the specification also has to be a correct interaction of the implementation *and* vice versa. In [2], we presented an approach to generate a test suite to test the former. In this paper, we propose how to test if undesired (i.e., unspecified) interactions are implemented in the service. Because the tester does not need knowledge about the code of the implementation, our approach is a *black-box testing* method.

This paper is organized as follows. Section 2 describes how a service implementation can be tested using partners. Section 3 recalls the approach to generate test cases and introduces an example protocol we use to demonstrate the approach. Section 4 provides the main contribution of this paper with the elaboration of test cases that are able to detect unspecified behavior of an implementation. Section 5 concludes the paper and gives directions to future work.

2 Testing interacting services

An adequate software test needs to check all specified properties. Due to the fact that protocols are an essential part of service specifications, they likewise have to be considered during service testing. A protocol describes in which state a service can send or receive which messages (and also which messages are not allowed in that state). In this way, all possible interactions with other services are defined which leads to an implicitly specified set $\mathcal{S}$ of all *partners*. Thereby a partner is again a service. If the specified service is implemented correctly, a partner always interacts deadlock-freely with it. In contrast *non-partners* violate the protocol; that is, during interaction the implemented service might deadlock. For this paper, we assume "correct interaction" between two services means deadlock freedom of their composition. We are well aware that there are other possibilities for defining "correct interaction". Nevertheless, deadlock freedom will certainly be art of any definition, so this paper can be seen as a step towards a more sophisticated setting.

We claim that like a function call is the most natural test case to test a function of a classical program, a partner service is likewise the most natural test case for a given service [2]. Thus, testing the protocol means testing whether the implemented service does not exclude specified partners, and also whether it does not include undesired partners. The former is important, because partner exclusion might have a financial impact if the service implements a business process — loosing a partner might result in deadlocks which again yields to down-time and contractual penalties. Testing the latter aims to exclude that additional behavior is implemented. For instance, an online shop should not terminate successfully if a customer did not pay for received goods. Consequently, we distinguish two kinds of test cases: services $P \in \mathcal{S}$ (partners), from which

we expect a deadlock-free interaction with the implementation, and services $Q \notin \mathcal{S}$ (non-partners), from which we expect the interaction does not terminate successfully (e.g., a deadlock will occur). Figure 1 illustrates our idea of exploiting the properties of the partners and non-partners for testing services. Ideally, the implementation contains exactly the specified behavior, that means, that $\mathcal{S}$ should be equal to the set $\mathcal{I}$ of correct interacting partners of the implementation ($\mathcal{S} = \mathcal{I}$). Otherwise, the implementation excludes a specified partner ($\mathcal{S} \backslash \mathcal{I} \neq \emptyset$) or the implementation is able to terminate successfully with a non-partner ($\mathcal{I} \backslash \mathcal{S} \neq \emptyset$).

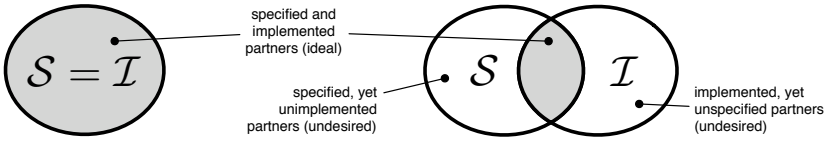


Fig. 1. Possible relations between the specified and implemented behavior $\mathcal{S}$ and $\mathcal{I}$.

To calculate $\mathcal{S}$ we can transform the specification into a formal model from which a description (operating guideline) of all partners can be generated [3]. Unfortunately, the set $\mathcal{I}$ cannot be calculated in the setting of black box testing. Thus, we need to test how $\mathcal{I}$ behaves with respect to $\mathcal{S}$.

The general test procedure for interacting services can be sketched as follows. We have a test suite containing the test cases $P \in \mathcal{S}$ and another one containing the test cases $Q \notin \mathcal{S}$. The service to be tested is deployed in a testing environment together with the two test suites. To process both test suites, the contained test cases are executed one after the other. Thereby, each test case interacts by message exchange with the service under test. The testing environment then is responsible for logging and evaluating exchanged messages. We thereby assume the test environment is able to detect whether the implementation terminates properly; for instance, whether a BPEL process instance has completed successfully.

A test run with a service $P \in \mathcal{S}$ fails if the test run terminates, but the implementation did not reach a final state. Then a deadlock has occurred and the implementation is incorrect. P can serve as a witness that the implementation excludes at least this specified partner service ($\mathcal{S} \backslash \mathcal{I} \neq \emptyset$). A test run that does not terminate, or terminates, but leaves the implementation in a final state, is inconclusive; that is, the run neither witnesses an error nor can proof overall correctness. Note that our approach is currently centered around deadlock freedom. In this setting, infinite runs are considered correct, because they do not contain a deadlock. Furthermore, the test environment cannot detect infinite runs and has to abort the test at some point, yielding an inconclusive result. But even if, for each partner service $P \in \mathcal{S}$, the run terminates and the implementation reaches a final state, we cannot conclude that the implementation does not exclude a specified partner service. We only know, that there *exists* a successful run for each specified partner. Since during testing the partner service (the tester) cannot control all internal decisions of the service. Hence, an incorrect part of the implementation may not executed during the test at all. However, a systematic

test approach with a significant number of test cases can increase the likelihood of detecting an error.

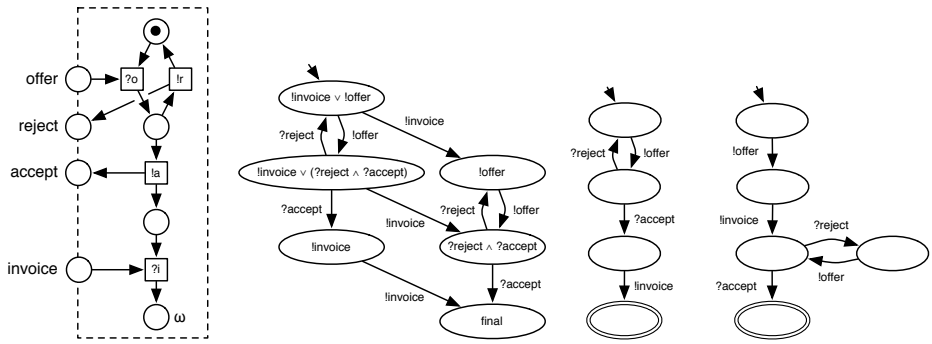
The set $\mathcal{S}$ contains a large number of many services (usually even infinitely many). It is therefore not practical to test a service with *every* $P \in \mathcal{S}$. In [2], we presented an approach how a small subset of $\mathcal{S}$ can be selected. With this reduced test suite, it is possible to detect all errors that could also be detected with the complete set $\mathcal{S}$.

Canonically, a test run with a non-partner service $Q \notin \mathcal{S}$ fails, if the test run terminates and the implementation is in a final state. Thereby, Q is a witness that the implementation supports undesired partner services ($\mathcal{I} \backslash \mathcal{S} \neq \emptyset$). Test runs of the composition of the implementation and the test case that neither terminate nor leave the implementation in a final state are inconclusive.

In Sect. 4 we will explain how the non-partner can be derived from the specification. This is a nontrivial task, because not all partner services $Q \notin \mathcal{S}$ are suitable for testing. As demonstrated by the following example, there exist some $Q' \notin \mathcal{S}$ which can fail even trough the implementation is correct.

3 Example

To formally reason about the specification of a service, an exact mathematical model is needed. We use *open nets* [4], a special class of Petri nets [5] as a formalism for protocols. Using existing translations, they can be easily derived from industrial languages such as WS-BPEL or BPMN. As running example for this paper, consider the open net from Fig. 2(a). It models a protocol of a buying service which receives and evaluate offers. It either accepts an offer and waits for an invoice, or rejects the offer and returns to its initial state. The final marking of this net is the marking $[\omega]$ which only marks the place ω: the control flow reached its end and all message channels are empty. Final markings distinguish desired end states from undesired deadlocks.



(a) specified protocol (b) operating guideline (c) test case (d) test case

Fig. 2. A buyer protocol modeled as an open Petri net (a). Its operating guideline (b) characterizes $\mathcal{I} \backslash \mathcal{S}$-test cases, for instance the seller services (d) and (e).

From this model, an *operating guideline* (OG) [3] can be calculated (see Fig. 2(b)). An operating guideline is a finite automaton whose edges are labeled with message events (sending events are preceded by ! and receiving events are preceded by ?) and whose states are annotated with Boolean formulae. These formulae express which edges of a partner must be present to guarantee deadlock freedom. For example, "!invoice $\vee$!offer" expresses the requirement that every partner has to initially send an invoice or an offer, or make a decision between both during runtime. Likewise, "?reject $\wedge$?accept" states that the receipt of *both* messages must be possible in that state. The operating guideline finitely characterizes the infinite set of all deadlock-free interacting partners $\mathcal{S}$ of the buying service and hence all test cases for the specified protocol.

Two test cases are depicted in Fig. 2(c) and 2(d). While the first test case sends the invoice only after receipt of an acceptance message, the second exploits asynchronicity and sends the invoice right after sending the offer. In both cases, the OG's annotations are fulfilled.

4 Testing for unspecified behavior

As mentioned earlier, the operating guideline can used to generate the test suite containing all necessary partners the test if $\mathcal{S}\backslash\mathcal{I} = \emptyset$ holds. To be able to make a statement about these $\mathcal{I}\backslash\mathcal{S}$ deviations (i.e., unspecified partners), a different approach is needed. This is due to the fact that a service that is not characterized by the does not *necessarily* yield a deadlock. For example, a service without the ability to receive a rejection message (see Fig. 3(a)) may deadlock with the buyer service if the latter rejects the offer. If, however, the offer is accepted, no deadlock occurs. Hence, any result of this test case would be inconclusive.

To this end, the operating guideline of the specified protocol cannot be used to derive $\mathcal{I}\backslash\mathcal{S}$-test cases. To characterize all services that are expected to deadlock when composed to the implementation, we need to reinterpret the final states of the protocol: If we complement the set of final markings, then the operating guideline of this *anti-protocol* (called the *anti operating guideline*) characterizes partners that interact without deadlock with the anti-protocol. That is; the interaction terminates in a final state of the anti-protocol. This in turn is a deadlock of the original protocol. Hence, the anti operating guideline characterizes partners that are expected to deadlock with the original protocol.

For the example of Fig. 2(a), any marking but $[\omega]$ would be a final marking of the anti-protocol. Note that the structure of the anti-operating guideline of Fig. 2(b) is very similar to the original operating guideline (cf. Fig. 3(b)) and we see that the interactions that would lead to the a final state in Fig. 2(b) (e.g. !offer ?accept !invoice) lead to the only state without final in the annotation in Fig. 3(b).

Three test cases characterized by the anti operating guideline are depicted in Fig. 3(c)–3(e): the first two exploit the fact that premature termination will lead to a deadlock with a conformant implementation, whereas the last test case avoids a valid final marking by sending another offer.

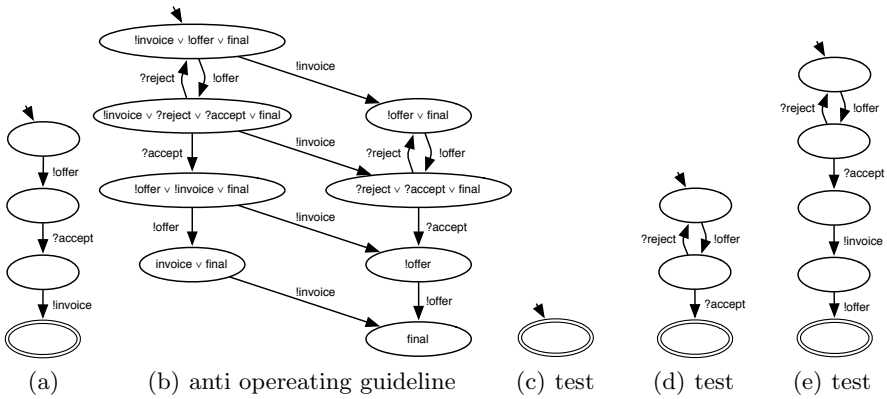


Fig. 3. The service (a) is not characterized by the operating guideline (cf. Fig. 2(b)), but does not necessarily deadlock with the protocol from Fig. 2(a). The anti operating guideline (b) for the protocol characterizes $\mathcal{S}\backslash\mathcal{I}$-test cases. For example, communication should deadlock if a seller sends no message (c), aborts the interaction prematurely (d) or sends too many messages (e).

5 Related Work

Several works exist to systematize testing of Web services (see [6] for an overview). Test case generation can be tackled using a variety of approaches such as control flow graphs [7], model checking [8], or XML schemas [9]. These approaches mainly focus on code coverage, but do not take the interacting nature of Web services into account. In particular, internal activity sequences are not necessarily enforceable by a test case. Therefore, it is not clear how derived test cases can be used for black-box testing in which only the interface of the service under test is available.

To the best of our knowledge, none of the existing testing approaches take stateful business protocols implemented by Web services into account, but mainly assume stateless remote procedure calls (i. e., request-response operations), see for instance [9]. Bertolino et al. [10] present an approach to generate test suites for Web services and also consider nonfunctional properties (QoS). Their approach, however, bases on synchronous communication and is thus not applicable in the setting of SOAs.

Finally, several tools such as the Oracle BPEL Process Manager [11] or Parasoft SOAtest [12] support testing of services, but do not specifically focus on the business protocol of a service. In contrast, Mayer et al. [13] underline the importance of respecting the business protocol during testing, but do not support the generation of test cases in their tool BPELUnit.

6 Conclusion

We presented an approach to generate test cases from a specification protocol and show how they can be used to check whether an implementation introduces unspecified behavior. This test suite is not complete in the sense that we cannot

detect all deviations from the specification. The reason for this lies in the nature of services in which decisions may not always be enforceable by the environment. To this end, we introduced anti operating guidelines to characterize $\mathcal{S}\backslash\mathcal{I}$ test cases that must deadlock when composed to a compliant implementation. The results of this paper are defined in terms of open nets and are independent of a concrete protocol specification language. With existing translation, e.g. from BPEL to open nets and vice versa [14, 15], the approach is easily applicable to industrial languages.

In future work, we plan to propose a test case coverage criterion to select a small number of test cases which still can detect the same errors that are detectable with the complete set of test cases. Indications are that this criterion should be similar to the criterion we proposed for $\mathcal{I}\backslash\mathcal{S}$ test cases [2]. Finally, the test case selection procedures need to be implemented to validate the test suite generation with industrial case studies.

References

1. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Communications of the ACM **44**(4) (2001) 71–77
2. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC Workshops 2008. LNCS, Springer (2009)
3. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341
4. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
5. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
6. Baresi, L., Nitto, E.D., eds.: Test and Analysis of Web Services. Springer (2007)
7. Yan, J., Li, Z., Yuan, Y., Sun, W., Zhang, J.: BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In: ISSRE 2006, IEEE (2006) 75–84
8. García-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for BPEL compositions of Web services using SPIN. In: WS-MaTe 2006. (2006) 83–94
9. Hanna, S., Munro, M.: An approach for specification-based test case generation for Web services. In: AICCSA, IEEE (2007) 16–23
10. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: TESTCOM/FATES 2008. LNCS 5047, Springer (2008) 266–282
11. Oracle: BPEL Process Manager. (2008) `http://www.oracle.com/technology/products/ias/bpel`.
12. Parasoft: SOAtest. (2008) `http://www.parasoft.com`.
13. Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. In: TAV-WEB '06, ACM (2006) 33–42
14. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007. Volume 4937 of LNCS., Springer (2008) 77–91
15. Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In: Modellierung 2008. Volume P-127 of LNI., GI (2008)

A Finite Representation of all Substitutable Services and its Applications

Jarungjit Parnjai[1,⋆], Christian Stahl[1 2,⋆⋆], and Karsten Wolf[3,⋆⋆⋆]

[1] Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{parnjai, stahl}@informatik.hu-berlin.de
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[3] Universität Rostock, Institut für Informatik
18051 Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. We present a finite representation of all *substitutable* services P' of a given service P. We show that our approach can be used for at least two applications: (1) given a finite set of services $\mathcal{P} = \{P_1, ..., P_n\}$, we provide a representation of all services P' that can substitute every $P_i \in \mathcal{P}$, and (2) given a service P'' that cannot substitute a service P, we find the most similar service P^* to P'' that can substitute P.

1 Introduction

The paradigm of Service-Oriented Computing (SOC)[1] uses a service as a building block for designing flexible business processes by means of service composition. The behavior of a service is subject to changes. Driven by the cost and time to meet the deadline, a new version of a service is hardly reconstructed from scratch. Instead, a service will be *substituted* by a new version, which can be derived by updating its current functionality or adding in a new functionality.

We consider a service P to be *substitutable* by a service P' if P' cooperates deadlock-freely with every partner that P cooperates deadlock-freely with. That is, substituting P by P' *preserves* every deadlock-free cooperating partner of P.

In this paper, we present an *operating guideline* approach to represent the set of all substitutable services. This representation is helpful for at least two applications. Given a finite set of services $\mathcal{P} = \{P_1, ..., P_n\}$, we show that a finite representation of all services P' that can substitute every $P_i \in \mathcal{P}$ can be computed with the help of operating guidelines [2, 3] of services. Furthermore, we show that errors in a non-substitutable service can be corrected automatically using a simulation-based graph edit distance as introduced in [4].

⋆ Funded by the DFG-Graduiertenkolleg 1324 "METRIK".
⋆⋆ Funded by the DFG project "Substitutability of Services" (RE 834/16-1).
⋆⋆⋆ Supported by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

The remainder of this paper is organized as followed. Section 2 recalls some formalisms and substitutability notion. Section 3 presents a finite representation of all services P' that can substitute a given service P. Section 4 outlines a method to compute a finite representation of all services P' that can substitute all services $P_i \in \{P_1, ..., P_n\}$ that are given. Section 5 shows how to correct errors in a non-substitutable service P'' with respect to a given service P. Finally, Section 6 concludes the paper.

2 Background

We model the behavior of a service P with a *service automaton*. A service automaton is a finite state automaton with a set Q of states, a set $F \subseteq Q$ of final states, an initial state $q_0 \in Q$, a set I of input interfaces, a set O of output interfaces (I and O are pairwise disjoint), and a *non-deterministic* transition relation $\delta \subseteq Q \times \{I \cup O \cup \{\tau\}\} \times Q$. The edges are labeled with output message $x \in O$ sent to (labeled "!x") the environment, or input message $x \in I$ received from (labeled "?x") the environment, or internal move (label "τ"). A non-final state with no outgoing transition is called a *deadlock*.

Given two service automata P and R, their composition $P \oplus R$ is a service automaton in which its set of states is the cartesian product of Q_P, Q_R, and the set of all multisets of pending messages between P and R. We assume that two composable service automata have compatible interfaces ($I_P = O_R$ and $I_R = O_P$), but all other constituents are pairwise disjoint. The composition $P \oplus R$ is deadlock-free if $P \oplus R$ does not contain a deadlock. R is a *strategy* of P iff $P \oplus R$ is deadlock-free. The strategy relation is symmetric, that is, R is a strategy of P implies P is also a strategy of R. We write $Strat(P)$ to denote the set of all strategies of P. See [2] for further details.

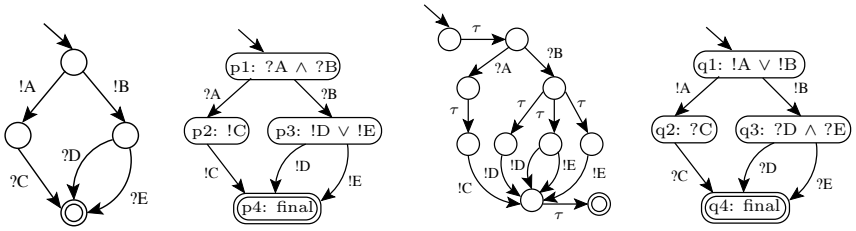
Throughout this paper, we assume a deadlock-free composition, i.e., there always exists at least one strategy for a given service.

An operating guideline $OG(P)$ of P is a deterministic service automaton S^ϕ where each state q of S is annotated with a Boolean formula $\phi(q)$. A matching relation between states of a service automaton and S^ϕ are used to characterize a set of service automata. We write $Match(S^\phi)$ to denote the set of all service automata S' that satisfies a matching relation with S^ϕ. $OG(P)$ characterizes the (possibly infinite) set of all strategies of P, i.e., $Match(OG(P)) = Strat(P)$ [2].

Figure 1(a) depicts a service automaton P_1 and Fig. 1(b) depicts an operating guideline of P_1.

We define our substitutability notion called *accordance*. A service P' substitutes a service P under accordance (P' accords with P) iff every strategy of P is also a strategy of P', i.e., $Strat(P) \subseteq Strat(P')$. We assume that P and P' are *interface equivalent* ($I_P = I_{P'}$ and $O_P = O_{P'}$) and write $Accord(P)$ to denote the (possibly infinite) set of all services P' that substitute P under accordance.

[3] presents an algorithm to decide whether P' substitutes P under accordance using their operating guidelines.



(a) P_1 (b) $OG(P_1)$ (c) $MS(P_1)$ (d) $OG(MS(P_1))$

Fig. 1. (a) service automaton P_1, (b) an operating guideline of P_1, (c) *MS* of P_1, and (d) an operating guideline of *MS* of P_1.

3 Representing All Substitutable Services

Given a service P we show how to calculate an operating guideline that represents the set $Accord(P)$ of all services P' that can substitute P under accordance.

Definition 1 (Maximal Strategy, *MS*). *Let P be a service automaton and $OG(P) = S^{\phi}$ be its operating guideline. A* maximal strategy *of P, denoted $MS(P)$, is obtained from S by replacing every node q by a non-deterministic internal choice between all the valid combinations of outgoing edges from q w.r.t. satisfying assignment in $\phi(q)$.*

Figure 1(c) depicts a maximal strategy $MS(P_1)$ of P_1 (in Fig. 1(a)) and Fig. 1(d) depicts its operating guideline $OG(MS(P_1))$.

$MS(P_1)$ is obtained from the underlying service automaton of $OG(P_1)$ according to Definition 1. For example, the node p_3 of the underlying service automaton of $OG(P_1)$ is replaced by the non-deterministic τ choice between three valid combinations of outgoing transitions that satisfy assignment in $\phi(p_3) = !D \vee !E$ in $OG(P_1)$. These three combinations are (1) a transition labeled $!D$, (2) a transition labeled $!E$, and (3) two transitions labeled $!D$ and $!E$. Clearly, $MS(P_1)$ is a strategy of a service P_1. That is, $MS(P_1) \in Match(OG(P_1))$.

Mooij and Voorhoeve [5] have proven that for a strategy R of P, each strategy of $MS(P)$ is also a strategy of R.

Proposition 1 ([5]). *Let P be a service automaton such that $Match(OG(P)) \neq \emptyset$. Then for all $R \in Match(OG(P))$ holds: $Strat(MS(P)) \subseteq Strat(R)$.*

By the help of Proposition 1 we prove that $OG(MS(P))$ represents the set $Accord(P)$ of all service automata P' that can substitute a service automaton P under accordance.

Theorem 1 (Characterizing all substitutable services). *Let P and P' be two service automata. Let $OG(P)$ be an operating guideline of P. Then, P' substitutes P under accordance iff $P' \in Match(OG(MS(P)))$.*

Proof. We will show that $Accord(P) = Match(OG(MS(P)))$.

Consider $Accord(P) = \{P' \mid Strat(P) \subseteq Strat(P')\}$. Since the strategy relation is a symmetric relation, we conclude that $Accord(P) = \{P' \mid \forall R \in Strat(P) : P' \in Strat(R)\} = \bigcap_{R \in Strat(P))} Strat(R)$. Since $Match(OG(P)) = Strat(P)$, $Accord(P) = \bigcap_{R \in Match(OG(P))} Strat(R)$ follows.

Next, we will show that $\bigcap_{R \in Match(OG(P))} Strat(R) = Strat(MS(P))$. We know $MS(P) \in Strat(P)$ and $Strat(P) = Match(OG(P))$. Therefore, we can conclude that $\bigcap_{R \in Match(OG(P))} Strat(R) \subseteq Strat(MS(P))$. Proposition 1 asserts that for all $R \in Match(OG(P))$ holds: $Strat(MS(P)) \subseteq Strat(R)$. Therefore, we can conclude that $\bigcap_{R \in Match(OG(P))} Strat(R) \supseteq Strat(MS(P))$. Consequently, $\bigcap_{R \in Match(OG(P))} Strat(R) = Strat(MS(P))$ immediately follows.

We know $Strat(MS(P)) = Match(OG(MS(P)))$. Thus, we can conclude that $\bigcap_{R \in Match(OG(P))} Strat(R) = Match(OG(MS(P)))$.

Consequently, $Accord(P) = Match(OG(MS(P)))$. □

Theorem 1 shows that the operating guideline $OG(MS(P))$ is a finite representation of all P' that can substitute P under accordance.

Our result enables a service designer to effectively derive P' from $OG(MS(P))$. Clearly, P' can substitute P under accordance, as it matches with $OG(MS(P))$. The designer can also use P' as a template to tailor a new version P'' by filling P' with some internal actions. This way, it can be decided if P'' substitutes P under accordance by checking if $P'' \in Match(OG(MS(P)))$.

With our approach, we can also decide accordance (as presented in [3]) of two services P'' and P by checking if $P'' \in Match(OG(MS(P)))$.

4 Conjoining Substitutable Services

Suppose a service designer would like to design a new service which can support all potential customers of both a hotel booking service and a flight booking service. The representation of all services that can substitute both booking services is helpful for the designer. With this representation, the designer can decide whether such a new service does exist, and in case it does, a well-suited upgrade of a new service can be derived immediately from such a representation.

For a finite set $\mathcal{P} = \{P_1, .., P_n\}$ of service automata, we show that the intersection $\bigcap_{P_i \in \mathcal{P}} Accord(P_i)$ of sets of all services that accord with every P_i can be represented by the product of all operating guidelines of maximal strategy $MS(P_i)$ of P_i, where $P_i \in \mathcal{P}$.

The product of two operating guidelines [3] is defined as an operating guideline that characterizes the intersection of all service automata that match with these two operating guidelines. The product of two operating guidelines assumes that both operating guidelines are interface equivalent.

Proposition 2 ([3]). *Let $OG_{\otimes} = OG(S_1) \otimes OG(S_2)$ be the product of operating guidelines $OG(S_1)$ and $OG(S_2)$, Then, $Match(OG_{\otimes}) = Match(OG(S_1)) \cap Match(OG(S_2))$.*

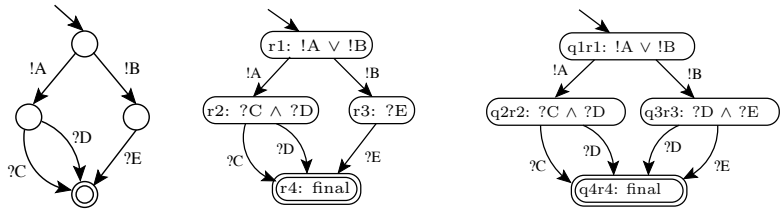
Corollary 1 (Characterizing intersection of substitutable services). *Let P_1 and P_2 be two service automata. Let $OG(P_1)$ be an operating guideline of P_1 and $OG(P_2)$ be an operating guideline of P_2. Then,*

$$Match(OG(MS(P_1)) \otimes OG(MS(P_2))) = Accord(P_1) \cap Accord(P_2).$$

Proof. Follows from Proposition 2 and Theorem 1. □

Corollary 1 shows that we can use the product of operating guidelines to compute the finite representation of all services that accords with both P_1 and P_2. In case the returned product describes an empty set, there is no service automaton P' that can substitute both P_1 and P_2 under accordance.

Since the product $\otimes$ of operating guidelines is commutative and associative, the result from Corollary 1 can be easily generalized to the product of any finite number n of operating guidelines of $MS(P_i)$, where $P_i \in \{P_1, .., P_n\}$.



(a) P_2 (b) $OG(MS(P_2))$ (c) $OG_\otimes$

Fig. 2. (a) service automaton P_2, (b) an operating guideline of MS of P_2, and (c) the product $OG_\otimes$ of $OG(MS(P_1))$ and $OG(MS(P_2))$.

Figure 2(c) depicts $OG_\otimes$ as a finite representation of all services that can substitute both P_1 (Fig. 1(a)) and P_2 (Fig. 2(a)) under accordance. $OG_\otimes$ is the synchronous product of $OG(MS(P_1))$ and $OG(MS(P_2))$, where each node is annotated with the conjunction of the two Boolean formulas of the corresponding states of $OG(MS(P_1))$ and $OG(MS(P_2))$. For example, the node $q2r2$ in $OG_\otimes$ is annotated with $\phi(q2r2) = ?C \wedge ?D$, which is the conjunction of $\phi(q2) = ?C$ in $OG(MS(P_1))$ and $\phi(r2) = ?C \wedge ?D$ in $OG(MS(P_2))$.

5 Correcting Non-Substitutable Services

Suppose a service designer has designed an ill-suited upgrade of a travel agency service that does not accord with the travel agency service. Synthesizing a new well-suited upgrade of the service using an approach proposed in Section 3 may not be sufficient, as the well-suited upgrade might be very different and totally ignore the structure of its ill-suited upgrade version. The designer might prefer to reuse an ill-suited upgrade of the service instead of synthesizing a new well-suited upgrade of the service.

To reuse an ill-suited upgrade of the service, the errors found in the ill-suited upgrade can be fixed manually. Nevertheless, the manual correction is a tedious and error-prone procedure. This scenario motivates a method to synthesize a well-suited upgrade of the service automatically from its ill-suited upgrade.

Given two service automata P and P'' where P'' does not accord with P, we propose a procedure to correct P'' with respect to P. The errors in P'' can be detected and corrected automatically using a simulation-based graph edit distance, as introduced in [4] to fix a faulty service to cooperate deadlock-freely in a choreography. The approach takes P'' and $OG(MS(P))$ as its input, computes the most similar service automaton P^* to P'' such that $P^* \in Match(OG(MS(P)))$, and returns the edit actions that are necessary to transform P'' into P^*. Clearly, P^* can substitute P under accordance, as it matches with $OG(MS(P))$. That is, P^* cooperates deadlock-freely with every strategy of P. This way, P'' can be reused, as P^* is most similar to P'', yet accords with P.

So far the simulation-based graph edit distance approach is applicable only for acyclic and deterministic services [4].

6 Conclusion

We have proposed an approach to characterize the set $Accord(P)$ of all services P' that can substitute a service P under accordance. We have shown that a finite representation of $Accord(P)$ can be computed using the concept of a maximal strategy [5] and its operating guideline [2]. With this representation, we can decide accordance of two services and derive from it a new service that accords with a given service.

We have shown two applications of our approach. Given a finite set of services $\mathcal{P} = \{P_1, ..., P_n\}$, we provide a representation of the intersection of $Accord(P_i)$ for all $P_i \in \mathcal{P}$ with the help of the product of operating guidelines [3]. For a service P'' that cannot substitute a service P, we provide an automatic correction procedure to transform P'' into the most similar P^* such that P^* accords with P with the help of the simulation-based graph edit distance [4].

References

1. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (2007)
2. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: ICATPN 2007. Volume 4546 of LNCS., Springer-Verlag (2007) 321–341
3. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. LNCS ToPNoC **II**(5460) (2008) 172–191
4. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In Dumas, M., Reichert, M., eds.: BPM 2008. Volume 5240 of LNCS., Springer-Verlag (2008) 132–147
5. Mooij, A.J., Voorhoeve, M.: Proof techniques for adapter generation. In Bruni, R., Wolf, K., eds.: WS-FM 2008 Milan, Italy, Proc. LNCS, Springer-Verlag (2008)

Prozessunterstützung für temporäre, ehrenamtliche und private Gruppen

Daniel Schulte

FernUniversität in Hagen, 58084 Hagen, Germany,
Daniel.Schulte@FernUni-Hagen.de

Zusammenfassung Der Bedeutung von Geschäftsprozessen wird in der Dienstorientierung (*Service-oriented Computing*) durch Lösungen zur Automatisierung und Unterstützung bspw. mit WS-BPEL Rechnung getragen. Da allerdings ehrenamtlichen und privaten Gruppen weder entsprechendes Wissen noch notwendige Soft- und Hardwareprodukte zur Verfügung stehen, können sie derartige Prozessunterstützungen nicht für ihre Einsatzzwecke nutzen und setzen statt dessen weiterhin insbesondere auf E-Mails zur Koordination ihrer Prozesse. Diese Arbeit ermittelt daher die speziellen Anforderungen solcher Gruppen an eine Prozessunterstützung und stellt einen entsprechenden dienstorientierten Architekturstil vor.

1 Einführung

Computer und Internet unterstützen Anwender nicht nur bei Berechnungen und Informationsbeschaffung. Zusätzlich werden sie heute oft zur Koordination ganzer Prozesse in der Geschäftswelt aber auch in ehrenamtlich agierenden Gruppen wie z. B. Vereinen und im privaten Umfeld genutzt. Während die Steuerung von Prozessen in der Geschäftswelt bereits vielfältig adressiert und bspw. in dienstorientierten Architekturen mit WS-BPEL unterstützt wurde, sind bisherige Lösungen kaum für ehrenamtliche oder private Gruppen geeignet, da diese anderen Rahmenbedingungen unterliegen. Solche Gruppen nutzen bisher meist lokal installierte Software und E-Mails für ihre Prozesse. So werden initialer Einrichtungsaufwand, Einarbeitung und mögliche Kosten vermieden, aber Medienbrüche, unkontrollierte Informationsweitergabe, Verlust einzelner Informationen, mehrfache Datenhaltung, Dateninkonsistenz und Verlust der Prozessübersicht in Kauf genommen. Erfolgreiche Prozesse sind nur schwer wiederholbar, da Wissen über sie nur implizit vorhanden ist und somit ständig rekonstruiert werden muss. Kollaborative integrierte Prozessschritte wie bspw. das Sammeln von Tagesordnungspunkten sind nur durch erheblichen manuellen Aufwand realisierbar. Dabei machen vereinzelte Lösungen wie Konferenzmanagementsysteme deutlich, dass auch außerhalb des klassischen industriellen Umfeldes Prozessunterstützung hilfreich sein kann, um aufwendige und fehleranfällige händische Koordinationen zu vermeiden. Aufgrund unterschiedlicher Voraussetzungen sind aber Techniken aus dem Unternehmensumfeld nicht ohne weiteres für ehrenamtliche und private Gruppen adaptierbar.

Diese Arbeit analysiert daher in Abschnitt 2 die Anforderungen dieser Gruppen an eine Prozessunterstützung und leitet einen entsprechenden dienstorientierten Lösungsansatz her, der verschiedenen verteilten Diensten erlaubt, gezielt mit Anwendern in Kontakt zu treten. In Abschnitt 3 wird diese Lösung in Form eines Architekturstils präzisiert und in Abschnitt 4 eine ausstehende Fallstudie motiviert. Die Betrachtung verwandter Arbeiten in Abschnitt 5 und ein kurzer Ausblick in Abschnitt 6 beschließen diese Arbeit.

2 Prozesse für Gruppen

Prozesse von Gruppen, die nur vorübergehend existieren, deren Mitglieder nur temporär diesen Gruppen oder deren Mitglieder zugleich mehreren anderen unabhängigen Gruppen angehören (z. B. ehrenamtlich agierende Vereine, Veranstaltungen mit temporären Organisationsteams wie Benefizkonzerte, aber auch Workshops oder Konferenzen) haben gemein, dass sie primär aus manuellen Arbeitsschritten (Aufgaben genannt) bestehen, die von entsprechenden Mitarbeitern zu bearbeiten sind. Typische Rahmenbedingungen sind dabei:

- Es ist kein IT-Fachpersonal und oft nur private Hardware verfügbar.
- Es sind weder Zeit noch Geld für den initialen Aufbau einer technischen Infrastruktur verfügbar.
- Gruppen ändern sich beständig in ihrer Zusammensetzung oder arbeiten nur kurze Zeit zusammen.
- Gruppenbildung und Verantwortlichkeiten innerhalb dieser können auf Vereinstrukturen basieren, von Initiatoren bspw. im Rahmen von Workshops vorgegeben werden, oder auch z. B. bei Interessensgemeinschaften frei verhandelbar sein.
- Gruppenmitglieder arbeiten parallel an Projekten anderer Gruppen.
- Prozesse werden nur selten in einer Gruppe wiederholt, aber ähnliche Prozesse finden in vielen Gruppen statt (z. B. jährliche Mitgliederversammlungen).
- Prinzipielle Prozessabläufe sind kein Geschäftsgeheimnis.

Einige Forderungen an eine Prozessunterstützung sind demnach:

- Konzentration auf freie Technologien, Infrastrukturen, Dienste und Prozesse für weitestgehend kostenlose Lösungen
- Leicht formulier- und anpassbare Prozesse und Dienste; Ausnahmesituationen können beim Auftreten dynamisch gelöst werden
- Leichte Anwendung (und individuelle Anpassung) von Standardprozessen in verschiedenen Kontexten
- Einfacher und automatisierter Bezug von Aufgaben
- Unterstützung von Aufgaben verschiedener Art und Komplexität (von einfachen Überprüfungen bis hin zu komplexen, kooperativ zu bearbeitenden Aufgaben)
- Flexible Verwaltung von Gruppenmitgliedern und Verantwortlichkeiten

Insbesondere der einfache Zugang von Anwendern zu ihren Aufgaben soll in diesem ersten Schritt einer Lösungsentwicklung betrachtet werden. Denn — so weit vorhanden — setzen bisherige Lösungen wie Konferenzmanagementsysteme darauf, dass Anwender bestimmte Dienstseiten oder Portale regelmäßig aufsuchen, um sich über anstehende Aufgaben zu informieren. Dies weist aber zwei wesentliche Probleme auf: (1.) Der Anwender ist in einer Holschuld, er muss aktiv die entsprechenden Informationen über ausstehende Aufgaben einholen. Bei einer wachsenden Anzahl genutzter Dienste und einem zudem nur sporadischen Auftreten der Aufgaben ist eine zeitnahe Kenntnisnahme anstehender Aufgaben daher schwierig, so dass bisher oft ergänzend E-Mails eingesetzt werden. (2.) Die Verwendung von E-Mails kann zwar sicherstellen, dass Anwender zeitnah von Aufgaben erfahren, letztendlich werden aber weder die zeitliche Einplanung und Bearbeitung noch die Weiterverfolgung von Aufgaben nebst zugehörigen Prozessen unterstützt.

Die wichtigsten Anforderungen in diesem ersten Lösungsschritt lassen sich daher zunächst auf folgende Punkte reduzieren:

- Es darf — auch wenn Prozesse und Dienste mehrerer Anbieter (*Service Provider*) genutzt werden — nur einen Einstiegspunkt für den Anwender geben, an dem ihm alle wesentlichen Informationen zu ausstehenden Aufgaben und zur Verfolgung von Prozessen (ohne sein aktives Eingreifen) zur Verfügung gestellt werden.
- Dieser Einstiegspunkt muss einen flexiblen Zugriff auf zur Aufgabenausführung notwendige unterstützende Dienste und Informationen bieten.
- Dieser Einstiegspunkt muss von Prozessen und Diensten mit aktuellen Informationen, z. B. zum Status einer Aufgabe, versorgt werden können.

Die oft verwendeten E-Mails bieten durch ihre i. d. R. personenbezogene Adressierung zwar einen zentralen Einstiegspunkt, eignen sich aber nicht zur übersichtlichen Aufgabenverwaltung und -darstellung und erlauben zudem keine komfortablen Aktualisierungen bspw. des Aufgabenstatuses. Die Verwendung von Portalen und proprietären Lösungen kann zwar eine gute Aufgabenverwaltung bieten, führt aber wieder zu dem Problem multipler Einstiegspunkte. Ein E-Mail-ähnlicher Ansatz mit strukturierten Informationen und einer dienstorientierten Schnittstelle kann beide Vorteile zusammenführen und personenbezogene Aufgabenzuweisungen aber auch Aktualisierungen von Informationen zu Aufgaben aus beliebigen Prozessen und Diensten heraus unterstützen.

Diese Idee wird im Folgenden in Form eines *Aufgabenlistendienstes* aufgegriffen, der über eine eindeutige, einem Anwender zugeordnete URI adressierbar ist und erlaubt, über im Detail noch zu entwickelnde Dienste Aufgaben inkl. Metainformationen wie Deadline, Status, o. ä. einem Anwender zuzuordnen. Der Anwender kann über diesen Dienst auf seine aktuellen Aufgaben zugreifen und diese verwalten, und sie bspw. auch nach Metadaten sortieren. Die Dienstorientierung erlaubt dabei sowohl auf Seiten des Dienstanbieters und des Prozesses als auch auf Seiten von Klientanwendungen für den Anwender flexible und innovative Lösungen unabhängig von verwendeten Programmiersprachen u. ä. Der nächste Abschnitt wird nun einen Architekturstil vorstellen, der dieser Idee entspricht.

3 Architekturstil

Nach Fielding [4] ist ein Architekturstil eine Menge von architektonischen Bedingungen, die die Rollen und Funktionen architektonischer Elemente und die erlaubter Beziehungen zwischen diesen für eine konforme Architektur einschränken. Ein Architekturstil für den Ansatz aus Abschnitt 2 wird in Abbildung 1 skizziert.

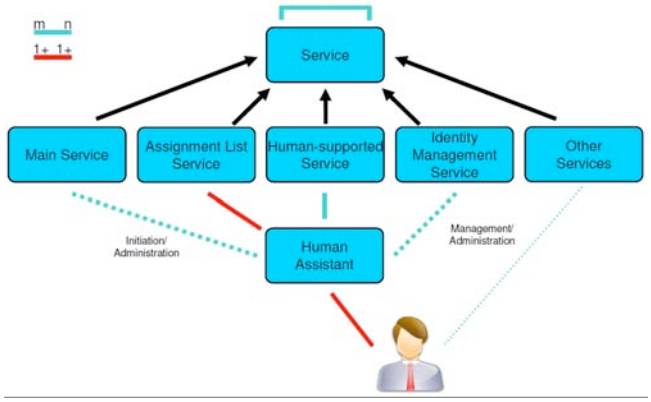


Abbildung 1. Architekturstil

Anwender werden durch eindeutige URIs, unter denen ein Aufgabenlistendienst (*assignment list service*) neu zugewiesene Aufgaben sowie Aufgabenaktualisierungen entgegennimmt, zu adressierbaren Entitäten (ähnlich einer E-Mail-Adresse nebst Postfach). Dienste, die den Eingriff von Anwendern benötigen (*human-supported services*), informieren diese per Aufgabenlistendienst über ausstehende Aufgaben und geben dabei neben einigen Metainformationen u. a. eine eindeutige URI für den Aufruf einer Repräsentation der Aufgabe an. Um größtmögliche Flexibilität bei der Aufgabenrealisierungen zu ermöglichen, werden Daten über potentielle Nutzer (z. B. bei der Dienst-Initialisierung übergeben) und interne Zustände von den Diensten eigenverantwortlich verwaltet. Eine Auslagerung dieser Funktionalitäten ist in Form weiterer Dienste oder Entwicklungsframeworks optional denkbar.

Zusätzliche Identitätsmanagementdienste (*identity management services*) erlauben eine späte, dynamische Bindung von Rollen an konkrete Anwender und eine Wiederverwendung von Organisationsstrukturen und Anwenderinformationen. Selbst Randbedingungen wie Urlaub oder Arbeitsauslastung ließen sich bei dieser dynamischen Bindung berücksichtigen. Den Diensten werden dann statt Adressen von Anwendern Rollenbezeichungen und die Adresse eines Identitätsmanagementdienstes übergeben, sodass erst bei Auftreten einer Aufgabe die konkreten Anwenderadressen ermittelt und genutzt werden.

Lokale und netzbasierte Klienten (*human assistant*) erlauben den Anwendern den komfortablen Zugriff auf die sie betreffenden Aufgaben, und können bspw. durch Kalenderintegration zur Bearbeitungsplanung Mehrwerte bieten.

Für den praktischen Einsatz ist dieser Stil u. a. um standardisierte Schnittstellen für Identitätsmanagementdienste und Aufgabenlistendienste zu konkretisieren, um gemäß o. g. Anforderungen anbieterübergreifend arbeitende Lösungen zu ermöglichen. Über alle andere Dienste und Klienten sind dagegen möglichst wenig Annahmen zu machen, um flexible und innovative Realisierungen zu ermöglichen.

4 Fallbeispiel

Folgendes, noch umzusetzendes, einfaches Fallbeispiel ist zur Demonstration der Realisierbarkeit geplant: Für die Vorbereitung einer Sitzung, z. B. eines Mitarbeitertreffens eines Vereines oder einer öffentlichen Versammlung einer Bürgerbewegung, wird ein Prozess nebst notwendigen Diensten entwickelt werden, der das gemeinsame Sammeln von Tagesordnungspunkten und das Versenden von Einladungen zur Sitzung unter Angabe der Tagesordnung unterstützt. Der Prozess besteht aus folgenden Schritten:

1. Rahmendaten wie Datum, Raum, Sitzungsleitung, etc. sind zu spezifizieren (demonstriert die Initialisierung einer Prozessinstanz mit anwenderspezifizierten Daten inkl. Benennung von potentiellen Anwendern als Sitzungsleitung, etc. und die Konfiguration der Instanz z. B. bzgl. eines optionalen Anmeldeschrittes).
2. Tagesordnungspunkte sind zu sammeln (Beispiel für eine kooperative Aufgabe; wenn Anwender nicht gegenseitig ihre Beiträge zur Sammlung einsehen können, erhöht sich die Wahrscheinlichkeit von Mehrfachnennungen signifikant, einzelne Punkte können aber auch leichter vergessen werden).
3. Wenn der Sitzungsleitung nicht alle potentiellen Teilnehmer bekannt sind, ist ggf. eine Anmeldung z. B. über ein Webschnittstelle zu ermöglichen (Beispiel einer dynamisch wachsenden Gruppe).
4. Tagesordnung ist zu genehmigen/überprüfen (einfache Aufgabe).
5. Einladungen mit Tagesordnung sind an alle Teilnehmer zu versenden (klassischer, automatisierter Dienst).

Der Einsatz eines Identitätsmanagementdienstes ist optional zu ermöglichen, um das Potential der Wiederverwendung von Anwenderinformationen insbesondere für temporär stabile Gruppen aufzuzeigen. Gleichzeitig kann aber der Einsatz mit dem Beispiel einer Bürgerbewegung auch für dynamisch wachsende Gruppen demonstriert werden. Darüber hinaus wird die Fallstudie zeigen müssen, dass etablierte Lösungen wie z. B. YAWL [8] zur Prozesssteuerung adaptiert werden können, insbesondere solange der einfache Zugang von Anwendern zu ihren Aufgaben im Fokus steht.

5 Verwandte Arbeiten

Gängige Workflow- und Task-Management-Systeme bspw. von IBM, SAP oder Active Endpoints basieren u. a. auf WS-BPEL, BPEL4People und WS-HumanTask, um Prozesse in einem dienstorientierten Kontext zu modellieren und manuelle Aufgaben einzubinden. Alle drei Techniken entstammen dem Unternehmensumfeld und bieten daher für Gruppen im Fokus dieser Arbeit keine direkte Unterstützung: WS-HumanTask [1] spezifiziert Aufgaben (*Tasks*) als nicht-teilbare Arbeitseinheiten mit einem eigenen Lebenszyklus und einer programmiersprachenunabhängigen (aber XML-basierten) Schnittstelle insbesondere zur Verwaltung dieser Aufgaben. WS-BPEL [3] ist eine XML-basierte Sprache zur Orchestrierung von Diensten und wird durch BPEL4People [2] um eine Integration von Aufgaben (*Human Activities*) erweitert.

WS-BPEL und BPEL4People bieten interessante Möglichkeiten zur Formulierung von Prozessen. Im Rahmen einer allgemeinen und flexiblen Architektur ist aber von konkreten Sprachen soweit möglich zu abstrahieren, um auch alternative Implementierungen zu ermöglichen. WS-HumanTask dagegen ist zwar sprachunabhängig gehalten, unterstützt aber z. B. mit seinem Fokus auf nichtteilbare Aufgaben keine kollaborativen Aufgaben (der definierte Lebenszyklus erlaubt keine derartigen Aufgabentypen). Die auf diesen Techniken basierenden Lösungen leiden für den hier anvisierten — aber bei deren Entwicklung nicht vorgesehenen — Einsatz an weiteren Problemen: Es gibt kein einheitliches Modell für Aufgabenlisten und -zuweisungen (Aufgabenverwaltung) und die Lösungen sind — da für das komplexe Unternehmensfeld mit hohen Anforderungen an Robustheit, Skalierbarkeit u. ä. entwickelt — für nicht professionelle Gruppen zu komplex und kostspielig. Der Einsatz eines Enterprise Service Bus bspw. ist nicht für das Internet und die Nutzung frei verfügbarer Dienste optimiert, die aber im Rahmen der o. g. Anforderungen hier wesentlich sind.

Das freie und leichtgewichtige Workflow-Management-System YAWL (Yet Another Workflow Language) [7, 8] ermöglichen den Einsatz einer Prozessunterstützung für verteilte Gruppen mit eingeschränkter IT-Infrastruktur wie bspw. von Ouyang et al. [5] beschrieben. Da jedoch Prozessausführung (in Form der *Workflow Engine*) und Aufgabenliste (*Worklist*) eng miteinander verzahnt sind, wird der Einsatz der integrierten Aufgabenverwaltung mit anderen Workflow-Management-Systemen nicht unterstützt. Das Ziel eines einzigen Einstiegspunktes für Anwender wird — sofern nicht alle Prozesse von einem Anbieter stammen — demnach nicht erreicht.

Selbst im geschäftlichen Umfeld führt der Mangel einer standardisierten Aufgabenverwaltung dazu, dass Anwender, die in mehreren Unternehmen oder Abteilungen mit eigener Softwareinfrastruktur involviert sind, mehrere Aufgabenlisten verwalten und abarbeiten müssen, und dass eine Auslagerung einzelner Aufgaben und Teilprozesse nicht unterstützt wird. Unger et al. [6] analysieren ausführlich Anforderungen an eine Lösung, die die unternehmensübergreifende Bearbeitung von Aufgaben ermöglicht, und stellen eine resultierende Architektur vor. Auch wenn einige gemeinsame Anforderungen wie z. B. die prinzipielle Forderung einer gemeinsamen Aufgabenverwaltung (*Task Management*) bestehen,

sind aufgrund der unterschiedlichen Rahmenbedingungen (bspw. (kein) Rückgriff auf IT-Fachpersonal und eigene IT-Infrastruktur) wesentliche Unterschiede festzuhalten, die in anderen Architekturstilen münden: Während bei Unger et al. [6] Task Engines für die Ausführung von einzelnen Aufgaben verantwortlich sind und Klienten auf mehrere Task Engines zugreifen können (den Klienten müssen also initial die Task Engines bekannt gemacht werden), wird in dieser Arbeit eine Anwenderadressierung eingeführt, die flexiblere Aufgabenzuweisungen und Aufgabenausführungsumgebungen ermöglicht, dafür aber bspw. die Einhaltung von Zuweisungsrichtlinien durchaus erschweren kann.

6 Zusammenfassung und Ausblick

Aufgaben von Anwendern in Prozessen können vielfältiger Natur sein: Einfache Genehmigungs- und Prüftätigkeiten oder kreative Aufgaben, alleine oder kooperativ auszuführende Aufgaben, der Organisation des Anwenders, einmaligen Veranstaltungen oder ehrenamtlich und privaten Projekten entstammende Aufgaben, und in bspw. Java, XML oder PHP realisierte Aufgaben. Letztendlich muss aber jede Aufgabe ihren Bearbeiter (Anwender) finden. Dazu wurde hier ein einfacher, von der E-Mail-Infrastruktur inspirierter Architekturstil vorgestellt, dessen Konkretisierung Gegenstand kommender Forschungsarbeit werden wird.

Auf diese Architektur aufbauend sind Datenschutz, Authentifizierung und Sicherung gegen Spam komfortabel zu ermöglichen. Aber auch die Flexibilität, Kontrollierbarkeit und Nachvollziehbarkeit ganzer Prozesse sind in Zukunft noch zu adressieren.

Literatur

1. Agrawal, A. et al.: Web Services Human Task (WS-HumanTask), Version 1.0, 2007.
2. Agrawal, A. et al.: WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007.
3. Alves, A. et al.: Web Services Business Process Execution Language Version 2.0, 2007.
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, 2000.
5. Ouyang, C., La Rosa, M.; ter Hofstede, A.H.M., Dumas, M. & Shortland, K.: Toward Web-Scale Workflows for Film Production. IEEE Internet Computing 12(5), 53 – 61, 2008.
6. Unger, T. & Bauer, T.: Towards a Standardized Task Management. Multikonferenz Wirtschaftsinformatik, GITO-Verlag, Berlin, 2008.
7. van der Aalst, W.M.P. & ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems, 30(4), 245 – 275, 2005.
8. YAWL Foundation: YAWL Yet Another Workflow Language. http://www.yawl-system.com [2009-02-20]

Anforderungen der industriellen Produktion an eine serviceorientierte Architektur

Jochen Traunecker

gridsolut GmbH + Co. KG
Rauberweg 26
D-73249 Wernau
jochen@traunecker.net

Zusammenfassung Anhand des Beispiels einer fiktiven Toasterproduktion werden Schlüsselkomponenten der industriellen Produktion im Kontext einer serviceorientierten Architektur vorgestellt. Die atomare Organisationseinheit der industriellen Produktion in Form des Arbeitssystems wird näher betrachtet und abstrahiert. Die Anforderungen der industriellen Produktion an eine serviceorientierte Architektur werden anhand eines skizzierten Metamodells vorgestellt und am Beispiel der Datenversorgung und Entsorgung verdeutlicht.

1 Industrielle Produktion

Unter industrieller Produktion versteht man die Erzeugung von Ausbringungsgütern (Produkten) aus materiellen und nichtmateriellen Einsatzgütern (Produktionsfaktoren) nach bestimmten technischen Verfahrensweisen. Der industrielle Produktionsprozess setzt sich aus einzelnen Abschnitten, die jeweils einen bestimmten Teilprozess der Produktion eines Erzeugnisses umfassen, zusammen. Die Ausführung eines Abschnittes erfolgt in organisatorischen Einheiten, den Arbeitssystemen: Ein Arbeitssystem ist die kleinste selbstständig arbeitsfähige Einheit in einem Produktionssystem [1].

2 Einführendes Beispiel Toasterproduktion

Als einführendes Beispiel soll eine fiktive Fließproduktionslinie für Toaster dienen, deren Aufbau in Abbildung 1 dargestellt ist.

In dieser Produktionsstätte werden aus Blechen in der Gehäusepresse die Rohgehäuse für Toaster gepresst. Die Rohgehäuse bekommen in der Gehäuselackierung ihre Lackierung und werden in der Gehäusevormontage zusammengefügt. Die erste Qualitätsprüfung findet in der Prüfung für mechanische Bauteile statt. Die Steuerelektronik des Toasters sowie der Netzanschluss werden danach eingabut. Zum Schluss werden Warnhinweise angebracht. Abschließend wird vor dem Versand eine Endkontrolle durchgeführt.

Die Fließproduktionslinie zur Toasterproduktion bietet die Möglichkeit, Toaster individuell mit bestimmten Lacken zu lackieren (siehe Abbildung 2). Dazu wird

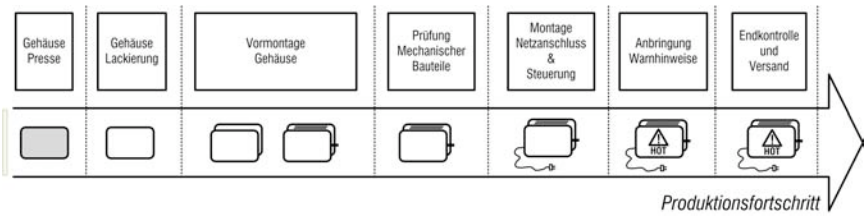


Abbildung 1: Anordnung der Arbeitssysteme einer beispielhaften Toasterproduktion

jeder Toaster mit einem Fertigungsauftrag versehen, der auch die Lackfarbe beinhaltet. Die Gehäuselackierung kann also anhand des Fertigungsauftrags entsprechend konfiguriert werden um den passenden Lack aufzubringen. Die Prüfung mechanischer Bauteile beinhaltet die Prüfung auf korrekte Lackierung eines jeden Toasters. Auch die folgenden Arbeitssysteme werden entsprechend des Fertigungsauftrags konfiguriert: die Auswahl des länderspezifischen Netzteils, die Anbringung der Warnhinweise, sowie die für das jeweilige Land gesetzeskonforme Endprüfung des Toasters.

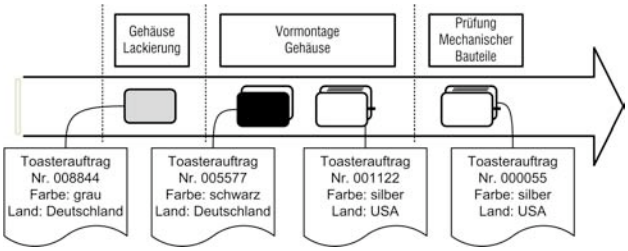


Abbildung 2: Variantenreiche Serienfertigung am Beispiel Lack

2.1 Arbeitssysteme der Toasterproduktion

Eine atomare Organisationseinheit (Arbeitssystem Montage Netzanschluss) wird in Abbildung 3 detailliert dargestellt: Ein Barcodescanner liest vom Gehäuse dessen Auftragsnummer. Abhängig vom Auslieferungsland des Auftrages wird in der Konfigurationsdatenbank eine entsprechende Prozessdefinition zur Konfiguration der Steuerelektronik, zur Montage und zur Prüfung der Leistungsaufnahme abgefragt. Anschließend wird die eigentliche Montage entsprechend der Prozessbeschreibung automatisiert abgearbeitet und bei Bedarf mit dem Bedienpersonal des Arbeitssystems interagiert. Alle Arbeitssysteme der Beispielproduktion sollen möglichst lange autark arbeiten können. Sollte zum Beispiel die Auftragsdatenbank temporär nicht verfügbar sein, darf dies nicht zu einem unmittelbaren Produktionsstop am Arbeitssystem führen. Durch die Replikation einer Teilmenge

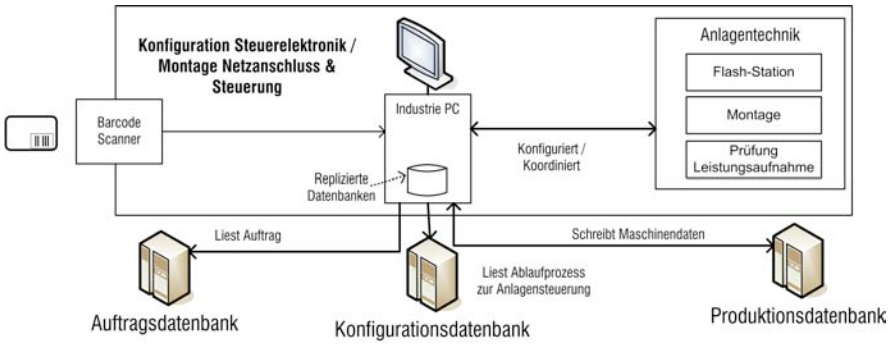


Abbildung 3: Arbeitssystem Montage Netzanschluss

des Datenbestandes der Auftrags- und Konfigurationsdatenbank auf den Industrie-PC des Arbeitssystems können nicht langdauernde Störungen der Infrastruktur kompensiert werden.

Die Arbeitssysteme werden durch diverse Datenbanksysteme mit Daten versorgt und können in diese ihrerseits Daten der Produktion entsorgen. Durch Produktionsüberwachungssysteme kann die Produktion beobachtet werden. Mit den Systemen zur Produktionskonfiguration können die Arbeitssysteme verwaltet werden.

2.2 Abstrahiertes Arbeitssystem

Ein Arbeitssystem kann, wie in Abbildung 4 dargestellt, abstrahiert beschrieben werden. Das Arbeitssystem baut sich dabei aus folgenden Bausteinen auf:

Prozessverzeichnis Die für ein bestimmtes Arbeitssystem relevanten Prozessdefinitionen sind in diesem Verzeichnis hinterlegt. In Abbildung 5 ist eine Prozessdefinition auszugweise skizziert und beschreibt die Interaktion des Arbeitssystems mit dem Produkt, dem Informationsraum, den technischen Anlagen sowie den Werkern.

Informationsraum der Produktion In diesem Informationsraum liegen alle produktionsrelevanten Informationen, die sowohl abgefragt als auch aktualisiert werden können.

Technische Anlagen und Werkerinteraktion Die technischen Anlagen interagieren mit dem Produkt.

Laufzeitumgebung Die Prozessdefinitionen können durch die Laufzeitumgebung abgearbeitet werden. Die Laufzeitumgebung sucht sich dabei für jedes zu bearbeitende Produkt aus dem Prozessverzeichnis die gültigen Prozessdefintionen aus.

Anhand der im Prozessverzeichnis hinterlegten Prozessdefinitionen können die Schnittstellen eines Arbeitssystems zum Informationsraum abgeleitet werden. Auch lassen sich damit die Abhängigkeiten zwischen den Arbeitssystemen folgern.

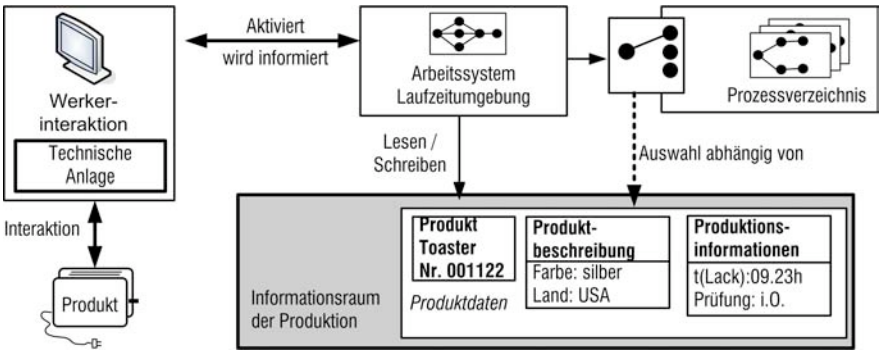


Abbildung 4: Abstrahiertes Arbeitssystem

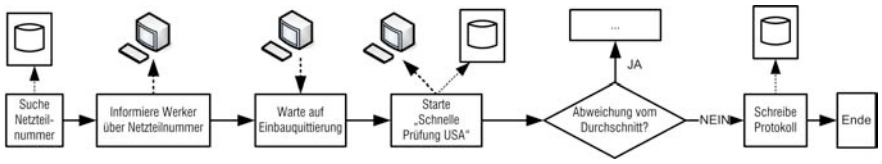


Abbildung 5: Skizzierte Prozessdefinition eines Arbeitssystems

3 Anforderungen der industriellen Produktion an eine SOA

Die Eigenschaften einer serviceorientierten Architektur und deren Realisierung in Form von Web-Services eines Service Bus werden in [2] ausführlich aufgezeigt. Als fundamentale Anforderung der industriellen Produktion an eine SOA kann ihr Beitrag zur Minimierung der Kosten der informationstechnologischen Infrastruktur gesehen werden:

- Betriebs- und Wartungskosten
- Integrationskosten der Arbeitssysteme
- Kosten der Restrukturierung der Arbeitssysteme

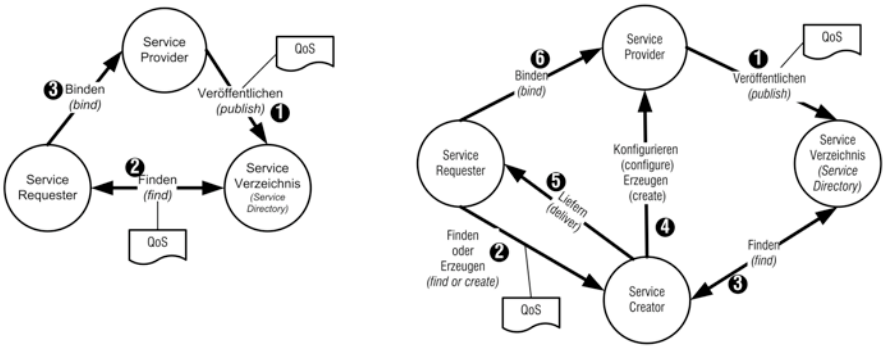
Eine SOA sollte also bestrebt sein, lediglich die tatsächlich geforderten Qualitätsansprüche der gebundenen Service Requester zu bedienen.

Die drei Basisrollen einer SOA (Service Provider, Service Requester und das Service Verzeichnis) und deren Interaktionsmuster (Veröffentlichen, Finden und Binden) werden in Abbildung 6 (a) dargestellt. Im Kontext der industriellen Produktion sollte die Rolle des Service Creators hinzugefügt werden (siehe Abbildung 6 (b)): Ein Service Requester sucht dabei nicht mehr direkt im Service Verzeichnis, sondern nutzt einen Service Creator, um einen Dienst mit entsprechenden Qualitäten (QoS [3]) gegebenenfalls erzeugen zu lassen. Die neue Rolle ist notwendig, da davon ausgegangen werden kann, dass geforderte Servicequalitäten bestimmter Dienste in der SOA zum Zeitpunkt der Suche noch nicht vorhanden sind. Fehlen Infrastrukturkomponenten oder Kapazitäten zur

Erzeugung eines geforderten Service Providers, dann sollte der Service Creator Empfehlungen zur Ausgestaltung der Infrastruktur geben.

Abbildung 6: Rollen in einer SOA

(a) Basis Rollen einer SOA (b) Erweiterte Rollen einer SOA



Die Realisierung einer SOA für die industrielle Produktion sollte ein möglichst umfassendes Metamodell pflegen. In dieses Modell könnten dabei folgende Aspekte einfließen:

Netzwerkinfrastruktur Wie ist das zugrunde liegende Netzwerk aufgebaut? Welche Komponenten sind in der Netzwerktopologie vertreten, welche Servicequalitäten können diese anbieten? Wie sind die individuellen Service Provider und Service Requester angebunden? Kann auf Servicequalitäten des Netzwerks Einfluss genommen werden [4, 5]?

Rechenkapazität Welche Rechenkapazität ist vorhanden? Wie sind die individuellen Arbeitssysteme ausgestattet? Können Rechner der Arbeitssysteme temporär Daten aufnehmen? Kann Software auf die individuellen Arbeitssysteme ausgerollt werden?

Software Welche Softwarekomponenten sind verfügbar? Welche Infrastruktur wird von den Softwarekomponenten erwartet? Wie können sie installiert werden?

Domänenmodelle Welche Metamodelle sind Grundlage der Domänenmodelle [6, 7]? Wie werden Produkte der industriellen Produktion modelliert [8]? Welche Produktmodelle gibt es?

Prozessmodelle Wie werden die Prozessmodelle zur Bearbeitung der Produkte modelliert [9]? Welche Prozessdefinitionen gibt es?

Datenlieferanten Welcher Service liefert Daten einer bestimmten Qualität, Quantität und zu welchen Zeiten [10, 11]?

Datenkonsumenten Welcher Service benötigt Daten einer bestimmten Qualität, Quantität und zu welchen Zeiten?

Der Service Creator könnte zum Beispiel - wie in Abbildung 7 skizziert - anhand der Metadaten für ein Arbeitssystem einen Service Endpunkt erzeugen, der Daten direkt ins Langzeitarchiv übermittelt. Im Fehlerfall (Netzwerkseparierung, Nichtverfügbarkeit des Langzeitarchivs) würde dem Aufrufer des Services ein Fehler gemeldet werden.

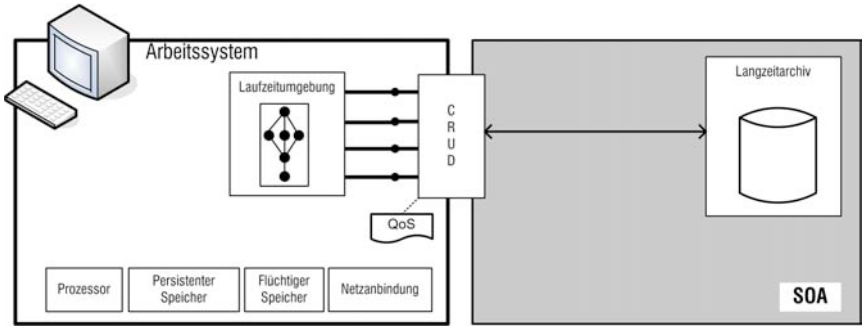


Abbildung 7: Service Endpunkt Beispiel A

Alternativ könnte der Service Creator wie in Abbildung 8 skizziert auf einem Arbeitssystem Software installieren, die der Laufzeitumgebung einen Service Endpunkt anbieten könnte. Dieser Service Endpunkt würde alle Daten auf dem Arbeitssystem zwischenspeichern und erst dann an das Langzeitarchiv übermitteln. Darüber hinaus könnte eine Peer-to-Peer-Verbindung zu einem weiteren Arbeitssystem aufgebaut werden, um Daten direkt zu übermitteln. Sollte das Langzeitarchiv temporär nicht verfügbar sein, würde dies zu keinem Fehler führen und die beiden direkt abhängigen Arbeitssysteme könnten ihre Aufgaben erfüllen.

4 Ausblick

Das hier vorgestellte Beispiel der Toasterproduktion stellt eine starke Vereinfachung der realen industriellen Produktion dar. Eine methodisch fundierte Erhebung der Besonderheiten der industriellen Produktion im Hinblick auf die Einbettung in eine serviceorientierte Architektur könnte zu einem umfassenden Katalog an Anforderungen führen.

Das skizzierte Metamodell einer SOA in der industriellen Produktion könnte detailliert modelliert werden. Darüber hinaus können Methoden und Algorithmen zur dynamischen Konfiguration der Akteuere recherchiert und auf Brauchbarkeit hin geprüft werden [12–16].

Literatur

1. Günther, H.O., Tempelmeier, H.: Produktion und Logistik. Springer-Verlag, New York (2005)

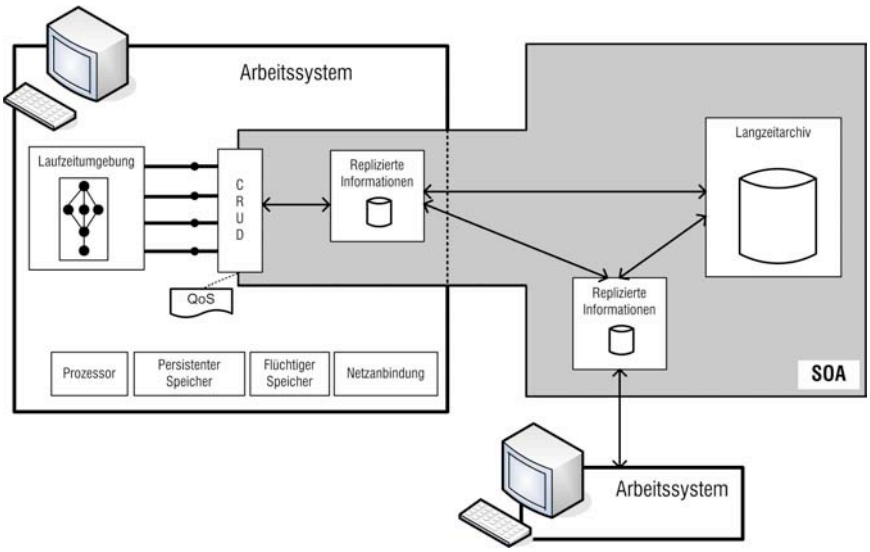


Abbildung 8: Service Endpunkt Beispiel B

2. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson Education (2008)
3. Sabata, B., Chatterjee, S., Davis, M., Sydir, J.J., Lawrence, T.F.: Taxonomy for qos specifications. In: Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), IEEE Computer Society (1997)
4. Tanenbaum, A.S.: Computer Networks, Fourth Edition. Pearson Education, Upper Saddle River, New Jersey (2003)
5. Schmidt, K.: High Availability and Disaster Recovery. Springer-Verlag, Berlin Heidelberg (2006)
6. Allemang, D., Hendler, J.: Semantic Web for the Working Ontologist. Morgan Kaufmann (2007)
7. Date, C., Darwen, H.: Foundation for Object / Relational Databases. Addison Wesley Longman (1998)
8. Evans, E.: Domain-Driven Design. Pearson Education (2004)
9. Leymann, F., Roller, D.: Production Workflow Concepts and Techniques. Prentice-Hall, Upper Saddle River, New Jersey (2000)
10. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer-Verlag, Berlin (2006)
11. Batini, C., Scannapieco, M.: Data Quality. Springer-Verlag, Berlin (2006)
12. Murch, R.: Autonomic Computing. Pearson Education (2004)
13. Fellenstein, C.: On Demand Computing. Pearson Education (2005)
14. Foster, I., Kesselman, C.: The Grid2 Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (2004)
15. Berman, F., Hey, A., Fox, G.: Grid Computing: making the global infrastructure a reality. John Wiley and Sons Ltd (2003)
16. Joseph, J., Fellenstein, C.: Grid Computing. Pearson Education, Inc. (2004)

Towards Choreography Transactions

Oliver Kopp, Matthias Wieland, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
{lastname}@iaas.uni-stuttgart.de

Abstract. The focus of choreography modeling is to capture the message exchange between processes. Common choreography modeling languages do not provide capabilities to group activities of different participants together into an all-or-nothing group. This paper presents choreography spheres as a modeling technique for cross-process transactions based on BPEL4Chor and sketches a mapping to BPEL.

1 Introduction

BPMN [1] is an established standard for modeling processes and process choreographies. While BPMN is mostly used for modeling business processes, BPEL [2] is used for executing processes. BPEL itself defines orchestration only. Support for process choreographies is added by BPEL4Chor [3]. Since the execution semantics of BPMN is still not defined for all cases, we focus on BPEL4Chor, which has an agreed semantics [4].

In BPEL4Chor a choreography is modeled by modeling the behavior of each participant as BPEL process and interconnecting the pools using message links. Internal transaction behavior is modeled in BPEL by scopes. A scope groups activities together. If a scope is completed, its whole work may be compensated as a response to failures in subsequent steps. The compensation is either performed by explicitly modeled compensation behavior or by the default compensation. The default compensation executes the (explicitly modeled) compensation actions for each activity in reverse order of their execution. Thus, an all-or-nothing behavior may be achieved even for activities not providing an "undo" operation themselves. However, there exist scenarios where it is helpful for process modelers to use modeling constructs spanning over different processes to express that the selected activities of different processes belong together and have to be coordinated. Usually, the coordination has to be modeled manually including the activities needed to communicate with a coordinator. The contribution of this paper is the introduction of *choreography spheres*, which represent this missing construct. The main advantage of using choreography spheres is that the coordination between the participant does not have to be modeled explicitly. By using choreography spheres, the coordination is done automatically. We present two distinct types of choreography spheres, each being a new modeling construct for the two common transaction types: short running and long running transactions.

In general, we define a choreography sphere as shown in Definition 1.

Definition 1 (Choreography Sphere). *A choreography sphere ensures transactional behavior of all enclosed activities that can be part of multiple processes.*

The definition leads to a all or nothing semantic of the sphere, which means all activities are either executed successfully or the effects are undone.

In the following, an overview on background and related work is given in Section 2. Afterwards, two types of spheres with different transaction properties are described in Section 3 and Section 4. Finally, Section 5 concludes and presents an outlook on future work.

2 Background and Related Work

The concept of spheres with transactional behavior was first introduced in [5]. The spheres were allowed to overlap, but the semantics for choreography spheres was put as future work. Currently, there exist no work on cross-organizational transactions, where arbitrary activities may be chosen to be coordinated.

There are two different types of transaction styles available in the the business area: business transactions and ACID style transactions [6]. Both are supported by spheres. But they have different implications on the modeling and also the technical coordination needed. Because of that we define in this paper 2PC Spheres in Section 3 for the ACID transactions and the BPEL Sphere in Section 4 for the Business transactions.

In the field of Web services, the standards published by the OASIS Web Services Transaction (WS-TX) Technical Committee are the standards used in practice. They provide transaction support across different vendors. The WS-Coordination specification [7] describes the general transaction framework. It describes the protocols for participant registration and defines a transaction context. This context can be passed using messages to enable the recipient to register as participant of the same transaction. WS-Coordination describes a protocol service, which handles the concrete coordination protocols. WS-Coordination is open to any transaction protocol. Up to now, WS-Atomic Transaction (WS-AT, [8]) and WS-Business Activity (WS-BA, [9]) are defined. WS-AT defines the two-phase-commit protocol, which is used for ACID transactions and thus provides and all-or-nothing behavior [10]. WS-BA is used for long-running transactions, which may last for years. It builds on the Saga model [11], where each activity has an associated compensation activity. In Saga, the activities are executed one after another. If an activity fails, the executed activities are compensated in reverse order. An overview of the history of transaction handling and current approaches in the field of Web services is given in [12].

It is shown in [13] how transactional behavior of services called by a BPEL process can be enforced. A transaction policy is attached to a group of BPEL activities. The transaction policy states which transaction protocol is used at invoking the grouped activities. We re-use this idea in our approach to enable transaction context propagation from the choreography sphere to the called services.

A variant of cross-process spheres is presented in [14], called "split BPEL scopes". These scopes result from a split of a BPEL process among different participants. The split BPEL scopes keep the semantics of the BPEL scope in the unsplit process and are coordinated using the WS-Coordination infrastructure [15]. Our approach allows for picking arbitrary activities in a choreography to be coordinated.

In the field of WS-Coordination, transaction protocols are defined using a coordination protocol graph and additional textual description. The approach presented in [16] shows how the coordination protocol graphs can be transformed to an abstract BPEL process for the coordinator and one BPEL process for the participant. Each of them has to be manually refined to adhere the requirements of the textual description of the transaction protocol. This has to be done only once for each transaction protocol. We use the resulting executable BPEL coordination logic in our approach as described in Section 3.

3 2PC Spheres

The two-phase commit protocol (2PC for short) is a well-known protocol to coordinate distributed ACID transactions enabling an all-or-nothing behavior. Due to their nature, ACID transactions are used for short-term operations (a few seconds). The two booking workflows presented on the left side (participant behavior descriptions) in Fig. 1 are executed in parallel, but need to be coordinated to ensure an all-or-nothing semantics of the two pay activities. To achieve that they are nested in a 2PC sphere, which ensures that no money is transferred if one activity fails. Only if both are executed successfully the whole sphere is

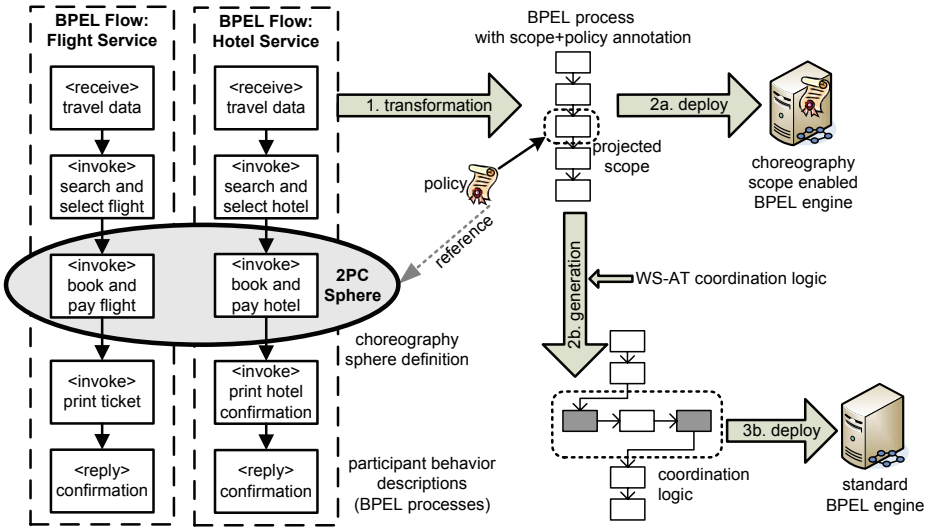


Fig. 1. Transformation steps required for the execution of choreography spheres

committed. Thus, one is sure that no money needs to be claimed back in any case of failure. Until now, 2PC spheres are defined for the flow activity only and cannot be nested.

The sphere definition has to be represented in each BPEL process to enable proper execution. The necessary steps for that are presented in Fig. 1. In step 1, each participant behavior description is transformed to a BPEL process, where a scope is put around the activities belonging to the choreography scope. The scope is annotated with a policy stating that the scope belongs to a choreography scope. In step 2a, each process is deployed on a choreography scope enabled BPEL engine. Step 2b is motivated by the non-existence of BPEL engines supporting choreography scopes. Thus, the alternative approach is to generate the coordination logic into each BPEL process. Finally, step 3 deploys each executable process on a standard BPEL engine.

4 BPEL Spheres

In contrast to short-running transactions, business transactions are usually long-running [17] as shown in example presented in Fig. 2. In this example a production company plans a new product and orders the parts for that product from different subcontractors. They produce the parts and send them to the production company that assembles them to the end product. In case of a fault in the inner BPEL sphere, the production company tires to find a new subcontractor. If this is not possible or any other error is propagated to the outer BPEL sphere, all running activities inside the outer BPEL sphere are terminated and all successful completed activities are compensated. The inner BPEL sphere is treated as child

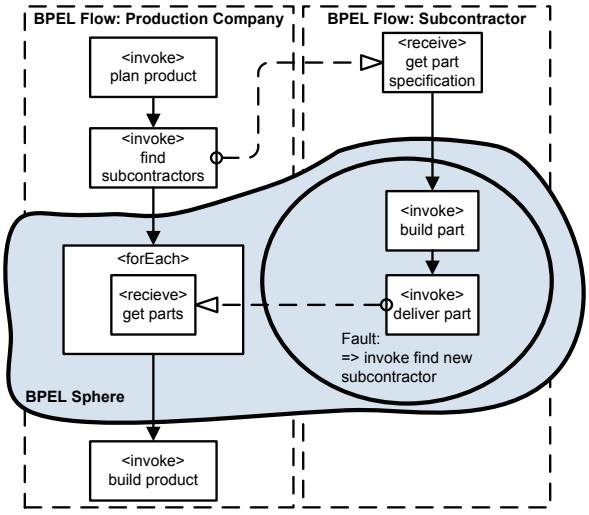


Fig. 2. BPEL sphere extending BPEL's scope semantics across BPEL processes

activity of the outer sphere and is handled the same way. This kind of process is running significantly longer than the money transfer actions presented in Section 3. Due to the long processing time, it is not possible to lock all data used in the processes. Thus, WS-AT cannot be used for the involved Web services as done in the case of 2PS spheres. The transformation steps presented in Section 3 are similar for BPEL spheres. The only change is that the used coordination protocol changes from WS-AT to WS-BA.

In BPEL, long-running transactions are realized by scopes. The concept of BPEL spheres extends BPEL's scope semantics to choreographies. A BPEL sphere groups activities together to form a transactional unit. The BPEL sphere is a long-running sphere and therefore uses compensation as undo operation.

Each BPEL sphere may have fault handlers and a compensation handler attached. BPEL spheres must be properly nested similar to scopes. BPEL spheres may not overlap with BPEL scopes similar to 2PC spheres. The activities in these handlers must be annotated which the participant, where the respective activities run. After the choreography is defined and it comes to the mapping to separate BPEL processes, the activities inside the handlers are split as presented in [15]. The remainder of the transformation and the deployment follows the procedure presented in Fig. 1. The projected scopes are annotated with a policy including the reference to the choreography scope definition.

It is possible to require that at least one activity in each participant runs in a BPEL sphere. This requirement ensures that all projected scopes run, which in turn is a requirement resulting from the premises by [14]. Thus, the projected BPEL scopes can be coordinated using the protocols described in [14]. An implementation of BPEL spheres has been described in [18], which is based on the pluggable framework [19].

5 Conclusion and Outlook

In this paper, we presented the concept of choreography spheres, where arbitrary activities of different processes can be grouped together. We showed a sketch of a possible implementation using the BPEL and WS-Coordination. We plan to add a full runtime support for choreography spheres to the Apache ODE engine.

Currently, choreography spheres may not overlap and the included activities may only be part of the same scope and loop. We sketched the nesting of BPEL spheres and the interplay with local scopes. A semantics for overlapping spheres is shown in [5]. Thus, our future work is to study possible semantics of overlapping BPEL spheres and of BPEL spheres overlapping with local BPEL scopes.

Motivated by [6,20], we research whether there are requirements for additional types of spheres. We are going to evaluate the applicability of these types in choreography settings.

Acknowledgments This work is supported by the BMBF funded project Tools4BPEL (01ISE08B) and the DFG project Nexus (SFB627).

References

1. Object Management Group (OMG): Business Process Modeling Notation (BPMN) Version 1.2. (2009) `http://www.bpmn.org/`.
2. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard. (2007)
3. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: ICWS. (2007)
4. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and Participant Synthesis. In: WS-FM. (2007)
5. Leymann, F.: Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems. In Lausen, G., ed.: BTW. (1995)
6. Greenfield, P., Fekete, A., Jang, J., Kuo, D.: Compensation is Not Enough. In: EDOC, Washington, DC, USA (2003)
7. OASIS: Web Services Coordination (WS-Coordination) Version 1.1. (2007)
8. OASIS: Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1. (2007)
9. OASIS: Web Services Business Activity (WS-BusinessActivity) Version 1.1. (2007)
10. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufman (1993)
11. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD. (1987)
12. Wang, T., Vonk, J., Kratz, B., Grefen, P.: A survey on the history of transaction management: from flat to grid transactions. Distributed and Parallel Databases **23**(3) (2008) 235–270
13. Tai, S., Khalaf, R., Mikalsen, T.A.: Composition of Coordinated Web Services. In Jacobsen, H.A., ed.: Middleware. (2004)
14. Khalaf, R., Leymann, F.: Coordination Protocols for Split BPEL Loops and Scopes. Technical Report Computer Science 2007/01, University of Stuttgart, Institute of Architecture of Application Systems (2007)
15. Khalaf, R.: Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective. Doctoral thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2008)
16. Kopp, O., Wetzstein, B., Mietzner, R., Pottinger, S., Karastoyanova, D., Leymann, F.: A Model-Driven Approach to Implementing Coordination Protocols in BPEL. In: MDE4BPM. (2008)
17. Leymann, F., Roller, D.: Production Workflow – Concepts and Techniques. Prentice Hall PTR (2000)
18. Steinmetz, T.: Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2008) *(In German).*
19. Khalaf, R., Karastoyanova, D., Leymann, F.: Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In: WESOA. (2007)
20. Leymann, F., Pottinger, S.: Rethinking the Coordination Models of WS-Coordination and WS-CF. In: ECOWS. (2005)

Realizability of Interaction Models

Gero Decker

Hasso-Plattner-Institute, University of Potsdam, Germany
gero.decker@hpi.uni-potsdam.de

Abstract. In scenarios where a set of independent business partners engage in complex conversations, interaction models are a means to specify the allowed interaction behavior from a global perspective. Atomic interactions serve as basic building blocks and behavioral dependencies are defined between them. The notion of *realizability* centers around the question whether there exist a set of roles that collectively realize the specified behavior. This notion has been studied in the literature in different flavors. This paper aims at providing an overarching framework for realizability.

1 Introduction

Two approaches for choreography modeling can be identified in the literature. *Interconnection models* are collections of observable behavior models for interacting roles. Each observable behavior model belongs to one role and contains communication activities and the behavioral dependencies between them. Corresponding communication activities are then interconnected. Choreography languages following this style are BPMN [1] and BPEL4Chor [5]. *Interaction models*, on the other hand, consist of interactions and global behavior dependencies between them. They are global in the sense that they are not explicitly assigned to any role and it remains unspecified who is responsible for enforcing them. Choreography languages following this style are WS-CDL [8] and iBPMN [3].

It turns out that some interaction models are not realizable. Imagine that an interaction between C and D must only happen after A and B have interacted. Here, C and D cannot know when the first interaction has actually happened. While this example is obviously not realizable, there are other scenarios where realizability might be given under certain assumptions. This paper will provide a classification of the different dimensions of realizability.

The remainder of this paper is structured as follows. The next section lists a number of motivating examples for the notion of realizability. Section 3 identifies the dimensions of realizability. Section 4 reports on related work, before Section 5 centers around realizability checking. Section 6 concludes.

2 Motivating Examples

Conversation models, as presented in [7], are a formalism for interaction models. They are finite state automata with an alphabet $R \times M \times R$. R denotes the set

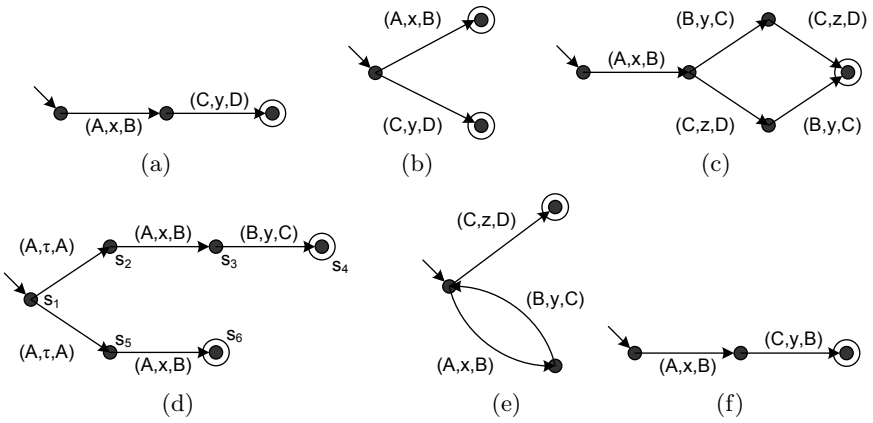


Fig. 1. Conversation model examples

of roles and M the set of message types. The 3-tuple (s, m, r) then describes the sender, the type and the receiver of a message.

Figure 1 illustrates a number of sample interaction models. States are depicted by circles and the transitions by arrows. The initial state is targeted by an arrow without source state and the final state is denoted by a double-circle.

Figure 1(a) shows the example described in the introduction. It is not possible to find interacting roles that exactly show the specified behavior. Nevertheless, it would be possible to find interacting roles that show a subset of the specified behavior: Imagine two roles A and B that interact and roles C and D simply do nothing. In this case, however, a conversation would not terminate properly, as the final state cannot be reached.

Figure 1(b) shows a choice between two interactions. Similarly to the previous example, A and B cannot know whether C and D have already interacted and vice versa. In contrast to the previous example, we can find roles that collectively realize a subset of the specified behavior with proper termination. Imagine again that only roles A and B interact while C and D do nothing. However, we are not able to find a set of roles that realize a subset of the behavior where all interactions from the conversation model are reachable.

Similarly to the first example, the enablement dependency between the AB interaction and the CD interaction is the problem in Figure 1(c). As a solution, C could wait for the message from B before interacting with D. That way, the resulting behavior would be a properly terminating subset of the initially specified behavior.

Figure 1(d) shows an example containing a non-deterministic choice. This conversation model represents that A should internally be able to decide whether B will interact with C later on. However, B cannot observe this decision as in any case it will get a message x from A. As A does not have any control over the BC interaction, the decision whether this interaction takes place or not will be independent form A's initial choice. When only considering the possible traces

of the conversation model we can easily create roles that collectively produce exactly the same traces. The main difference is that B or C can decide whether the final interaction takes place or not in the realization. We see that considering the branching structure is crucial whenever the ownership of (and the moment of) choices is of importance. It might be argued that local choices are irrelevant in choreographies. This might be true if choreographies are considered to be a collection of mere interaction sequences. However, from a business perspective it makes a major difference who makes a branching decision. Therefore, this should be reflected in the formal model as well.

Figure 1(e) shows a cyclic example containing a choice between an AB interaction and a CD interaction, similarly to the second example. The difference here is that by expanding the cycle to a sequence, we can at least find roles that realize a subset of the behavior.

Finally, Figure 1(f) shows an example that is perfectly realizable in a synchronous world, where C can block B until it has interacted with A. However, when considering an asynchronous world, where message sending and receiving do not happen in one step, the order of the send activities would not conform to the order of interactions in the conversation model.

3 Dimensions of Realizability

The examples from the previous section show that we need to distinguish different dimensions of realizability. The following three dimensions apply.

Complete behavior vs. subset of behavior. Choreographies define constraints and obligations of the roles involved. Constraints apply as the choreography enumerates all allowed interactions in every conversation state, obligations apply as a final state must be reached which is only possible through the execution of the given interactions.

In this context, we can either demand that it must be possible to carry out the complete behavior specified in the choreography. Or, a subset of the behavior might already be sufficient. Here, the follow-up question is what a valid subset would be. For instance, proper termination of conversations might be a basic criterion. Furthermore, reachability of all interactions from the original choreography might also be demanded.

Communication model. Synchronous communication could be assumed, where sending and receiving of messages must happen at the same time. Two flavors are possible in this context: it might be allowed that a sender blocks until the receiver is ready to receive the message. Alternatively, the conversation fails if a role can only send in a given state without any other role being able to receive the message.

In asynchronous settings, message send and receive do not happen in one step. Here, message buffers are introduced for storing the incoming messages. We might assume that there is only one queue, e.g. with FIFO message delivery, or that there is a buffer where any incoming message can be received from.

The order of interactions is of central importance. However, especially in the case of asynchronous communication, there are different options of what ordering relationships to consider. For instance, only the ordering of send transitions might be considered, or the ordering of receive transitions or the ordering of communication transitions within the individual roles might be of importance.

Equivalence notion. Having agreed on what ordering relationships to consider, it is important to choose an equivalence notion for comparing the original choreography and the collective behavior of the roles. Here, trace-based techniques can be applied. This is sufficient when dealing with deterministic behavior in the choreography and the roles. Branching structures are of relevance in the presence of non-determinism. Here, bisimulation-like techniques can be used.

In order to formally capture the different notions of realizability we need to introduce the following concepts. C denotes the set of all choreographies (also with silent transitions). R denotes the set of all role behavior models. $\oplus : \wp(R) \rightarrow C$ is a function that composes a choreography out of a set of role behavior models. $\sim \subseteq C \times C$ is a binary relation on choreographies.

Please note that $\oplus$ heavily depends on the communication model chosen and, in the case of asynchronous communication, the ordering relationships to be considered. In the special case of considering the order of communication transitions within a role, $\oplus$ depends on the role under investigation. $\sim$ depends on whether the complete or only a subset of the behavior is demanded and it also depends on the equivalence notion chosen.

Definition 1 (Realizability). *A choreography $c \in C$ is realizable, iff there exists a set of roles $r_1, \ldots, r_n \in R$ such that $\oplus(r_1, \ldots, r_n) \sim c$.*

4 Related Work

Realizability checking for conversation models was presented in [7]. Here, the notion of realizability does not consider branching structures in the conversation models and focuses on trace equivalence between the collective behavior of the roles and the original conversation model. Asynchronous communication is considered where each role has one FIFO queue for all incoming messages. Realizability is broken down to three requirements. (1) Synchronous compatible condition: The conversation model is projected to the different roles, which are then interconnected under the assumption of synchronous communication (called the *syn-configuration*). The condition for each state in the syn-configuration is that whenever a role is ready to send a message there must be another role that is ready to receive this message. (2) Autonomous condition: It is demanded for each role projection that there is no state where the role is ready to send *and* to receive a message. Rather, in each state the role projection is either ready to send one out of a set of messages or ready to receive one out of a set of message. Furthermore, it must not be possible to send or receive message in a final state.

(3) Lossless join condition: The join of the role projections must show exactly the same behavior as the original conversation model. Realizability for message sequence charts was studied in [2].

The notion of local enforceability was first introduced in [10]. Here, only a subset of behavior is demanded as well as the reachability of all interactions from the original choreography. Enforceability checking is carried out using structural rules rather than considering the state space. Synchronous communication was assumed. Realizability and local enforceability was also studied in the context of interaction Petri nets in [6]. Again, synchronous communication is assumed and proper termination and reachability of all interactions is demanded. In contrast to the previous work on local enforceability, enforceability checking is done using the state space. Realizability is defined based on branching bisimulation.

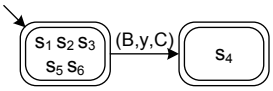
The notion of desynchronizability investigates whether a choreography that is realizable under the assumption of synchronous communication properly terminates under the assumption of asynchronous communication (with one buffer per message type) [4].

5 Realizability Checking

Similarly to the approaches in [7] and [6] we construct the role projections for every role in a conversation model and then study the composition of these projections. As [7] does not consider branching structures, role projection is based on minimal finite state automata containing only those interactions where the particular role is involved in.

As we want to preserve the branching structure we carefully need to consider the observability of choices within each role. We construct the role behavior for a role r in a similar way like in the operating guidelines approach [9]. (a) Start with the initial state $s = s_0$ and create a new node n. (b) Determine those states that can be reached from s without involvement of r. All these states are added to node n. (c) Identify all transitions with involvement of r that originate in one of the states belonging to n. For each transition label (s, m, r) determine if there is already a node n' corresponding to all states s' that are reachable via transitions with label (s, m, r). If such an n' does not exist, create it. For every s' and n' continue with step (b).

(a) Role projection for B



(b) Role projection for C

The nodes and their connections are the resulting role projection c_r for r. The initial node n becomes the initial state of c_r and all nodes containing final states become final states of c_r. Figures 2(a) and 2(b) illustrate this for roles B and C and the conversation model from Figure 1(d).

An exception to rule (c) applies to all those cases where several transitions with the same label originate in the same state. In this case, different nodes must be identified / created. If multiple states belonging to the same node have several transitions with the same label (s, m, r), nodes must be identified / created for the different combinations.

6 Conclusion

This paper motivated different dimensions for realizability and investigated role projection for those realizability notions that are based on bisimulation. Here, the branching structures must be considered carefully in order to cater for the moment of observability of choices. Composition of role projections for different communication models was outside of the scope of this paper due to the space restrictions. Also the binary relation for comparing the composition and the original conversation model was not covered.

The work presented in this paper is based on conversation models. Interaction Petri nets [6] are an alternative formalism for interaction models with concurrency. It is desirable to preserve concurrency as much as possible during role projection. Therefore, future work will center around an approach, where transformation rules similar to those presented in [6] are applied.

References

1. Business Process Modeling Notation, V1.1. Technical report, OMG, Jan 2008.
2. R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *ICSE*, pages 304–313, New York, NY, USA, 2000. ACM.
3. G. Decker and A. Barros. Interaction Modeling using BPMN. In *CBP*, number 4928 in LNCS, pages 206–217, Brisbane, Australia, September 2007.
4. G. Decker, A. Barros, F. M. Kraft, and N. Lohmann. Non-desynchronizable service choreographies. In *ICSOC*, LNCS, Sydney, Australia, Dec 2008. Springer Verlag.
5. G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *ICWS*, pages 296–303, Salt Lake City, Utah, USA, July 2007. IEEE Computer Society.
6. G. Decker and M. Weske. Local Enforceability in Interaction Petri Nets. In *BPM*, number 4714 in LNCS, pages 305–319, Brisbane, Australia, September 2007. Springer Verlag.
7. X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, November 2004.
8. N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation. Technical report, November 2005. http://www.w3.org/TR/ws-cdl-10.
9. N. Lohmann, P. Massuthe, and K. Wolf. Operating guidelines for finite-state services. In *ICATPN*, volume 4546 of *LNCS*, pages 321–341, Siedlce, Poland, June 2007. Springer Verlag.
10. J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker. Service Interaction Modeling: Bridging Global and Local Views. In *EDOC*, pages 45–55, Hong Kong, Oct 2006. IEEE Computer Society.

Realizability is controllability

Niels Lohmann and Karsten Wolf

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{niels.lohmann, karsten.wolf}@uni-rostock.de

Abstract. A *choreography* describes the interaction between services. It may be used for specification purposes, for instance serving as a contract in the design of an interorganizational business process. Typically, not all describable interactions make sense which motivates the study of the *realizability* problem for a given choreography.
In this paper, we show that realizability can be traced back to the problem of *controllability* which askes whether a service has compatible partner processes. This way of thinking makes algorithms for controllability available for reasoning about realizability. In addition, it suggests alternative definitions for realizability. We discuss several proposals for defining realizability which differ in the degree of coverage of the specified interaction.

1 Introduction

Dumas et al. discuss in [1] compatibility between services and introduce a number of related notions. Among these notions they mention *realizability* (the problem whether a choreography can be implemented by services) and *controllability* (the problem whether a service has a compatible partner), but state that "... a formal relation between controllability and realizability is yet to be established". This paper is dedicated to the establishment of this relation.

In Sect. 2, we introduce a formal framework which allows us to reason about choreographies in a formal and language-independent manner. In Sect. 3, we recall different realizability notions and introduce the novel concept of *distributed realizability*, which seamlessly complements existing notions. Section 4 formulates the realizability problem in terms of controllability and shows how algorithms for controllability can be used to proof realizability by synthesizing realizing services. Section 5 discusses further research questions and concludes the paper.

2 A Formal Framework for Choreographies

Throughout this paper, fix a finite set of message channels M that is partitioned into asynchronous message channels M_A and synchronous message channels M_S. From M, derive a set of message events $E := !E \cup ?E \cup !?E$, consisting of asynchronous send events $!E := \{!x \mid x \in M_A\}$, asynchronous receive events $?E := \{?x \mid x \in M_A\}$, and synchronization events $!?E := \{!?x \mid x \in M_S\}$.

Definition 1 (Conversation, choreography). *A* conversation γ *is a finite word over* E *such that, for all* $x \in M_A$, $\#_{!x}(\gamma) = \#_{?x}(\gamma)$ *and for every prefix* γ' *of* γ *holds:* $\#_{!x}(\gamma') \geq \#_{?x}(\gamma')$. *Thereby,* $\#_x(\gamma)$ *denotes the number of occurrences of the message event* x *in the word* γ. *A* choreography *is a set of conversations.*

The requirements for a conversation state that asynchronous events are always paired, and a send event always occurs before the respective receive event. A choreography is a set of desired message event sequences. A choreography is defined with respect to a set of peers which form a collaboration.

Definition 2 (Peer, collaboration). *A* peer $P = [I, O]$ *consists of a set of input message channels* $I \subseteq M$ *and a set of output message channels* $O \subseteq M$, $I \cap O = \emptyset$. *A* collaboration *is a set* $\{P_1, \ldots, P_n\}$ *of peers such that* $I_i \cap I_j = \emptyset$ *and* $O_i \cap O_j = \emptyset$ *for all* $i \neq j$, *and* $\bigcup_{i=1}^{n} I_i = \bigcup_{i=1}^{n} O_i$.

The requirements ensure that communication in a collaboration is always bilateral, yielding a closed system that models no message exchange other than that between the peers.

There are various languages to specify collaborations and choreographies, ranging from formal models such as *Interaction Petri Nets* [2] to industrial notations such as *Let's Dance* [3], UML collaboration diagrams, and BPMN. While these languages differ in syntax and semantics, concepts such as an underlying collaboration (i.e., the endpoints and the exchanged messages) and the choreography (i.e., the intended global behavior) can be easily derived from these languages.

To specify the behavior of a service (i.e., the concrete implementation of a peer), again a variety of languages exist. In essence, they all share a concept of a state (e.g., a state of a finite state machine, a marking of a Petri net, or control flow tokens in BPMN) and ways to specify message transfer. These basic concepts can be expressed with service automata.

Definition 3 (Service automaton). *A* service automaton $A = [Q, I, O, \delta, q_0, F]$ *is a tuple such that* Q *is a finite set of states,* $[I, O]$ *is a peer,* $\delta \subseteq Q \times (E_I \cup E_O \cup \{\tau\}) \times Q$ *is a transition relation,* $q_0 \in Q$ *is an initial state, and* $F \subseteq Q$ *is a set of final states. Thereby,* $E_I := \{?x \mid x \in I \cap M_A\} \cup \{\,!?x \mid x \in I \cap M_S\}$ *and* $E_O := \{!x \mid x \in O \cap M_A\} \cup \{\,!?x \mid x \in O \cap M_S\}$.

We say that A implements $[I, O]$, and for $(q, x, q') \in \delta$, we also write $q \xrightarrow{x} q'$. Beside internal (i.e., silent, non-communicating) τ steps and synchronous communication, service automata also model asynchronous communication, in which messages may overtake each other, but will never get lost. We claim that is — compared to FIFO queues for communicating finite state machines [4] — a more natural approach to model asynchronicity, because it makes less assumptions about the underlying infrastructure. In the composition of two or more service automata, pending asynchronous messages are represented by a multiset. Denote the set of all multisets over M_A with $Bags(M_A)$. Further denote the empty multiset with $[\,]$, and the multiset containing only one instance of $x \in M_A$ with $[x]$. Addition of multisets is defined pointwise.

Definition 4 (Composition of service automata). *Let $A_1, \ldots, A_n$ be service automata such that their peers form a collaboration. Define the* composition *$A_1 \oplus \cdots \oplus A_n$ as the automaton $[Q, \delta, q_0, F]$ with $Q := Q_1 \times \cdots \times Q_n \times Bags(M_A)$, $q_0 := [q_{0_1}, \ldots, q_{0_n}, [\,]]$, $F := F_1 \times \cdots \times F_n \times \{[\,]\}$, and, for all $i \neq j$ the transition relation δ contains excactly the following elements:*

- $[q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{\tau} [q_1, \ldots, q_i', \ldots, q_n, B]$,
  iff $[q_i, \tau, q_i'] \in \delta_i$ *(internal move by A_i),*
- $[q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{!x} [q_1, \ldots, q_i', \ldots, q_n, B + [x]]$,
  iff $[q_i, !x, q_i'] \in \delta_i$ *(asynchronous send by A_i),*
- $[q_1, \ldots, q_i, \ldots, q_n, B + [x]] \xrightarrow{?x} [q_1, \ldots, q_i', \ldots, q_n, B]$,
  iff $[q_i, ?x, q_i'] \in \delta_i$ *(asynchronous receive by A_i), and*
- $[q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n, B] \xrightarrow{!?x} [q_1, \ldots, q_i', \ldots, q_j', \ldots, q_n, B]$,
  iff $[q_i, !?x, q_i'] \in \delta_i$ *and* $[q_j, !?x, q_j'] \in \delta_j$ *(synchronization between A_i and A_j).*

A run of $A_1 \oplus \cdots \oplus A_n$ is a sequence of events $x_1 \cdots x_m$ such that $q_0 \xrightarrow{x_1} \cdots \xrightarrow{x_m} q_f$ with $q_f \in F$. For a run ρ, define the conversation of ρ as $\rho_{|E}$. The choreography of $A_1 \oplus \cdots \oplus A_n$, denoted $Chor(A_1 \oplus \cdots \oplus A_n)$, is the union of the conversations of all runs of $A_1 \oplus \cdots \oplus A_n$.

The composition is finite iff, for each state, the number of pending asynchronous messages is bounded. The choreography of a composition of service automata is the set of all observable event sequences that are produced by runs of the composition.

3 Realizability Notions

With the notion of realizability, the conversations generated by a composition of services can be related to a specified choreography. Bultan et al. [5, 6] define realizability of choreographies as follows:

Definition 5 (Complete realizability). *A choreography C is* completely realizable *w.r.t. a collaboration $\{P_1, \ldots, P_n\}$ iff there exists a tuple of service automata $[A_1, \ldots, A_n]$ such that, for all i, A_i implements P_i, and $Chor(A_1 \oplus \cdots \oplus A_n) = C$.*

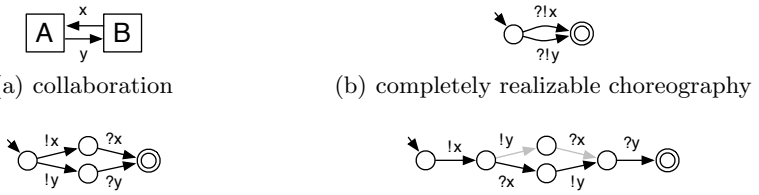
Complete realizability is a strong requirement, because it demands that the observable behavior of the endpoints exactly matches the choreography. In reality, it is often the case that not all aspects of a choreography can be implemented. To this end, Zaha et al. [7] introduce a weaker notion called *local enforceability* (or *partial realizability*) which only demands that a subset of the choreography is realized by the peer implementations:

Definition 6 (Partial realizability). *A choreography C is* partially realizable *w.r.t. a collaboration $\{P_1, \ldots, P_n\}$ iff there exists a tuple of service automata $[A_1, \ldots, A_n]$ such that, for all i, A_i implements P_i, and $Chor(A_1 \oplus \cdots \oplus A_n) \subseteq C$.*

Obviously, complete realizability implies partial realizability. Though this weaker notion ensures that all constraints of the choreography are fulfilled, it still fixes a single tuple of service automata. If there does not exist such tuple of automata that realizes the *complete* choreography, there might still exist a *set* of tuples — each partially realizing the choreography — which distributedly realizes the complete choreography:

Definition 7 (Distributed realizability). *A choreography C is* distributedly realizable *w.r.t. a collaboration $\{P_1, \ldots, P_n\}$ iff there exist tuples of service automata $[A_{1_1}, \ldots, A_{n_1}], \ldots, [A_{1_m}, \ldots, A_{n_m}]$ such that, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, (i) A_{i_j} implements P_i, (ii) $Chor(A_{1_j} \oplus \cdots \oplus A_{n_j}) \subseteq C$, and (iii) $\bigcup_{j=1}^{m} Chor(A_{1_j} \oplus \cdots \oplus A_{n_j}) = C$.*

Distributed realizability allows for build time coordination between peers. While being a stronger notion than partial realizability (i.e., more of choreography's behavior is implemented), it is still a weaker notion than complete realizability.



(a) collaboration

(b) completely realizable choreography

(c) distributedly realizable choreography

(d) partially realizable choreography

Fig. 1. Example choreographies with respect to a collaboration.

As an example, consider the collaboration depicted in Fig. 1(a) in which two peers, A and B, communicate via message channels x and y. The choreography $\{!?x, !?y\}$ in which the peers communicate synchronously (b) is completely realizable by a set of peers which synchronously decided whether to synchronize via message x or y. In case the messages are sent asynchronously (c), this is no longer possible. This choreography is not completely realizable, because there does not exist a single pair of service automata that implement the specified behavior. Instead, the implementations have to be coordinated: either peer A sends a message and peer B is quiet or the other way around. These two pairs distributedly realize the whole choreography. Finally, choreography (d) can only be partially realized, because the conversation !x!y?x?y cannot be implemented even if the peers are coordinated at build time. This is because the requirement that message x is sent before message y cannot be enforced, because the peers cannot coordinate this.

4 Synthesizing Realizing Peer Implementations

In this section, we show how the different realizability notions are related to controllability [8]. Controllability is a correctness criterion for services: a service A is controllable iff there exists a service B such that $A \oplus B$ is compatible (i.e.,

deadlock free). Controllability can be extended to multi-port services. In the following, we will transform a choreography into a multi-port service which is controllable if and only if the choreography is realizable.

For a regular[1] choreography C, there exists a deterministic finite state machine that accepts exactly the sequences of the choreography. We call this state machine the *monitor* for C [9]. It unobtrusively monitors the interaction between the peers and reaches a final state iff the monitored conversation was part of the choreography.

Definition 8 (Monitor, monitored composition). *Let C be a regular choreography w.r.t. a collaboration $\{P_1, \ldots, P_n\}$. Define the deterministic finite state machine accepting C (the* monitor *for C) as $M = [Q_M, \delta_M, q_{0_M}, F_M]$. Thereby, Q_M is a finite set of states, $\delta_M : Q_M \times E \to Q_M$ is a transition function, $q_{0_M} \in Q_M$ is an initial state, and $F_M \subseteq Q_M$ is a set of final states.*

Let $A_1, \ldots, A_n$ be service automata implementing $P_1, \ldots, P_n$. Define the monitored composition *$M \otimes (A_1 \oplus \cdots \oplus A_n)$ as the automaton $[Q, \delta, q_0, F]$ with $Q := Q_M \times Q_1 \times \cdots \times Q_n \times Bags(M_A)$, $q_0 := [q_{0_M}, q_{0_1}, \ldots, q_{0_n}, [\,]]$, $F := F_M \times F_1 \times \cdots \times F_n \times [\,]]$, and, for all $i \neq j$, the transition relation δ contains excactly the following elements:*

- $[q, q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{\tau} [q, q_1, \ldots, q_i', \ldots, q_n, B]$, *iff* $[q_i, \tau, q_i'] \in \delta_i$ *(internal move by A_i),*
- $[q, q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{!x} [q', q_1, \ldots, q_i', \ldots, q_n, B + [x]]$, *iff* $[q_i, !x, q_i'] \in \delta_i$ *and* $[q, !x, q'] \in \delta_M$ *(asynchronous send by A_i, monitored by M),*
- $[q, q_1, \ldots, q_i, \ldots, q_n, B + [x]] \xrightarrow{?x} [q', q_1, \ldots, q_i', \ldots, q_n, B]$, *iff* $[q_i, ?x, q_i'] \in \delta_i$ *and* $[q, !?, q'] \in \delta_M$ *(asynchronous receive by A_i, monitored by M),*
- $[q, q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n, B] \xrightarrow{!?x} [q', q_1, \ldots, q_i', \ldots, q_j', \ldots, q_n, B]$, *iff* $[q_i, !?x, q_i'] \in \delta_i$, $[q_j, !?x, q_j'] \in \delta_j$, *and* $[q, !?x, q'] \in \delta_M$ *(synchronization between A_i and A_j, monitored by M).*

The monitor synchronizes with the message events of the service automata, but does not constrain their behavior. The monitor only has an effect on the final states of the composition. Only if all service automata *and* the monitor reach a final state, this state is final in the monitored composition.

We can now change the point of view and regard the monitor as a service that is communicating with several other services by synchronous message events. Again, this service will reach a final state iff the message events from the environment are observed in the correct order.

Definition 9 (Monitor service automaton). *Let C be a regular choreography w.r.t. a collaboration $\{P_1, \ldots, P_n\}$ and let $M = [Q_M, \delta_M, q_{0_M}, F_M]$ be a monitor for C. Define the* monitor service automaton *$A_M := [Q, \mathcal{I}, \mathcal{O}, \delta, q_0, F]$ with $Q := Q_M$, $\mathcal{I} := \{I_1, \ldots, I_n\}$, $\mathcal{O} := \{O_1, \ldots, O_n\}$, $q_0 := q_{0_M}$, and $F := F_M$. Define $\delta : Q \times \{!?\langle x\rangle \mid x \in E\} \to Q$ with $\delta(q, !?\langle x\rangle) := \delta_M(q, x)$.*

[1] Choreographies specified by interaction Petri nets, UML collaboration diagrams, BPMN, or Let's Dance are always regular if they assume synchronous communication.

Due to the nature of a choreography to exist of message events (not the messages itself), all message events of the monitor service automaton are synchronous. The original nature of the event (synchronous or asynchronous) is, however, encoded in the events by using the event $?!\langle x \rangle$ for event x. The existence of service automata that are compatible to this monitor service automaton proof realizability of the choreography:

Theorem 1. *Let C be a regular choreography w.r.t. a collaboration $\{P_1, \ldots, P_n\}$ and A_M a monitor service automaton for C.*

1. *C is partially realizable iff A_M is decentralized controllable.*
2. *C is distributedly realizable iff A_M is decentralized controllable and for the set of strategies $\mathcal{S}$ holds: $\bigcup_{[A_1,\ldots,A_n] \in \mathcal{S}} Chor(A_1 \oplus \cdots \oplus A_n) = C$.*
3. *C is completely realizable iff A_M is decentralized controllable and there exists a strategy $[A_1, \ldots, A_n]$ such that $Chor(A_1 \oplus \cdots \oplus A_n) = C$.*

A strategy for the monitor service automaton is a set of service automata such that the overall composition is compatible. While these automata communicate solely synchronously with the monitor service automaton, a tuple of service automata that actually realize the original choreography can be derived by changing every message event $!?\langle x \rangle$ back to x (e.g. "$!?\langle !x \rangle$" to "$!x$"). These service automata then can also communicate asynchronously with other peers, yet still follow the specified choreography.

5 Conclusion

In this paper, we linked the realizability problem to controllability, making existing tools and algorithms applicable to check choreographies. In particular, our approach allows for specification and synthesis of asynchronous peer implementations. This avoids a subsequent analysis and correction when trying to "asynchronize" peer implementations [10]. The current results are independent of a concrete choreography or service description language, but can be easily adapted.

In future work, we plan to study how choreography and service design can be mixed. For example, the realizability of a choreography can be checked w.r.t. some already completely specified peers. Such an approach would nicely complement the participant synthesis introduced in [11] by using controllability for both interaction models and interconnection models.

Acknowledgements This work is funded by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

References

1. Dumas, M., Benatallah, B., Nezhad, H.R.M.: Web service protocols: Compatibility and adaptation. IEEE Data Eng. Bull. **31**(3) (2008) 40–44

2. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: BPM 2007. LNCS 4714, Springer (2007) 305–319
3. Zaha, J.M., Barros, A.P., Dumas, M., Hofstede, A.H.M.t.: Let's dance: A language for service behavior modeling. In: OTM 2006. LNCS 4275, Springer (2006) 145–162
4. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2) (1983) 323–342
5. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. **328**(1-2) (2004) 19–37
6. Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. SOCA **2**(1) (2008) 27–39
7. Zaha, J.M., Dumas, M., Hofstede, A.H.M.t., Barros, A.P., Decker, G.: Service interaction modeling: Bridging global and local views. In: EDOC 2006, IEEE Computer Society (2006) 45–55
8. Wolf, K.: Does my service have partners? LNCS ToPNoC **II**(5460) (2009) 152–171
9. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007. LNCS 4714, Springer (2007) 271–287
10. Decker, G., Barros, A., Kraft, F.M., Lohmann, N.: Non-desynchronizable service choreographies. In: ICSOC 2008. LNCS 5364, Springer (2008) 331–346
11. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: WS-FM 2007. LNCS 4937, Springer (2008) 46–60

Do We Need Internal Behavior in Choreography Models?

Oliver Kopp and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
{lastname}@iaas.uni-stuttgart.de

Abstract. Choreographies capture the message exchanges between multiple processes. Certain choreography languages ignore the internal behavior completely, other languages offer the possibility to model internal behavior. This paper presents an example modeled in both types of languages and discusses the need to integrate internal behavior in choreographies.

1 Introduction

A choreography defines the message exchanges between several processes. While it is fundamental that a choreography language has to include the ability to express message exchanges, it is an open question whether a choreography language should offer constructs to model internal behavior. In this paper, we present a sample choreography and use this example to discuss, whether internal behavior should be modeled in a choreography.

The example is introduced in Sect. 2. Afterwards, it is modeled in Let's Dance (Sect. 3) and BPMN (Sect. 4). An overview on the support of modeling internal behavior of the most prominent choreography modeling languages is given in Sect. 5. Finally, Sect. 6 provides a conclusion and points out future work.

2 Drop-dead Order Scenario

Fig. 1 presents the drop-dead order scenario, where a customer requests a distributor to ship products until a given drop-dead date. The distributor neither produces the products by himself nor owns a delivery. Therefore, the distributor asks a supplier whether he can produce the requested products in time and the distributor asks a carrier whether he can carry the produced products by time. If the supplier and the carrier agree, then the customer's order is accepted and rejected otherwise. The example itself was first presented in [1] and is used in [2] to develop transaction requirements for services. We use this example in this paper to illustrate the difference between different choreography languages.

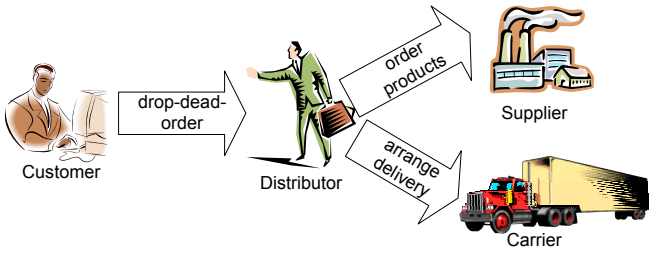


Fig. 1. Drop-dead oder (adapted from [1])

3 Let's Dance Model

Let's Dance [3] is a choreography language having the interaction as basic building block. Therefore, the model is called "interaction model" [4]. A choreography model of the drop-dead order scenario is presented in Fig. 2. The topmost box denotes that a customer sends an order to a distributor. Control flow is modeled using directed arcs. The first arc points to a group, labeled with a circled A in the figure. In this group, the message exchange between the customer and the shipper and between the customer and the carrier happens in parallel. The crossed line between the acceptance and the rejection denotes that exactly one of the two message exchanges may happen. After both the supplier and the carrier have sent an answer, they have either accepted or rejected. Afterwards, group B is active. In case both the supplier and the carrier accepted the request of the distributor, they receive an acceptance of the distributor and the products are built and delivered to the customer. Otherwise, the order is rejected at the supplier and the carrier. The client's order is rejected, too.

In the presented choreography, there is no internal behavior modeled. The messages in the choreography suggest that the supplier is somehow producing products and that the carrier delivers them. However, there is no explicit description of these tasks.

4 BPMN Model

Figure 3 presents the drop-dead order scenario using BPMN [5]. BPMN is a choreography language belonging to the category of interconnection models [4]. In interconnection models, the control flow is modeled per participant. While the interaction is one building block in interaction models, the interaction is split up in send and receive activities in the case of interconnection models. To provide a better overview on the model, one path through the model is highlighted. This path denotes the case, where both the supplier and the carrier accept the request of the distributor. First, the customer sends an order to the distributor, which asks the supplier and the carrier in parallel, whether they can produce/deliver in time. The supplier and the carrier decide and both accept the order. Since both have accepted, they are notified that the distributer accepted, too. Finally, the

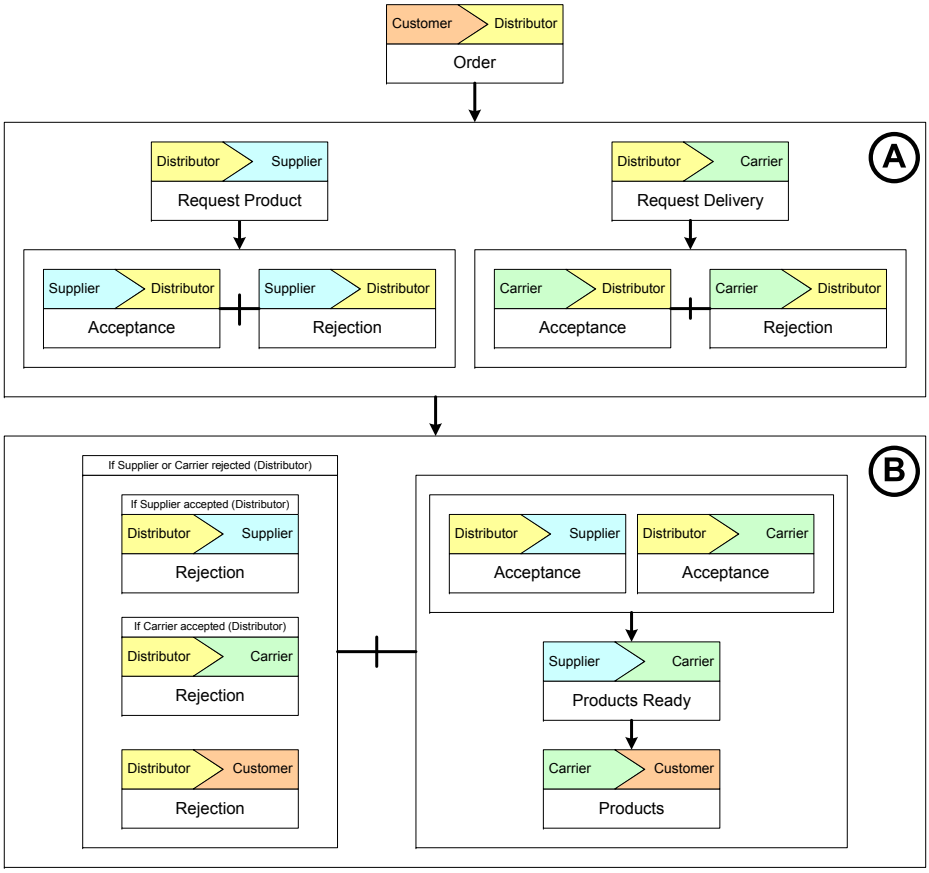


Fig. 2. Drop-dead order scenario modeled in Let's Dance

producer produces, notifies the carrier, which picks up the products and delivers them to the customer.

Besides the split of the interactions into interconnections of send and receive activities, two internal activities of the supplier and the carrier are shown. In case of the supplier, the choreography model shows the decision activities and the produce activity. When it comes to implement a supplier based on the choreography description, the internal behavior can be used as basis for an executable BPEL process [6]. In case the choreography model is detailed enough, an IT export just has to add concrete WSDL information to the activity to turn the process into an executable BPEL process.

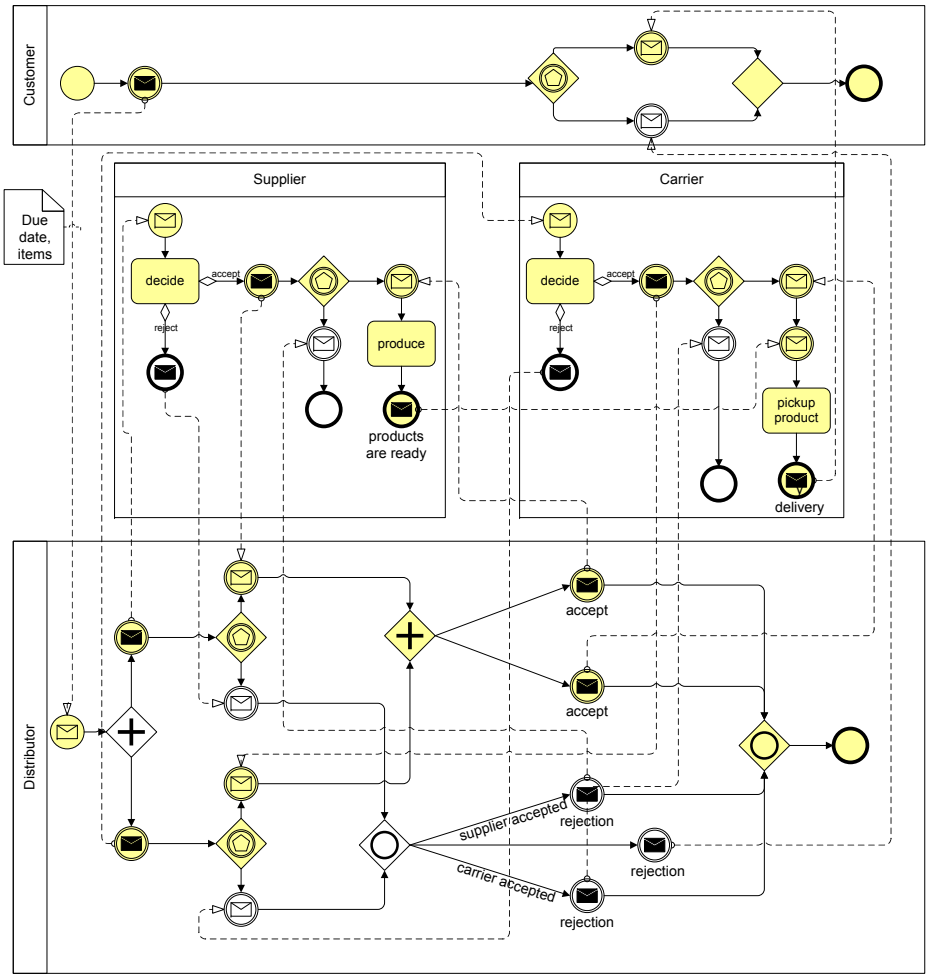


Fig. 3. Drop-dead order scenario modeled in BPMN

5 Support of Internal Actions

Language	Model Type	Constructs for Internal Behavior Available?
BPEL4Chor	interconnection	+
BPMN 1.2	interconnection	+
iBPMN	interaction	–
Let's Dance	interaction	–
WS-CDL	interaction	+

Table 1. Support of modeling internal actions in choreography languages

Table 1 lists five choreography languages and their support of internal actions. The languages are chosen, because they are the most prominent choreography languages. The second column denotes whether the language allows to model interaction models or interconnection models.

BPEL4Chor [7] is a choreography language extending BPEL with choreography constructs. Since BPEL is re-used, so called opaque activities can be used to model internal behavior. The Business Process Modeling Notation Version 1.2 (BPMN 1.2) has been used in Sect. 4 to model the drop-dead order example. It has been shown that this language supports internal behavior. iBPMN [8] is an extension for BPMN to support interaction modeling. Each message exchange is atomic and does not have to be modeled with two distinct send and receive operations. iBPMN does not support the inclusion of internal behavior. It is shown in Sect. 3 that Let's Dance does not support the inclusion of internal behavior. The Web Services Choreography Language (WS-CDL, [9]) provides the "silent action activity" to express internal behavior. Thus, WS-CDL is the only language following the interaction approach and providing support for internal actions.

6 Conclusion and Outlook

In this paper, we gave an overview on the support of modeling internal behavior in choreography languages. The drop-dead order scenario was introduced and modeled using Let's Dance and BPMN. Finally, we showed the support of internal actions in five choreography languages. All languages supporting interconnection models also support the modeling of internal behavior. In the case of interaction models, two of the three languages do not support modeling internal behavior. WS-CDL is the only language based on interaction models, where internal behavior can be modeled.

The examples suggest that it depends on the use-case of the choreography whether internal actions are needed. If the choreography is used to serve a basis

for executable processes, the internal actions can be seen as placeholders for calls to internal services. Thus, the effort to build executable BPEL processes seems less. We are going to describe concrete scenarios and to use them as basis for a comparison.

Acknowledgments This work is supported by the BMBF funded project Tools4BPEL (01ISE08B).

References

1. Haugen, B., Fletcher, T.: Multi-Party Electronic Business Transactions. Technical report, UNCEFACT (2002)
2. Sun, C., Aiello, M.: Requirements and Evaluation of Protocols and Tools for Transaction Management in Service Centric Systems. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), IEEE Computer Society (2007) 461–466
3. Zaha, J.M., Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Let's Dance: A Language for Service Behavior Modeling. In: CoopIS 2006: Proceedings 14th International Conference on Cooperative Information Systems. Volume 4275 of Lecture Notes in Computer Science., Springer (2006) 145–162
4. Decker, G., Kopp, O., Barros, A.: An Introduction to Service Choreographies. Information Technology **50**(2) (2008) 122–127
5. Object Management Group (OMG): Business Process Modeling Notation (BPMN) Version 1.2. (2009) `http://www.bpmn.org/`.
6. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard. (2007)
7. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In Society, I.C., ed.: Proceedings of the IEEE 2007 International Conference on Web Services, IEEE Computer Society (2007) 296–303
8. Decker, G., Barros, A.P.: Interaction Modeling Using BPMN. In ter Hofstede, A.H.M., Benatallah, B., Paik, H.Y., eds.: CBP: Proceedings of the 1st International Workshop on Collaborative Business Processes. Volume 4928 of Lecture Notes in Computer Science., Springer (2007) 208–219
9. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0. W3C. (2005) `http://www.w3.org/TR/ws-cdl-10`.

All links were followed on 2009-02-19.

Creating Message Profiles of Open Nets

Jan Sürmeli and Daniela Weinberg

Humboldt–Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{suermeli,weinberg}@informatik.hu-berlin.de

Abstract. In a network of services, external and internal decisions, asynchronous message exchange, and concurrency induce complex interaction protocols. In this paper we introduce the notion of a message profile of a service that is modeled as a special kind of Petri net.
The message profile is obtained solely from properties of the given net without requiring knowledge of interacting nets. It provides insight into the interactional behavior of the service. The information may then be used to enhance existing service analysis techniques as well as to verify the service model on a message basis.

1 Introduction

The central part of the evolving paradigm of Service-Oriented Computing (SOC) are services. A service represents a self-contained software unit that offers an encapsulated functionality over a well-defined interface. The promising goal of a SOC architecture is to ensure for each participating service to be loosely coupled with another service with little effort [1] and thus, creating a network of services, that is able to handle certain tasks. In contrast to other paradigms it is possible to create a heterogenous network that crosses organizational boundaries.

In general, a service is not designed to be used stand-alone. It is the stateful interaction of different services that adds significant value to SOC. Therefore, with respect to SOC, we are interested in whether every service instance will eventually terminate in a well-defined state with no useless (dead) activities being pending. This idea has already been formalized as usability in [2]. We use the term *controllability* instead of usability to avoid misunderstandings w.r.t. other well established meanings of "usability". We analyze whether two services S and S' can *interact properly* [3]. A service S' that properly interacts with service S is a *controller* of S. In our approach, we model a service as an open net [4, 5], which is a special class of Petri nets that extends classical Petri nets by an interface for communication with other open nets. We assume an asynchronous setting in which the order of sending messages does not necessarily correspond to the order of receiving those messages.

In a network of two or more services, external and internal decisions, asynchronous communication, and concurrency induce complex interaction protocols. In this paper, we introduce techniques that can be used to create a message profile of a given service S. This profile may then serve as a guide for a controller

C with respect to which messages may be sent to S as well as how often S accepts a particular message. The message profile is gained from analyzing the open net model of S. Thus, knowledge of C is not required. With the help of the behavioral properties stored in the message profile, we are able to characterize possible controllers. So, we can exclude certain controllers before-hand which enhances established methods such as partner synthesis [6] or the computation of the operating guidelines of S [5]. Furthermore, we enable the modeler of S to verify that the model mirrors its designated interactional behavior.

This paper is structured as follows. First, we briefly introduce open nets and controllability notions in Sect. 2. In Sect. 3, we present techniques to build up the message profile and show how it can be obtained by analysis of an open net. Finally, we conclude our results in Sect. 4.

2 Open Nets and Controllability

We model services with open nets [4], which enhance classical Petri nets. An open net is a tupel $N = (P, T, F, P_{in}, P_{out}, m_0, \Omega)$ with P being the set of places, T the set of transitions and F the flow relation. The set $P_{in} \subseteq P$ ($P_{out} \subseteq P$) represents the *input channels* (*output channels*) of the service. For the rest of this paper, we call P_{in} (P_{out}) *input* (*output*) *places* and $P_{in} \cup P_{out}$ the *interface* of N. For node $n \in P \cup T$ the set ${}^\bullet n = \{x \mid (x, y) \in F\}$ ($n^\bullet = \{y \mid (x, y) \in F\}$) is the *preset* (*postset*) of n. We demand that ${}^\bullet p = \emptyset$ ($p^\bullet = \emptyset$) for every $p \in P_{in}$ ($p \in P_{out}$) and $P_{in} \cap P_{out} = \emptyset$. Transition $t \in T$ with ${}^\bullet t \subseteq P_{in} \,/\, t^\bullet \subseteq P_{out} \,/\, {}^\bullet t \cup t^\bullet \not\subseteq P_{in} \cup P_{out}$ is called a *receiving / sending / internal* transition. m_0 is the initial marking and Ω is the set of final markings, which constitute the set of final states that the service should reach. The *inner* of a net N is obtained from N by removing the input/output places and adjacent edges from N. The set $LL = \{m_k, \ldots, m_n\}$ is a livelock iff all $m_i \in LL$ are mutually reachable and from no m_i an $m_j \notin LL$ is reachable.

Figure 1 depicts our example open net N1. The net has got five input places, namely a, b, c, d, e and one output place F. The initial marking of N1 is [p0] (depicted as a black token on place p0) and the set of final markings is {[p2], [p7]}. We can easily see that from [p0] N1 is not able to reach a different marking unless there is an additional token either on input place a, b or e. A token on an input/ output place represents a message. The open net N1 is able to send a message to the output place and to receive messages from its input places. Therefore, we are able to model the interaction between different open nets and thus have a formal notion for modeling the interaction of services.

The interaction of two different open nets N and C is expressed by their composition $N \oplus C$ which is obtained by merging every input place of one open net with the equally labeled output place of the other net (if that one is present).

Intuitively, controllability of an open net N means that N can *properly interact* with some other net. So, N is *controllable* if there exists an open net C, such that the composed open net $N \oplus C$ fulfills certain properties. Throughout

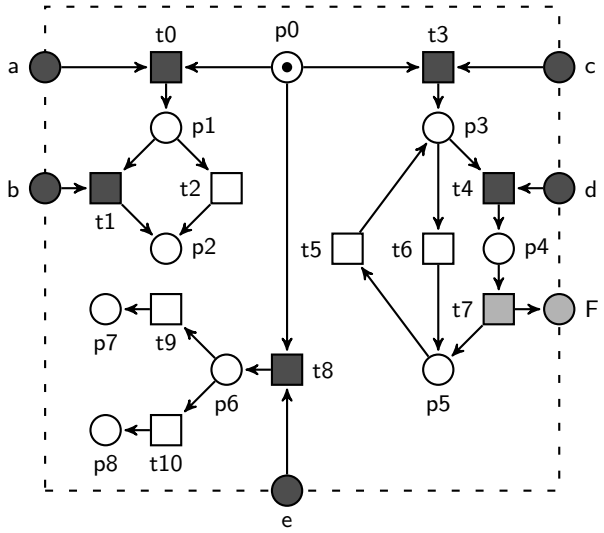


Fig. 1. Example open net N1.

this paper we will call C a *controller* of N. We distinguish between DF-, WT-, and RI-controllers based on the following notions [3].

Deadlock-Freedom (DF) states that all deadlocks of the composition $N \oplus C$ are final states of $N \oplus C$ and *Weak-Termination (WT)* (equal to *Livelock-Freedom*) specifies that from every marking of $N \oplus C$ a final marking of $N \oplus C$ is reachable. The property of open nets called *responsiveness* [3] can easily be mapped to the composition of two services – *Responsive Interaction (RI)*. A composition is responsive if either from every marking m of $N \oplus C$ a final marking of $N \oplus C$ is reachable, or from m a marking m' is reachable such that either N or C has sent or received a message. We further demand that each message sent will eventually be received.

Controllability is only decidable for those open nets whose reachability graph of the inner net is finite [7]. We also demand that the communication between two open nets is limited – there are no more than k messages ($k \in \mathbb{N}$) on any interface place at any reachable marking of the composition [5, 7].

3 Message Profile

In this section, we introduce techniques that analyze the net N in order to create a message profile of N. All methods avoid state space exploration and do not require knowledge of a controller C of N.

In the message profile we store different kinds of information – how often a specific message can be received, dependencies between messages, and which messages are not to be sent to N by a controller.

Intuitively, there are two requirements for an arbitrary service S to receive a message x: (1) S is in a state in which it accepts x, (2) a message x has been sent by controller C already. These requirements can easily be mapped to an open net N. Thus, (1) a receiving transition for x is enabled in the inner of N and (2) a token is available on the corresponding input place.

When analyzing a receiving transition t in the inner of N one less precondition for t to be enabled has to hold – the edge to its corresponding input place has been removed. Thus firing t does not depend on the number of tokens on that input place any more. So, we conclude that t can not fire more frequently in $N \oplus C$ than in the inner of N. Knowing how many times a receiving transition fires in the inner of N can thus lead to insights regarding the receiving behavior of N in $N \oplus C$. A *firing limit* for a transition t is a natural number n such that there exists no path in the reachability graph with t occurring more than n times.

There exist methods to compute the firing limit of a transition in a Petri net that avoid state space exploration. Currently, we are looking for an approach that fits best w.r.t. to the open net models of real processes. In this paper, usage of the term *firing limit* will refer to some firing limit determined by an arbitrary method. In the inner of our example open net N1 (Fig. 1) 1 is a firing limit for {t0,t1,t2,t3,t9,t10} and ∞ for transitions {t4,t5,t6,t7}.

We can use the firing limit for receiving transitions in the inner of the net as a basis and as an additional constraint for the creation of the message profile without requiring any knowledge about C.

3.1 Receiving Limit

The idea behind the *receiving limit* n of a message x is to determine how often x can be received by a service N at most, i.e. how often a receiving transition for x fires in $N \oplus C$. As we aim at being as general and controller-independent as possible, we make use of the firing limit for receiving transitions in the inner of N.

We calculate the receiving limit of message a of the example net N1 (Fig. 1). The only receiving transition for message a is t0. The firing limit for t0 is 1. Obviously, there is no guarantee that N1 will actually receive a, but it is safe to say that N1 will receive a only up to once. We set the *receiving limit* for a to 1.

Let us take a look at transitions t4 and t6 with $^\bullet$t4 = $^\bullet$t6 = {p3}. Due to the cycle, the firing limits for both t4 and t6 are ∞. Thus, we cannot determine a finite number for the receiving limit of message d. So, we set it to ∞.

In N1 there exists exactly one receiving transition for each input place. Generally, for a message x, we set the receiving limit to the sum of the firing limits of *all* receiving transitions of x. If there exists no receiving transition for x, the receiving limit is 0 – N does not accept x in any marking. If we set the receiving limit for x to ∞, there are two possible reasons – (1) There exists no bound for N receiving x. Or, (2) we can not narrow down a finite number for such a bound. Either way, a receiving limit of ∞ does not allow further conclusions.

Finding a finite receiving limit n for a message x proves to be very useful. Although n might not be precise, as there is no guarantee that x will be received exactly n times, it is always safe to say that sending x more than n times leads to x being ignored. Therefore, we can conclude that every RI-controller respects the receiving limit of each message. Thus, we include it in the message profile. Summarizing, the receiving limits of the example net N1 are a:1,b:1,c:1,d:∞,e:1.

3.2 External one-time Decisions

So far, the message profile consists of a static number – the receiving limit for each input message, only. In an open net N, however, there might exist *dependencies* between messages that influence the actual acceptance of them. From the structure of N we are able to extract conflicting receiving transitions. The set of *external one-time decisions* contains all messages that these transitions receive, $\{x, y, z\}$. Basically, a controller C influences the further course of N by sending a message x, for instance. Because $\{x, y, z\}$ are in conflict, messages $\{y, z\}$ will now be ignored by N. We include such a set of messages into the message profile.

We will take a look at the open net N1 of Fig. 1 again. Examining the transitions t0, t3 and t8 in the inner of N1, we can easily see that there exists a conflict between them: $^\bullet$t0 = $^\bullet$t3 = $^\bullet$t8. The decision between t0, t3 and t8 is non-deterministic in the inner of N1. But, considering the interface, the choice is made by the controller – by sending one of the messages $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$. Assume message a is received by N1. Then, no marking is reachable where one of $\{\mathsf{t0}, \mathsf{t3}, \mathsf{t8}\}$ is enabled again. Thus, sending message c or e would result in N1 ignoring that message.

In general, we construct such a set of messages M of an open net N as follows. We start by finding a set of receiving transitions that are in conflict with each other in the inner of N, set D. Then, we remove all transitions with a firing limit of 0 from D, set D'. From the receiving transitions in D' we extract the corresponding messages, set M. Set M does not necessarily reflect a global situation. Hence, we check whether the receiving limit of every message $m \in M$ is 1. This way we ensure that there exists no receiving transition $t \in T \setminus D$ for m. If that condition does not hold for a message, we remove it from M. We repeat that process until the condition holds for each message of M. The resulting set is now globally valid. So, sending more than one message from M always leads to N ignoring at least one message. Therefore, no RI-Controller sends more than one message from M which we include in the message profile. The set of conflicting messages $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$ forms an external one-time decision of N1.

3.3 Internal Decisions

Internal decisions describe dependencies between receiving transitions and internal or sending transitions. They potentially induce that messages cannot be received. Based on the receiving limit we can decide if such a conflict leads to ignored messages or not.

We look at t1 and the internal transition t2 in N1 (Fig. 1). In the inner of N1, we find $^\bullet$t1 = $^\bullet$t2 = {p1}. Suppose marking m with m(p1) = m(b) = 1. The decision between firing t1 or t2 is non-deterministic. Thus, we call t1 *blocked* by an internal transition. The firing limits are t1:1,t2:1. Once $t \in \{\mathsf{t1}, \mathsf{t2}\}$ fires, no marking m' is reachable such that any $t' \in \{\mathsf{t1}, \mathsf{t2}\}$ is enabled. Hence, N1 decides non-deterministically between ignoring and receiving b .

For transitions t4 and t6 a similar pattern holds in the inner of N1, $^\bullet t4 = {}^\bullet t6$. The difference is, that whenever one of $\{\mathsf{t4}, \mathsf{t6}\}$ fires, a marking will be reached in which both transitions are enabled again. Assume now a message d is sent to N1 – even if t6 is fired consecutively, d can still be received in any reachable marking.

More generally, a receiving transition t is *blocked* by an internal or sending transition t' if $^\bullet t' \subseteq {}^\bullet t$ holds in the inner of N. We call a a *message x blocked* by an internal decision if (1) we find a finite receiving limit for x and (2) each receiving transition for x is either blocked or has a firing limit of 0. No RI-controller sends x. DF-controllers might send x, but then the composition will always contain a livelock. We include an according set of messages in the message profile. For N1 only message b is blocked by an internal decision, since condition (1) does not hold for message d.

3.4 Trap Messages

Deficient modeling, modification of an existing service or deliberate design of error conditions can lead to a structure, where the postset of an internal place (neither an input nor an output place) of the underlying open net N is either empty or consists of transitions with a firing limit of 0 in the inner of N. If such a place is not marked in any final marking, firing a transition in its preset *traps* N in a state from which no final state is reachable. We show under which circumstances messages can be tagged as *trap messages* in the message profile.

Examining place p8 in the example open net N1 (Fig. 1), we notice that from any marking m with m(p8) > 0 there will be no marking m' reachable with m'(p8) = 0. Additionally, m_f(p8) = 0 for each final marking m_f. Because of $^\bullet$p8 = {t10}, transition t10 should never fire. We call m a *trap marking*, p8 a *trap place* and t10 a *trap transition*. We propagate this property. Because $^\bullet$t10 = {p6}, we conclude that a token on p6 induces firing of t10 and thus to trapping the net. Thus, p6 is a trap place and all transitions of $^\bullet$p6 (= {t8}) are trap transitions. So, transition t8 is a trap transition. t8 is also a receiving transition for message e. Since there exists no other receiving transition for message e, it is obvious that sending message e will either lead N1 into a trap state (if t8 fires) or it leads N1 to ignore message e (if t8 does not fire). Therefore message e is not sent by any WT-controller.

To make sure that a specific message m is to blame for leading to a trap marking, we analyze its receiving transitions and check if each of them is either a trap transition or has a firing limit of 0. In that case, we tag m as a trap message in the message profile. In N1, only message e is a trap message.

4 Conclusion and Future Work

With the help of the example open net $\mathsf{N1}$ (Fig. 1) we have shown that we are able to gain knowledge about the interactional behavior directly from the inner of $\mathsf{N1}$ without knowing any controller C or building up the reachability graph of $\mathsf{N1} \oplus C$. We now know that any RI-Controller C sends messages a and c not more than once and completely avoids to send messages b and e. After having sent a message $x \in \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$, C sends no more messages from $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$.

Currently, we work on a prototypical implementation of our results. We explore solutions for finding firing limits for transitions. Further, we improve our analysis methods and work on combining the methods to accomplish synergy effects. So far, we only focus on receiving messages. It is also possible to extend the message profile not only by further dependencies between receiving messages, but to include information about sending messages as well. Regarding that, we aim at developing a concept of compatibility of message profiles of two services in order to improve matching of two services.

References

1. Papazoglou, M.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (2007)
2. Martens, A.: Analyzing Web Service Based Business Processes. In: FASE 2005. Volume 3442 of LNCS., Springer-Verlag (2005) 19–33
3. Wolf, K.: Does my service have partners? Transactions on Petri Nets and Other Models of Concurrency (2008) (Accepted for publication in November 2008).
4. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. AMCT **1**(3) (2005) 35–43
5. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: ICATPN 2007. Volume 4546 of LNCS., Springer-Verlag (2007) 321–341
6. Weinberg, D.: Efficient controllability analysis of open nets. In: WS-FM 2008. LNCS, Springer-Verlag (2008) accepted.
7. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Inf. Process. Lett. (2008) accepted.

An efficient necessary condition for compatibility

Olivia Oanea* and Karsten Wolf

Universität Rostock, Institut für Informatik
18051 Rostock Germany
{olivia.oanea,karsten.wolf}@uni-rostock.de

Abstract. Composing services makes sense only if they are compatible, i.e. composition does not lead to problems such as livelocks or deadlocks. In general, compatibility can be checked using state space explorations on any kind of formal models of services.
Petri nets, one of the formal models in use, offer a rich theory for reasoning without exploring a state space. Among the techniques is the so-called *state equation* which forms a linear algebraic necessary condition for reachability of states.
In this article, we show how the state equation can be applied for a necessary condition for compatibility. This way, the number of expensive state space based compatibility checks can be drastically reduced. The condition can be applied even if compatibility is achieved through the construction of a behavioral adapter (mediator).

1 Introduction

Service behaviors are compatible if their composition forms a closed system (every outbound channel of a service is merged to an inbound channel of some other service) and all involved services can execute their control flow completely. Compatibility can be augmented with the requirement that all or certain activities in the participating services can occur or other semantical constraints.

In this paper we show an approach for alleviating the costs of the compatibility check for services modeled with Petri nets using their state equation. The state equation provides a necessary condition for reachability of the final states of the services in the composition under several constraints such as the enabling of some events or choice covering. This result can be applied directly to adapter synthesis [1]. Service adaptation (mediation) is a semi-automatic approach of correcting incompatibilities between services in which transformation rules are provided normally by hand to correct the message flow. The state equation provides a necessary condition for the existence of such an adapter that uses the specified rules.

In the remainder of this article, we first introduce notations for Petri net models for services and the state equation. Section 3 gives the necessary conditions for compatibility and derives other necessary conditions for compatibility under some additional constraints. Section 4 presents a necessary condition to adaptability. Section 5 concludes the paper.

* Supported by German Research Foundation (DFG) under grant WO 1466/11-1

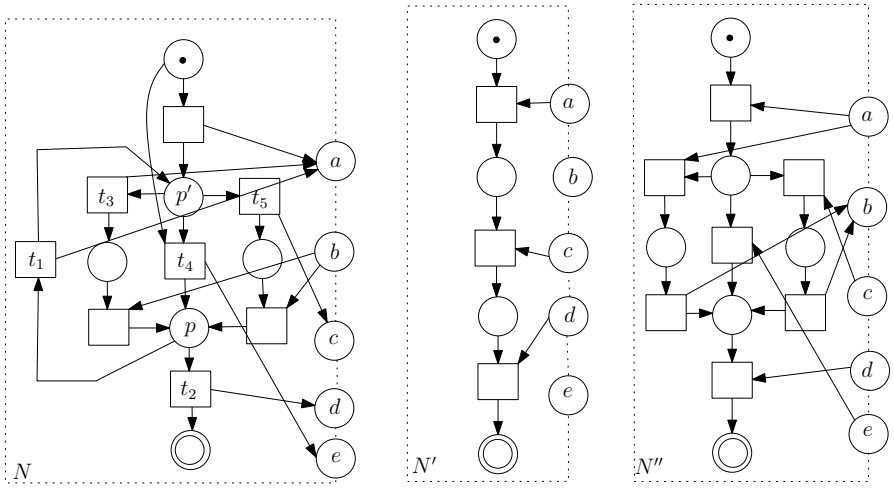


Fig. 1: An open net N and its partner open nets N' and N''

2 Petri nets as models of services and the state equation

Let $\Sigma = \{a, b, c, \dots\}$ be a finite message type set, $?\Sigma = \{?a, ?b, ?c \dots\}$ a finite set of receive events, and $!\Sigma = \{!a, !b, !c \dots\}$ a finite set of send events. We also write $\overline{?\Sigma} = !\Sigma$ and $\overline{\overline{?\Sigma}} = ?\Sigma$.

We consider services modeled as *open nets*. An open net [2] is a Petri net [3] with a special set of interface places which represent the communication channels with other nets.

Definition 1. *An* open net *is a tuple* $N = (P \cup P_i \cup P_o, T, F, m_0, M_f, l)$*, where*

- P, P_i, P_o *are the pairwise disjunct finite sets of internal/input/output places;*
- T *is the finite set of transitions so that* $(P \cup P_i \cup P_o) \cap T = \emptyset$ *which are labeled by the partial function* $l\colon T \to ?\Sigma \cup !\Sigma$*;*
- $F\colon ((P \cup P_i \cup P_o) \times T) \cup (T \times (P \cup P_i \cup P_o)) \to \mathbb{N}$ *represents the flow function so that* $F(p,t) = F(t',p') = 0$*, for all* $(p,t) \in P_o \times T$ *and* $(t',p') \in T \times P_i$*;*
- m_0, M_f *represent the initial state (marking) and the finite set of final states, respectively. We consider states as vectors over the set of places.*

An open net is called closed when its interface is empty, i.e. $P_i \cup P_o = \emptyset$. The projection of an open net on its transitions and internal places is a closed net denoted by $\widehat{N}$. Open nets over $?\Sigma \cup !\Sigma$ are composed [2] by merging their interface places (i.e. an input and with an output place denoting the same message channel) and is denoted by $\oplus$, with the corresponding initial and final markings. Figure 1 shows three open nets N, N', N'', each with the final marking with a token on its double circled place.

A transition $t \in T$ is *enabled* in a marking m if $F(p,t) \leq m(p)$ for all places p. An enabled transition may fire yielding a (reachable) marking m' so that

$m'(p) = m(p) - F(p,t) + F(t,p)$ for all places p, which is denoted by $m \xrightarrow{t} m'$. The reachability relation can be extended to sequences of transitions $\sigma \in T^*$, which is denoted by $\xrightarrow{\sigma}$. Two open nets are called *compatible* if their composition weakly terminates, i.e. from each state reachable from the initial state of the composition, it is possible to reach a final state of the composition. A weaker notion of compatibility is deadlock-freedom, i.e. at each non-final reachable state (in the composition) it is possible to fire a transition.

Reachability analysis for Petri nets can be achieved by using typical structural methods, e.g. methods which find algebraic approximations of the state space with finite representation. The *state equation* [4] relates the behavior of a net (given by states and firing sequences) and its structure (incidence matrix) and can be solved using standard linear programming [5].

The *incidence matrix* $C_N \in \mathbb{N}^{(P \cup P_i \cup P_o) \times T}$ is defined by $C_N(p,t) = F(t,p) - F(p,t)$ for all $(p,t) \in (P \cup P_i \cup P_o) \times T$. Let $\sigma \in T^*$ be transition sequence. The Parikh vector of σ is a vector $\bar{\sigma} \in \mathbb{N}^T$ which assigns to each transition $t \in T$ its number of occurrences in σ. Let $\bar{\sigma}(a) = \sum_{t \in T : l(t) = a} \bar{\sigma}(a)$ denote the number of occurrences of all transitions labeled by $a \in !\Sigma \cup ?\Sigma$. Given a firing sequence $m \xrightarrow{\sigma} m'$ of N, the firing equations for all places of N and all transitions in σ can be written in matrix form $m' = m + C \cdot \bar{\sigma}$, which is called the *state equation*.

Proposition 1 (Necessary condition for reachability). *For every finite firing sequence $m \xrightarrow{\sigma} m'$ of N, the state equation $m' = m + C_N \cdot \bar{\sigma}$ holds.*

3 Necessary condition for compatibility

We state now a necessary condition for compatibility as weak termination of two composed open nets. The first conditions represent the state equations of the open nets without their interface places. The last condition means that in all solutions to the equation the number of occurrences of receiving events should be equal to the number of occurrences for sending events for each such event.

Corollary 1. *If N and N' are compatible (w.r.t. weak termination), then the system $LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m'_0, m_f, m'_f, x, x')$ is feasible.*

$$LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m'_0, m_f, m'_f, x, x'): \quad \begin{array}{ll} m_f = m_0 + C_{\widehat{N}} \cdot x & x \in \mathbb{N}^T \\ m'_f = m'_0 + C_{\widehat{N'}} \cdot x' & x' \in \mathbb{N}^{T'} \\ x(a) = x'(\bar{a}) & \forall a \in ?\Sigma \cup !\Sigma \end{array}$$

If the equation does not have any solution then the final marking will not be reachable in the composition from the initial marking.

Remark 1. In case services have more final states, separate systems of equations are solved for each possible combination. For the nets N and N' in Figure 1 $LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m'_0, m_f, m'_f, x, x')$ does not have any solution. Therefore, N and N' are incompatible. Note that the converse does not hold, e.g. the nets N and N'' in Figure 1, $x''(?a) = x(!a) = 2$, $x(?b) = x''(!b) = 1$, $x(!d) = x''(?d) = 1$, $x(!c) = x''(?c) = 0$ and $x(!e) = x''(?e) = 0$ is a solution for $LP(C_{\widehat{N}}, C_{\widehat{N''}}, m_0, m''_0, m_f, m''_f, x, x'')$, however N and N'' are incompatible as we shall see in the remainder.

If $N \oplus N'$ is deadlock-free then at each non-final reachable marking in the composition there is an enabled transition, i.e. adding the disabling condition for each transition leads to an infeasible system.

Corollary 2 (deadlock-freedom). *If $N \oplus N'$ is deadlock-free then the following system of inequations has no solution:*

$$\begin{aligned} m &= m_0 + C_{\widehat{N}} \cdot x & x \in \mathbb{N}^T, m \in \mathbb{N}^P \\ m' &= m'_0 + C_{\widehat{N'}} \cdot x' & x' \in \mathbb{N}^{T'} m' \in \mathbb{N}^{P'} \\ x(a) &= x'(\bar{a}) + m''(p_a) & \forall a \in ?\Sigma \cup !\Sigma \\ m <> m_f \wedge m' <> m'_f \wedge m'' <> 0^{P_i \cup P_o} & & m'' \in \mathbb{N}^{P_i \cup P_o} \\ \bigvee_{p:F_{N \oplus N'}(p,t)>0}((m+m'+m'')(p) < F_{N \oplus N'}(p,t)) & & \forall t \in T \cup T' \end{aligned}$$

3.1 Necessary conditions for compatibility under constraints

Several variations for compatibility notions have been introduced [6–8] which define behavioral constraints which can imposed on interacting services. Among these settings we mention transition cover and place cover.

Message and event cover

Definition 2. *We call an action a in $?\Sigma \cup !\Sigma$* covered locally/globally *iff a transition/all transitions labeled by a in the composition eventually becomes enabled in the composition. A message place (channel) $p \in P_i \cup P_o$ is called covered if $m(p) > 0$, for some reachable marking m in the composition.*

Let N and N' be two open nets and $a \in ?\Sigma \cup !\Sigma$. We state now conditions which should be added to $LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m'_0, m_f, m'_f, x, x')$ to enforce local, global event cover, place and message cover.

local event cover $x(t) > 0$ ($t \in T : l(t) = a$) or $x'(t') > 0$ ($t' \cup T' : l(t) = a$);
place cover for $p \in P$ there exists a $t \in T$ so that $F(p,t) > 0$ and $x(t) > 0$ (similarly if $p \in P'$);
global event cover $x(t) > 0$, for all $t \in T : l(t) = a$ or $x'(a) > 0$;
message channel cover $x(a) > 0$ and $x'(\bar{a}) > 0$.

In $N \oplus N''$ in Figure 1, a is locally covered but not globally covered (transition t_1). The message channel e is covered neither in $N \oplus N'$ nor in $N \oplus N''$.

Free-choice sending cover Here, we want to strengthen the previously stated condition by taking into account that compatibility does not refer to a single execution (as the state equation would suggest). If an execution passes an internal decision of one service then its partner needs to be able to react to all possible outcomes for this decision. With the following consideration, we want to incorporate this observation into our condition at least for so-called free-choice decisions [3].

Let $x \in P \cup T$. The conflict cluster $\nu(x)$ of x is the smallest set satisfying (1): $x \in \nu(x)$, (2): $\forall q \in T: {}^\bullet q \cap \nu(x) \neq \emptyset \implies q \in \nu(x)$ and (3): $\forall q \in P: q^\bullet \cap \nu(x) \neq \emptyset \implies q \in \nu(x)$. We write ν when x is clear from the context. A conflict cluster $\nu(x)$ so that $|\nu(x)| > 2$ is called a sending free-choice conflict cluster (SC) iff for all $t_1, t_2 \in \nu \cap T$, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ implies ${}^\bullet t_1 = {}^\bullet t_2$ and $l(t) \in !\Sigma$ for all $t \in T \cap \nu$. In Figure 1 $\{p, t_1, t_2\}$ represents such a SC in N. Note that a SC in $\widehat{N}$ is also a SC in N.

A SC in the composition of two nets N and N' is called covered if each transition of the SC is in some firing sequence from the initial marking to the final marking of the composition. For compatible partners, every reachable SC in a service should be resolved by the partner.

Corollary 3. *Let ν be a SC with $\nu \cap T = \{t_1, t_2\}$ in N. If N and N' are compatible and ν is covered in $N \oplus N'$, then $CLP(C_{\widehat{N}}, C_{\widehat{N'}}, \nu)$ is feasible.*

$$
\begin{array}{ll}
 & LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m_0', m_f, m_f', x, x') \\
 & LP(C_{\widehat{N}}, C_{\widehat{N'}}, m_0, m_0', m_f, m_f', \bar{x}, \bar{x}') \\
CLP(C_{\widehat{N}}, C_{\widehat{N'}}, \nu): & x(t_1) > 0 \wedge x'(t_2) > 0 \\
 & \{\nu' \ SC \text{ in } \widehat{N'} | \nu' \cap T' = \{t_1', t_2'\} \wedge \bar{x}(t_1') > 0 \wedge \bar{x}'(t_2') > 0 \wedge \\
 & \qquad \wedge l'(t_1') = \overline{l(t_1)} \wedge l'(t_2') = \overline{l(t_2)}\}
\end{array}
$$

The last condition checks for the existence of a conflict cluster ν' receiving the messages sent by ν. The open nets N and N'' in Figure 1 are incompatible as $CLP(C_{\widehat{N}}, C_{\widehat{N'}}, \nu)$ has no solutions (the choice between the transitions labeled by $!a$ and $!d$ in N'' is not covered) even if $LP(C_{\widehat{N}}, C_{\widehat{N''}}, m_0, m_0'', m_f, m_f'', x, x'')$ has solutions.

Remark 2 (deadlock-freedom under constraints cover). We can relax the deadlock-freedom condition in Corollary 2 to express a necessary condition for local event (transition) cover and SC cover:

t **cover** $\bigvee_{p:F_{N\oplus N}(p,t)>0}(m + m' + m'')(p) \geq F_{N\oplus N'}(p,t)$, where $t \in T \cup T'$;
SC **cover** $\bigvee_{p:F_{N\oplus N}(p,t)>0}(m + m' + m'')(p) \geq F_{N\oplus N'}(p,t)$ for all $t \in \nu$.

Remark 3 (behavioral SC). The transition t_4 of N in Figure 1 is dead and removing it from N does not influence compatibility of N with any other partner. Hence we can consider "behavioral" SC's (e.g. $\{p', t_3, t_5\}$) to be checked for cover.

4 Necessary condition for adapter synthesis

The open nets N_1 and N_2 in Figure 2 do not satisfy the necessary condition in Corollary 1, hence they are incompatible. Adapters are used to solve incompatibilities between interacting services. We consider here the approach in [1] with weak termination as compatibility notion, where adapters are partially specified by transformation rules on messages called SEA (Specification of Elementary Actions). A general rule is described by $r: x \mapsto x'$, where $x \in \mathbb{N}^{!\Sigma}$ and $x' \in \mathbb{N}^{?\Sigma}$.

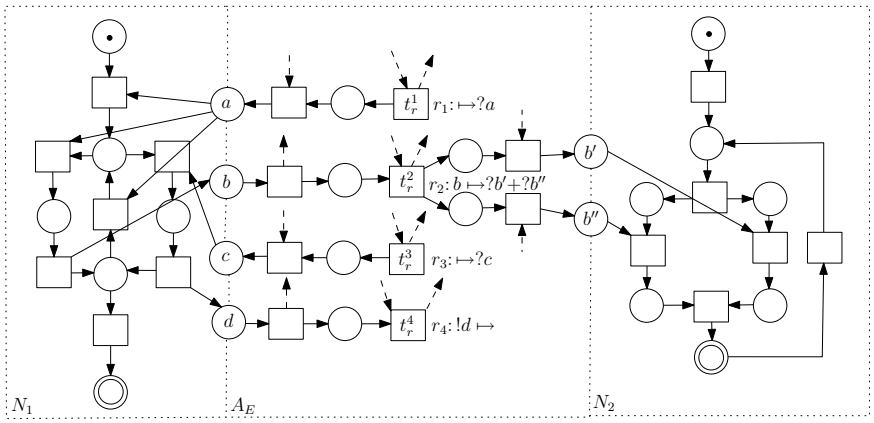


Fig. 2: Two open nets N_1 and N_2 and their partial adapter A_E

The example in Figure 2 shows typical transformation rules: creation of a message (e.g. $!d \mapsto$), deletion of a message ($\mapsto ?c$), splitting a message into two messages ($!b \mapsto ?b' + ?b''$). Each transformation rule is transformed into an open net which communicates with the initial services and with an entity which controls the application of these rules (e.g. the transition t_r^1) and the sending/receiving of messages (denoted by dashed arrows). The open net obtained from the transformation rules is called partial adapter A_E. The adapter synthesis procedure computes a partner C which controls $N_1 \oplus A_E \oplus N_2$ and the final adapter is $C \oplus A_E$.

A direct consequence of Corollary 1 is that compatible partners have a solution to their own state equation. We state this condition for the adapter setting.

Corollary 4. *If N_1 and N_2 are adaptable by the set of transformation rules R, then the state equation for $\mathcal{N}_1 \oplus \widehat{A_E} \oplus N_2$ with initial marking $m_0^1 + m_0^2$ and final marking $m_f^1 + m_f^2$ holds.*

The state equation for $N_1 \oplus \widehat{A_E} \oplus N_2$, where A_E is the partial adapter for the rules $\{r_1, r_3, r_4\}$, does not yield any solution, thus N_1 and N_2 are not adaptable by $\{r_1, r_3, r_4\}$.

In addition, we can formulate a necessary condition for transformation rule cover. Let $r\colon \sigma \longrightarrow \sigma'$. We add to the state equation of $N_1 \oplus \widehat{A} \oplus N_2$ the constraint $x(t_r) > 0$, where t_r is the transition corresponding to the application of the rule. Thus, we can eliminate rules which will never be fired in conjunction with a proper terminating execution. In Figure 2, r_3 and r_4 are redundant rules.

5 Conclusion

In this paper we stated some necessary conditions for service compatibility using the state equation for Petri nets. The advantage of using this approach to state

space methods (e.g. [9–11]) is its lower computational complexity [5] (polynomial for real solutions/exponential in the worst case for integer solutions). An area of application for this approach is service discovery and service composition [8, 2], i.e. finding well-behaved partners for a particular service in a repository of services. Service discovery and composition are inherently costly job (both from time and space) w.r.t. the size of the repository and of the services themselves. Using such a quick check can ease the task of a broker for discovering/adapting potentially compatible partners for a service by disposing of those services which do not satisfy the necessary criterion.

The approach presented in this paper allows for (in)compatibility to be analyzed in a compositional way (incorrectness of a component can be used to derive the incorrectness of the composition). This is complementary to structural methods used in soundness analysis [12, 13] of monolithic workflow. As future work, we plan to implement the state equation approach as a preliminary check for service composition and adaptability and evaluate the efficiency of this approach in the large on a set of case studies provided by industry.

References

1. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Technical Report CS-02-08, Universität Rostock, Rostock, Germany (2008)
2. Wolf, K.: Does my service have partners? LNCS ToPNoC **5460**(II) (2009) 152–171 Special Issue on Concurrency in Process-Aware Information Systems.
3. Desel, J., Esparza, J.: Free Choice Petri nets. Volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1995)
4. Schmidt, K.: Narrowing Petri net state spaces using the state equation. Fundam. Inform. **47**(3-4) (2001) 325–335
5. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience series in discrete mathematics. John Wiley & Sons (1986)
6. Wolf, K.: On synthesizing behavior that is aware of semantical constraints. In: Proceedings of AWPN 2008. Volume 380 of CEUR Workshop Proceedings., CEUR-WS.org (2008) 49–54
7. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007. Volume 4714 of LNCS. (2007) 271–287
8. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. (2008) (Accepted).
9. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: WWW '04, ACM (2004) 621–630
10. Schlingloff, B.H., Martens, A., Schmidt, K.: Modeling and model checking web services. Electr. Notes Theor. Comput. Sci. **126** (2005) 3–26
11. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: ICSOC. Volume 5364 of LNCS. (2008) 84–99
12. K. van Hee, Oanea, O., Sidorova, N., Voorhoeve, M.: Verifying generalized soundness for workflow nets. In: PSI. Volume 4378 of LNCS., Springer (2007) 235–247
13. Verbeek, H.M.W., van der Aalst, W.M.P.: Woflan 2.0: A Petri-net-based workflow diagnosis tool. In: ATPN 2000. Volume 1825 of LNCS., Springer (2000) 475–484

Umstrukturierung von WS-BPEL-Prozessen zur Verbesserung des Validierungsverhaltens

Thomas S. Heinze[1], Wolfram Amme[1], Simon Moser[2]

[1] Friedrich-Schiller-Universität Jena
{T.Heinze,Wolfram.Amme}@uni-jena.de
[2] IBM Entwicklungslabor Böblingen
smoser@de.ibm.com

1 Einführung

Innerhalb der letzten Jahre wurde eine Vielzahl von Methoden zur Analyse von verteilten Geschäftsprozessen der Sprache *Web Services Business Process Execution Language (WS-BPEL)* [2] entwickelt [3]. Mit Ausnahme einzelner Ansätze konzentrieren sich die meisten der Techniken auf die Analyse des Kontrollflusses und ignorieren die Datenabhängigkeiten der untersuchten Prozesse. Ein solches Vorgehen birgt aber die Gefahr der Verfälschung von Analyseergebnissen in sich. Insbesondere für die Analyse von Eigenschaften wie der *(Verhaltens-) Kompatibilität* [6] ist die Berücksichtigung der Datenabhängigkeiten von Bedeutung.

Ein Prozessfragment dass nicht fehlerfrei analysiert werden kann, falls Datenabhängigkeiten ignoriert werden, ist in Abbildung 1 dargestellt. Die abgebildete Aktivität `OrderingSequence` ist möglicherweise Bestandteil eines größeren Geschäftsprozesses zur Realisierung eines Online-Shop. Darin kann ein Kunde mehrere Bestellungen aufgeben (Nachricht `Order`), und so die Auftragsabwicklung (Aktivität `OrderProcessing`) zu jeder Bestellung einleiten. Nachdem der Kunde alle Bestellungen übertragen hat, kann er den Bestellvorgang beenden (Nachricht `Complete`). Zur Umsetzung enthält die Aktivität `OrderingSequence` eine Schleife (`While`), deren Ausführung durch die boolesche Variable `doOrder` gesteuert wird. Anfangs wird die Variable mit dem Wert true belegt und die Schleife daher durchlaufen. Die `Pick`-Aktivität innerhalb der Schleife führt dann entweder die Sequenz `OrderProcessing` aus, falls Nachricht `Order` empfangen wird, oder die Sequenz `Termination`, falls der Kunde die Nachricht `Complete` übermittelt. Im letzten Fall wird der Wert von `doOrder` auf false gesetzt und so die Schleife beendet. Der Schleifenabbruch wird demzufolge durch Empfang der Nachricht `Complete` ausgelöst, entsprechend nennen wir das verallgemeinernde Muster auch *nachrichtengesteuerter Schleifenabbruch*. Da im Sprachumfang von WS-BPEL derzeit kein Gegenstück zur break-Anweisung aus Sprachen wie Java enthalten ist [2], kann dieses Muster nur unter Verwendung einer Schleife mit einer booleschen Variablen als Schleifenbedingung realisiert werden.

Die in der Arbeit [6] angegebene und auf Petrinetzen basierende Kompatibilitätsanalyse führt für Prozesse, die Fragmente dieses Muster enthalten, zu fehlerhaften Ergebnissen. Dazu kann beispielsweise ein Partner zur Aktivität

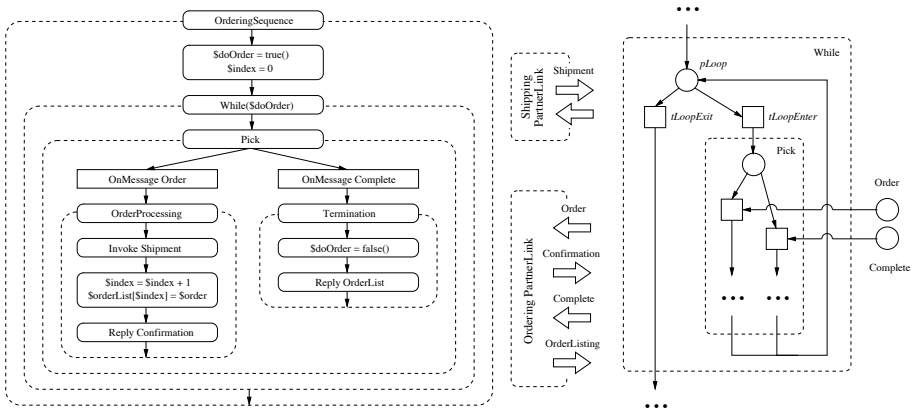


Abb. 1. `OrderingSequence` (links) und Teil des zugehörigen Petrinetzmodells (rechts)

`OrderingSequence` betrachtet werden, der genau eine Bestellung aufgibt. Die Kommunikation besteht dann aus den aufeinanderfolgenden Nachrichten `Order`, `Confirmation`, `Complete` und `OrderList`. Offensichtlich sind die beiden Fragmente kompatibel [6], da es zu keiner Verklemmung kommen kann. Die Kompatibilitätsanalyse kommt aber zum gegenteiligen Schluss. Um die Analysierbarkeit des darin verwendeten Petrinetzmodells zu gewährleisten, werden bedingte Schleifen und Verzweigungen durch Nichtdeterminismus modelliert. Die Schleife in `OrderingSequence` wird demnach auf die in Konflikt stehenden Transitionen *tLoopEnter* und *tLoopExit*, zur Repräsentation des Schleifenein- und -austritts, abgebildet (siehe auch Abbildung 1). Da der Konflikt willkürlich zu lösen ist, kann die Schleife beliebig oft durchlaufen werden. In der Folge ist möglich, dass die Aktivität `OrderingSequence` weitere Bestellungen erwartet, obwohl Nachricht `Complete` bereits empfangen wurde. Es kommt zu einer Verklemmung und die Analyse zum Ergebnis, dass die beiden Fragmente nicht kompatibel sind.

Zusammenfassend ist die Kompatibilitätsanalyse in [6] im Hinblick auf das beschriebene Muster nachrichtengesteuerter Schleifenabbruch fehleranfällig. Das Weglassen der Datenabhängigkeiten von bedingten Schleifen und Verzweigungen bedeutet eine zu starke Abstraktion innerhalb der verwendeten Petrinetzmodellierung. Um die Zahl der dadurch verursachten Analysefehler zu verringern, schlagen wir eine *Umstrukturierungsmethode* für Geschäftsprozesse der Sprache WS-BPEL vor. Diese stellen wir im Folgenden anhand des nun eingeführten Prozessfragments `OrderingSequence` vor. Die Methode erlaubt bedingte Schleifen immer dann so zu transformieren, dass deren Datenabhängigkeiten in Kontrollabhängigkeiten umgewandelt werden können, wenn deren Schleifenbedingungen zur Laufzeit nur auf Konstanten beliebigen Typs zugreifen. Das Resultat dieser Transformation ist ein semantisch äquivalenter Prozess, indem die Datenabhängigkeiten der Schleifen, dass heißt deren Bedingungen, entfernt werden können. Derart lässt sich die Anzahl von nichtdeterministischen Strukturen im Petrinetzmodell verringern und so diese mögliche Fehlerquelle einschränken.

2 Prozessrepräsentation

Um eine verlustfreie Repräsentation von Geschäftsprozessen der Sprache WS-BPEL zu ermöglichen, verwenden wir eine Erweiterung von *Workflow-Graphen*. Workflow-Graphen [8] werden häufig zur Analyse von Geschäftsprozessen genutzt, repräsentieren aber nur deren Kontrollfluss. Durch Anreicherung mit einem weiteren Repräsentationsformat, der *Concurrent Static Single Assignment Form (CSSA-Form)* [5, 7], lassen sich auch die Datenabhängigkeiten modellierter Prozesse wiedergeben. Wir nutzen daher eine Kombination beider Formate.

In Abbildung 2 ist der so erweiterte Workflow-Graph für das oben beschriebene Prozessfragment `OrderingSequence` dargestellt. Darin modellieren Knoten die Aktivitäten und Kanten verbinden die Knoten gemäß dem Kontrollfluss. Elementare Aktivitäten (beispielsweise `Reply Confirmation`) werden unter Verwendung eines einzelnen Knotens abgebildet. Sequenzen elementarer Aktivitäten sind dann durch mehrere sukzessive verbundener Knoten repräsentiert. Im Fall der Verzweigung `Pick` werden spezielle Knoten genutzt, um die Aufspaltung ($Pick$) und Vereinigung ($Merge$) des Kontrollflusses darstellen zu können. Dies gilt auch für die enthaltene Schleife `While`, für die der Knoten zum Aufspalten des Kontrollflusses ($Branch$) die Schleifenbedingung enthält und der Knoten zur Vereinigung ($Header$) des Kontrollflusses als Schleifenkopf bezeichnet wird.

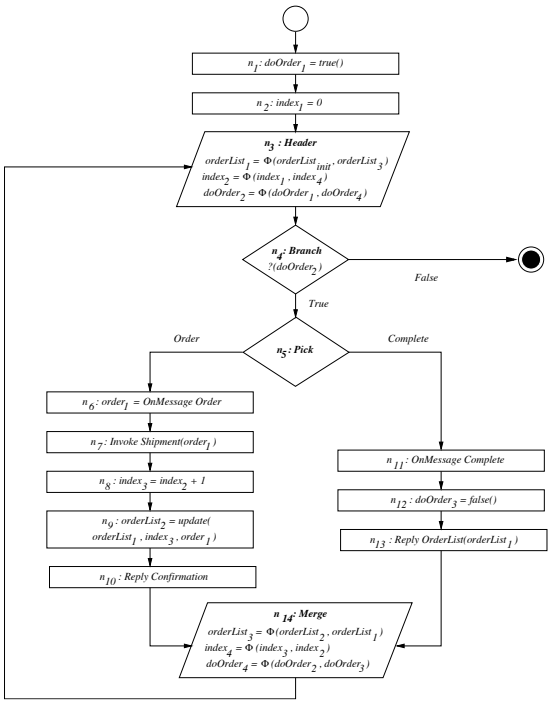


Abb. 2. Erweiterter Workflow-Graph

Grundlegende Eigenschaft der CSSA-Form ist, dass Variablen (statisch) nur einmal definiert werden dürfen.[1] Zu diesem Zweck werden die Variablen in `OrderingSequence` so umbenannt, dass jede Variablendefinition einen eigenen Namen besitzt (beispielsweise $doOrder_1, \ldots, doOrder_4$ für `doOrder`). Dadurch verhalten sich alle Variablen wie konstante Werte. Insbesondere sind Beziehungen zwischen Definition und Gebrauch einer Variablen nun explizit wiedergegeben. Ein besonderes Vorgehen ist notwendig, falls mehrere Definitionen einer Variablen auf verschiedenen Pfaden des Kontrollflusses in einem Knoten zusammentreffen (so in $Merge$ und $Header$). In diesem Fall werden Φ-Funktionen eingefügt, um die konkurrierenden Definitionen zusammenzufassen (beispielsweise $doOrder_4 = \Phi(doOrder_2, doOrder_3)$ in $Merge$). Die Operanden einer Φ-Funktion bilden gerade die Variablendefinitionen und der Funktionswert entspricht der Definition, die zur Laufzeit tatsächlich ausgeführt wurde.

3 Umstrukturierung

Aufbauend auf dieser Prozessrepräsentation lassen sich die Bedingungen von Verzweigungen und Schleifen analysieren. Die Bedingung der in `OrderingSequence` enthaltenen Schleife entspricht genau der Variablen $doOrder_2$. Deren Wert wird durch eine Φ-Funktion im Schleifenkopf $Header$ definiert. Diese führt die konkurrierenden Definitionen der Variablen vor Ausführung der Schleife ($doOrder_1$) und nach Ausführung eines Scheifendurchlaufs ($doOrder_4$) zusammen. Dabei ist die Definition nach Ausführung eines Durchlaufs ebenfalls durch eine Φ-Funktion angegeben, die die Werte auf den zwei möglichen Pfaden innerhalb der Schleife zusammenfasst ($doOrder_2$, $doOrder_3$). Da alle Definitionen entweder einer Konstantenzuweisung oder einer Φ-Funktion entsprechen, hängt der Wert von $doOrder_2$ lediglich vom Pfad des Kontrollflusses zur Laufzeit ab: Wird die Schleife zum ersten Mal ausgeführt, wird $doOrder_2$ der Wert von $doOrder_1$, also true, zugewiesen und die Schleifenbedingung daher erfüllt. Dasselbe gilt für jeden weiteren Durchlauf, solange bis die Zuweisung in Knoten n_{12} ausgeführt wird. Danach wird $doOrder_2$ der Wert von $doOrder_3$, also false, zugewiesen. In der Folge ist die Bedingung nicht mehr erfüllt und die Schleife wird abgebrochen.

Wir nennen Schleifenbedingungen dieser Art, in denen alle Variablen ausschließlich durch ineinander geschachtelte Φ-Funktionen und Konstantenzuweisungen definiert sind, *dynamisch konstant*. Da der Wert einer solchen Bedingung nur vom zur Laufzeit ausgeführten Kontrollflusspfad abhängig ist, können die Datenabhängigkeiten der Bedingung offenbar auch durch Kontrollabhängigkeiten repräsentiert werden. Durch eine geeignete Transformation der zugehörigen Schleife lassen sich die entsprechenden Kontrollabhängigkeiten erzeugen.[2] Auf diese Weise wird die Schleifenbedingung redundant und kann innerhalb des umstrukturierten Prozessfragments entfernt werden.

[1] Aufgrund der statischen Betrachtungsweise werden auch Variablendefinitionen innerhalb von Schleifen als einmalige Definitionen angesehen.

[2] Als eine Einschränkung der Umstrukturierungsmethode werden solche Φ-Funktionen ausgeschlossen, die innerhalb von Schleifenköpfen anderer Schleifen definiert werden.

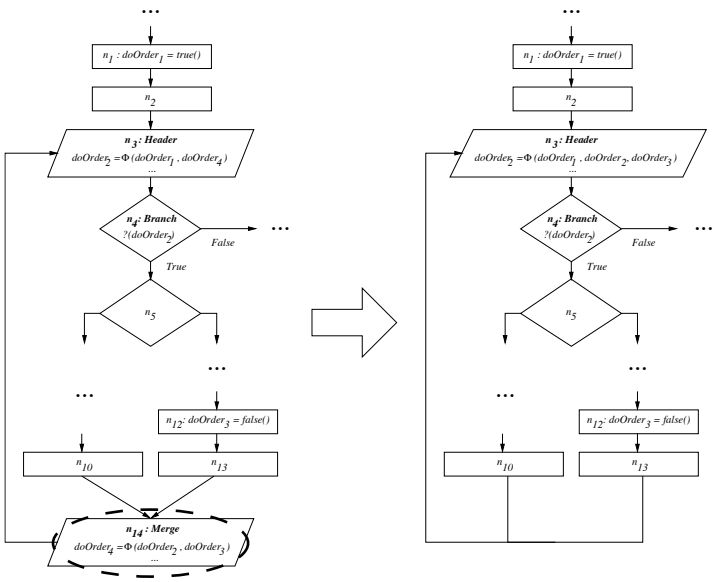


Abb. 3. Überführung der Schleife in Normalform

Vor der eigentlichen Transformation einer Schleife mit dynamisch konstanter Schleifenbedingung, wird diese in *Normalform* überführt. Die Normalform ist durch die Auftrennung aller Pfade des Kontrollflusses charakterisiert, auf denen in einem beliebigen Schleifendurchlauf unterschiedliche Werte für die Variablen der Schleifenbedingung definiert werden. Diese Überführung ist für die Schleife des Prozessfragments `OrderingSequence` in Abbildung 3 dargestellt. Die Variable der dort vorhandenen Bedingung ($doOrder_2$) kann, wie oben beschrieben, für einen beliebigen Schleifendurchlauf drei verschiedene Werte annehmen. Die Wahl des Wertes wird dabei durch den zuvor ausgeführten Kontrollflusspfad bestimmt. Um nun die Pfade aufzutrennen, muss der Knoten $Merge$ aufgelöst werden, da er zwei der drei möglichen Werte zusammenführt ($doOrder_2$ und $doOrder_3$). Da der unmittelbare Nachfolger dieses Knotens gerade dem Schleifenkopf $Header$ entspricht, reicht es dazu aus, die Vorgängerknoten n_{10} und n_{13} direkt mit dem Kopf zu verbinden und die Operanden der darin enthaltenen Φ-Funktionen anzupassen (Operand $doOrder_4$ durch $doOrder_2$ und $doOrder_3$ ersetzen). Anschließend kann der Knoten $Merge$ entfernt werden.

Die Normalform wird dann als Blaupause im folgenden Transformationsschritt genutzt. In diesem Schritt werden mehrere Instanzen der Blaupause erzeugt und miteinander verbunden. Wir nennen den Schritt dementsprechend *Schleifeninstanziierung*. Jede Instanz repräsentiert eine mögliche Belegung der in der Schleifenbedingung genutzten Variablen mit konstanten Werten. Für die Schleife in `OrderingSequence` werden so zwei Instanzen erzeugt, wie in Abbildung 4 dargestellt. In der ersten Instanz ($Instance\,1$) wird die Variable $doOrder_2$ durch den Wert true ersetzt, in der zweiten ($Instance\,2$) durch den Wert false.

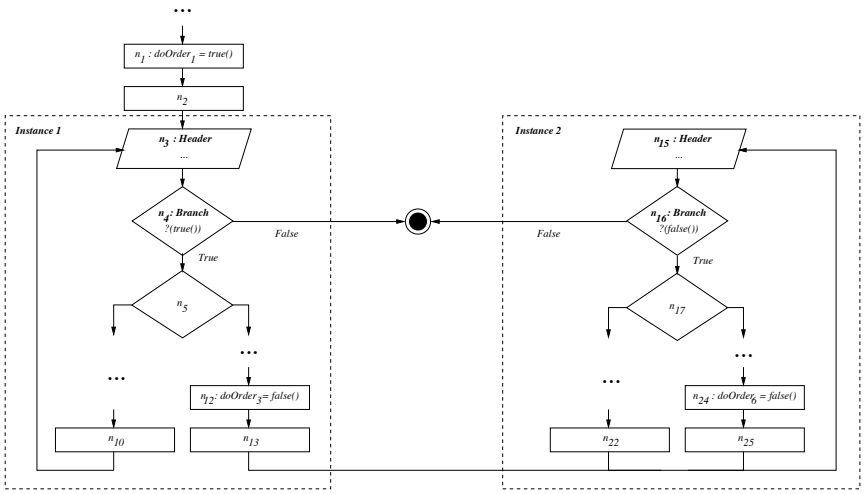


Abb. 4. Transformation unter Verwendung von Schleifeninstanzen

Dadurch ist die Schleifenbedingung in beiden Instanzen statisch auswertbar. In der Folge lassen sich die Bedingung und die unmöglichen Kontrollflusspfade in `OrderingSequence` entfernen (Kanten (n_4, End) und (n_{16}, n_{17})). Das Ergebnis dieser Umstrukturierung ist in Abbildung 5 angegeben. Offenbar ist darin der Abbruch der Schleifenausführung, nach Empfang der Nachricht `Complete`, explizit durch den Kontrollfluss wiedergegeben. Innerhalb des zugehörigen Petrinetzmodells [1] kann daher auf die Verwendung nichtdeterministischer Konflikte verzichtet, und so eine folgende Kompatibilitätsanalyse präzisiert werden.

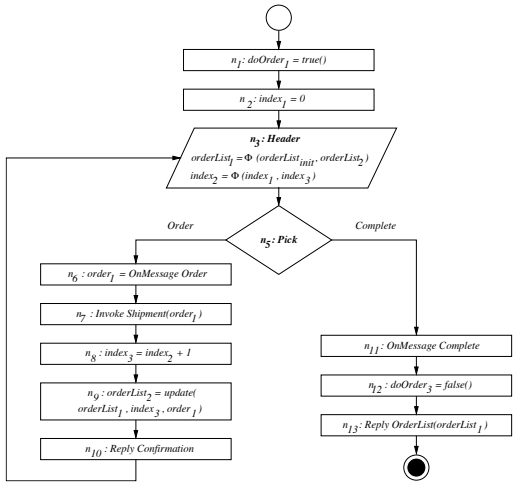


Abb. 5. Umstrukturiertes Prozessfragment

4 Zusammenfassung

Der vorliegende Beitrag beschreibt eine bereits vollständig entwickelte Umstrukturierungsmethode für verteilte Geschäftsprozesse der Sprache WS-BPEL. Die vorgestellte Methode ist in der Lage die Datenabhängigkeiten einer bestimmten Art von Schleifen in semantisch äquivalente Kontrollabhängigkeiten zu transformieren. Auf diese Weise können die Prozessmodelle bestehender Analysen präzisiert und so die Verfälschungen von Analyseergebnissen verringert werden.

Die vorgestellte Umstrukturierungsmethode kann auch auf Verzweigungsbedingungen angewendet werden und ist nicht auf Bedingungen mit einer einzelnen Variablen oder auf boolesche Variablen beschränkt. Eine detaillierte Darstellung der Methode, einschließlich der Beschreibung verwendeter Algorithmen und einem Korrektheitsbeweis, erfolgt in einem technischen Bericht [4].

Literatur

[1] Aalst, W. M. P. d. ; Hirnschall, A. ; Verbeek, H. M. W.: An Alternative Way to Analyze Workflow Graphs. In: Pidduck, A. B. (Hrsg.) ; Woo, C. (Hrsg.) ; Mylopoulos, J. (Hrsg.) ; Ozsu, M. T. (Hrsg.): *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002), May 27-31, 2002, Toronto, Canada*, Springer-Verlag, 2002 (LNCS 2348), S. 535–552

[2] Alves, Alexandre ; Arkin, Assaf ; Askary, Sid ; Barreto, Charlton ; Bloch, Ben ; Curbera, Francisco ; Ford, Mark ; Goland, Yaron ; Guízar, Alejandro ; Kartha, Neelakantan ; Liu, Canyang K. ; Khalaf, Rania ; König, Dieter ; Marin, Mike ; Mehta, Vinkesh ; Thatte, Satish ; Rijn, Danny van ; Yendluri, Prasad ; Yiu, Alex: *Web Services Business Process Execution Language Version 2.0*. 2007

[3] Breugel, Franck van ; Koshkina, Maria: *Models and Verification of BPEL*. 2006

[4] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: Resolving Conditional Branches in WS-BPEL Business Processes / Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik. 2009. – noch nicht erschienen

[5] Lee, Jaejin ; Midkiff, Samuel P. ; Padua, David A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: Li, Zhiyuan (Hrsg.) ; Yew, Pen-Chung (Hrsg.) ; Huang, Chua-Huang (Hrsg.) ; Chatterjee, Siddharta (Hrsg.) ; Sadayappan, P. (Hrsg.) ; Sehr, David (Hrsg.): *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC '97), August 7-9, 1997, Minneapolis, Minnesota, USA*, Springer-Verlag, 1998 (LNCS 1366), S. 114–130

[6] Martens, Axel ; Moser, Simon ; Gerhardt, Achim ; Funk, Karoline: Analyzing Compatibility of BPEL Processes. In: *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), February 19-25, 2006, Guadeloupe, French Caribbean*, IEEE Computer Society Press, 2006, S. 147

[7] Moser, Simon ; Martens, Axel ; Görlach, Katharina ; Amme, Wolfram ; Godlinski, Artur: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In: *Proceedings of the 2007 IEEE International Conference on Services Computing (SCC 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, IEEE Computer Society Press, 2007, S. 98–105

[8] Sadiq, Wasim ; Orlowska, Maria E.: Analyzing Process Models Using Graph Reduction Techniques. In: *Information Systems* 25 (2000), Nr. 2, S. 117–134

Improving Control Flow Verification in a Business Process using an Extended Petri Net

Ganna Monakova, Oliver Kopp, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{monakova, kopp, leymann}@iaas.uni-stuttgart.de

Abstract. In a business process, control flow decisions are based on the evaluation of conditions. Thus, conditions must be considered for control flow verification. This paper shows how the Petri nets based control flow verification can be improved by analysing conditions and logical relations between them. We outline a Petri net extension with predicate transitions, which are responsible for conditions evaluation based on the collected knowledge, and effect places, which contain fact tokens representing the effects of certain operations and decisions made.

1 Introduction

As the complexity of business processes grows, the need for automatic verification becomes more important. We show an approach for verification for processes modelled with Petri nets. The properties to verify represent the constraints on possible execution traces. An example of such a constraint is "the payment must always be followed by a shipment", which can be expressed using LTL as $G(Payment \rightarrow FShipment)$. To show that this constraint is fulfilled, it must be proved that there is no possible execution path that contains a payment before a shipment (temporal dependency) and that there is no execution path that contains payment without shipment (causal dependency). Note that the above constraint allows the execution of shipment without payment.

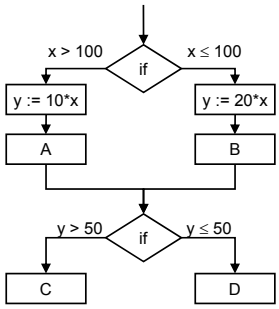


Fig. 1. Example process fragment

The process fragment depicted in Fig. 1 shows why the data relations should be considered for the control flow verification. The fragment presents two `if`-construcrts, each having two branches. We define an acivity *execution condition* as a Boolean expression that is constructed recursively by analyzing all conditions that have to be satisfied to enable the execution of this activity [1]. For example, the execution condition of activity A from Fig. 1 is $(x > 100)$ and the execution condition of B is $(x \leq 100)$.

Assume that the constraint "A must always be followed by C" must be satisfied for a process containing this fragment. This constraint is always satisfied,

since the execution condition of C is implied by the execution condition of A and the fact that y becomes equal to $10 * x$ if the left branch of the first if is executed. Current Petri net based verification approaches abstract from the data and their relations and therefore make non-deterministic choice in both of the switch-constructs, which makes the phantom execution path $(A; D)$ possible.

This paper shows how a Petri net can collect knowledge and use the collected knowledge to reason about the next step. Sect. 2 presents related work in this field. The proposed extensions are shown in Sect. 3. Sect. 4 describes how the process knowledge is collected during the Petri net analysis and Sect. 5 shows how the collected knowledge is used for the predicate transitions evaluation. Sect. 6 demonstrates the analysis of the process fragment shown in Fig. 1 using the presented approach before Sect. 7 concludes.

2 Background and Related Work

A number of non-deterministic decisions take place during the analysis of a Petri net: selection of a specific if-branch, skipping or executing an activity, and entering or exiting a loop. In the business process the decisions depend on the evaluation of the corresponding conditions: branching condition, join condition, loop condition. A join condition is a starting condition of a branch and is a Boolean formula over the states of all incoming links [2]. Typically, the translation of a business process to a Petri net ignores these conditions. The justification is that the actual data coming into the process is not known at static validation and therefore the conditions cannot be evaluated. The process model, however, contains read-write dependencies between the activities, which can be used for an advanced verification [3]. Logical relations between variables can be captured by execution conditions as described in [1]. In this paper, we show how logical relations can be used for a more precise Petri nets based control flow verification. The technique presented in this paper can be used as extension to the mapping of a BPEL process to the corresponding Petri net [4].

An overview of existing BPEL formalizations and verification approaches is provided in [5]. We presented in [1] a summary of the presented approaches and showed that none of them includes the interplay between previous and following decisions. Thus, all of the approaches include the phantom path $(A; D)$ in their analysis. The work of [1] put loops and scopes as future work. In the work presented here, we include loops and scopes, since the mapping of BPEL to Petri nets is complete [4].

3 An Extension for Petri Nets

The conditions constrain the execution of the business process. In addition, the control flow decisions made in the past can influence the decisions in the future, as the example of Fig. 1 illustrates: if the left branch of the first `if`-activity is taken, then y is set to $10 * x$, where $x > 100$ according to the branch condition. This will influence the branch selection of the second `if`-activity. This implies that the execution trace, which contains activity A and activity D, is impossible. It will, however, be considered as possible during the Petri net analysis if the data

conditions are neglected. Such phantom execution trace can only be detected if the relation between decisions leading to the execution of the activities A and D are known.

Each decision is bound to a certain condition. If a decision has to be made, the condition is evaluated and, depending on the evaluation result, a certain path in a workflow is chosen. During process runtime, the evaluation of the condition is simple, since instance data is available. At design time, however, instance data is not available and thus only relations between process variables and between conditions can be analysed. If a certain path in a workflow is chosen, then the condition of this path is true (e.g. $x > y$). Thus, even if we do not know the actual data, we know that the data relations captured in the path condition (e.g. $x > y$) are in force. We also say that the decision produces an *effect* relations.

We add a *predicate transition* to decide whether a certain path *can* be executed. A predicate transition is responsible for the evaluation of the condition for a certain execution trace based on the collected knowledge. A *decision transition* is the transition responsible for selecting the actual execution trace from all possible execution traces. A trace is considered to be an alternative if the corresponding condition was evaluated to true or unknown by the predicate transition, see Sect. 5. We record the effect of the decision made by producing a token for an *effect place* added after each fired decision transition. This token means that the condition on this path is true and thus the relation represented by this condition is in force.

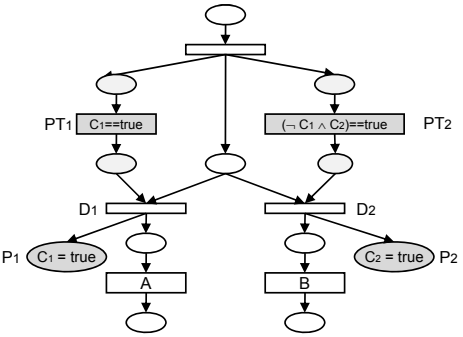
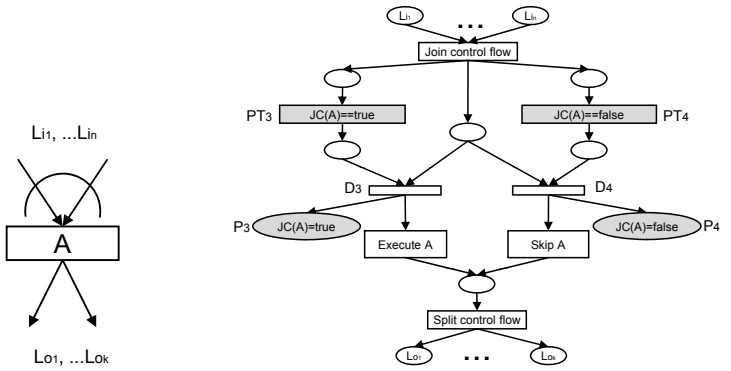


Fig. 2. `if` activity in an extended Petri net

Effect places, predicate and decision transitions for an if-activity with two branches are shown in Fig. 2. The predicate transitions PT_1 and PT_2 are responsible for the evaluation of the branch conditions: C_1 for the left and C_2 for the right branch. If the corresponding condition evaluates to true, the token will be produced in the outgoing place, which in its turn will enable the selection of the corresponding branch. If the condition evaluates to false, then the token from the incoming place will be consumed and no token for the outgoing place will be produced. Note, that the branch conditions are adjusted in such a manner that only one of the conditions can simultaneously evaluate to true. This complies with the `if`-activity execution semantic [6]. If evaluation of both conditions returns *unknown*, both predicate transitions will produce a token and the `if`-branch will be chosen non-deterministically. As soon as the branch selection decision has been made by a decision transition, the relations between process variables represented by the recorresponding branch condition come in force. This is indicated by a token in the corresponding effect place: P_1 for the left and P_2 for the right branch. Note that the effect places cannot be put directly after the decision transitions,



(a) `flow` activity

(b) `flow` activity extended Petri net

Fig. 3. `flow` activity

to honor the fact that both branch conditions can evaluate to unknown and thus the actual decision will be made by the decision transition.

Fig. 3(a) shows an activity in a BPEL `flow`. The activity has n incoming links and a join condition defined on the status of these links. Fig. 3(b) shows an extended Petri net for this activity. Here, the predicate transitions PT_3 and PT_4 are responsible for evaluation of the join condition, which helps the decision transitions D_3 and D_4 to decide whether the activity A is to be executed or skipped. The extension required for loop constructs is similar to the if-activity extension and is not shown in this paper due to the space limitations. An `assign`-activity establishes relations between process variables which are come in force after the activity has been executed. Therefore, an additional effect place is added after the transition representing an `assign` activity. The next section presents how the produced effects can be stored to enable reasoning on the colected knowledge.

4 Collecting Knowledge

The variable and condition relations currently in force are represented by the tokens in the effect places. The effect places, and thus the knowledge about the data relations, may either result from a decision effect or from a relation between process variables introduced by an `assign` activity. A decision effect is a result of a decision made for transition conditions, join conditions, `if` and `pick` branches. Let C be the condition of the path selected by the decision transition. Then a new fact $C = true$ is added to the knowledge base. The relations between process variables captured in the assign statements can also influence decisions. For example, if an assignment $y := x + a$ was executed and it is known that $a > 0$, then it can be derived that $y > x$. If an assign activity $x := f(y_1, \ldots, y_n)$ is executed, then the statement $x = f(y_1, \ldots, y_n)$ becomes a fact and is added to the knowledge base. Thereby, every occurrence of the same variable on the left side of assignment gets a new index each time an assignment fact is added to the knowledge base. If a variable occurs on the right side of the assignment, the variable with the highest index currently available in the knowledge base is used. Note that this is different to the CSSA approach [7], as in this case there is no need to consider the

exclusive or concurrent read-write accesses to the same variable (expressed with the ϕ and π-functions in CSSA). The reason for this is the step-by-step analysis considered in this paper, which implies that the knowledge base cannot contain contradictory information: only the relations captured in the assign statements on the chosen branch are added to the knowledge base and the order of the concurrent assignments is the one selected by the Petri net navigator. A `receive`, `pick` and `invoke` activity can be considered to contain an implicit assign of the message content to the process variables.

5 Reasoning on the Collected Knowledge

A predicate transition represents an invocation of a reasoner. A reasoner evaluates the transition condition based on the current knowledge in the knowledge base. This section shows how the evaluation of the condition can be reduced to the satisfiability problem. Let $F_1, \ldots, F_n$ be the current facts in the knowledge base, let C be the condition to be evaluated. The condition evaluates to true if it can be proved that C can be derived from the current facts in the knowledge base. Formally speaking, the following must hold: $F_1, \ldots, F_n \vdash C$. To prove this, the following formula is checked for its satisfiability: $\left(\bigwedge_{i\in\{1,\ldots,n\}} F_i\right) \wedge \neg C$. If this above formula is unsatisfiable, then C will always evaluate to true for this execution path and therefore a token will be produced by the predicate transition responsible for the evaluation of the condition C. If the above formula is satisfiable, the following formula is checked: $\left(\bigwedge_{i\in\{1..n\}} F_i\right) \wedge C$. If this formula is unsatisfiable, then C will always evaluate to false for this execution path. In this case the transition will not produce any token. If both formulas are satisfiable, then the decision can only be made based on the concrete data and therefore both cases should be considered for the analysis. In this case, the predicate transition produces a token which will compete with other tokens. The actual decision will be made non-deterministically by the decision transition in the same way as for the non-extended Petri net: one of the tokens will be consumed, the others will remain in their places and wait for the backtracking and selection/consumption of the next token. The collected facts in the knowledge base represent the logical relations between process variables currently in force. The satisfiability of the above formulas based on the current relations is checked using the Satisfiability Modulo Theories (SMT) solver Yices [8]. An SMT solver solves satisfiability problems for Boolean formulas containing predicates of underlying theories. Such theories can be, for example, theories of arrays, lists and strings [9]. In addition, an SMT solver can be extended with new theories as shown in [10].

6 Example

Fig. 4 illustrates the analysis of the process fragment from Fig. 1. Fig. 4(a) shows the first invocation of the reasoner on the current knowledge base. Since no previous information is available, the knowledge base is empty and therefore returns *unknown* for both conditions. Fig. 4(b) shows the status after the left branch was non-deterministically taken. This decision transition produces a token

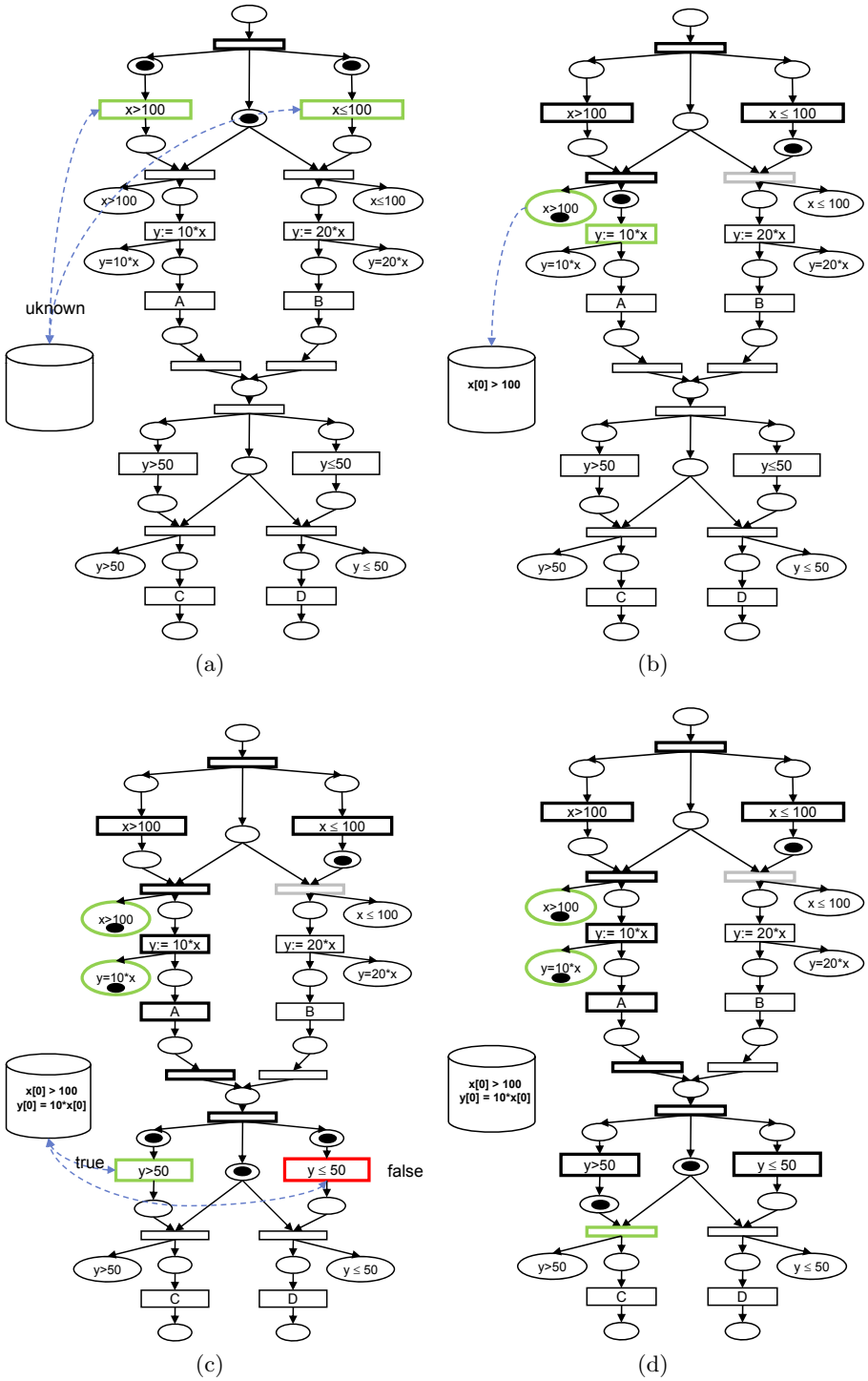


Fig. 4. Analysis using knowledge base and a reasoner

in the effect place $x > 100$ and the corresponding relation is added as a fact to the knowledge base. Fig. 4(c) shows the next invocation of the reasoner. There, the condition $y > 50$ evaluates to true and $y \leq 50$ evaluates to false. Fig. 4(d) shows the status after the firing of the predicate transitions for the second `if` statement. The predicate transition of the right branch consumes the token on its input place, but does not produce an output token, since $y \leq 50$ evaluates to false, while the predicate transition of the left branch produces an output token. Thus, only the left branch of second `if` activity is enabled.

7 Conclusions and Outlook

This paper showed how a Petri net based verification of a business process can be enhanced by adding effect places and predicate transitions. We showed how the conditions on the predicate transitions can be evaluated using the knowledge collected during the Petri net analysis. This enables resolving the non-deterministic decisions if the current decision strongly depends on the previously made decisions. Thus the "phantom" paths can be removed from the reachability graph which makes the analysis more effective and precise.

The presented approach can also be used to analyze compositions of business processes, called choreographies. In this case, the knowledge base is shared by all processes and thus each process is aware of the constraints on the input data.

Our future work is to investigate the impacts of our work on current Petri net reduction techniques. We are going to integrate the presented approach in LoLA [11] to prove the applicability of the approach.

References

1. Monakova, G., et al.: Verifying Business Rules Using an SMT Solver for BPEL Processes. In: BPSC. (2009)
2. Leymann, F., Roller, D.: Production Workflow – Concepts and Techniques. Prentice Hall PTR (2000)
3. Moser, S., et al.: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis, IEEE Computer Society (2007) 98–105
4. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: WS-FM. (2007)
5. Breugel, F.v., Koshkina, M.: Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf` (2006)
6. OASIS: Web Services Business Process Execution Language Version 2.0. (2007)
7. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: International Workshop on Languages and Compilers for Parallel Computing, Springer (1997)
8. Dutertre, B., de Moura, L.: The YICES SMT Solver (2008) Available at `http://yices.csl.sri.com/`.
9. Beckert, B., et al.: Intelligent Systems and Formal Methods in Software Engineering. IEEE Intelligent Systems **21**(6) (2006) 71–81
10. Nelson, G., D., O.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems **1**(2) (1979) 245–257
11. Schmidt, K.: LoLA: A Low Level Analyser. In: ICATPN. (2000) 465–474

Facilitating Rich Data Manipulation in BPEL using E4X

Tammo van Lessen, Jörg Nitzsche, and Dimka Karastoyanova

Institute of Architecture of Application Systems
University of Stuttgart
Universitaetsstrasse 38, 70569 Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de

Abstract. The Business Process Execution Language (BPEL) uses XML to specify the data used within a process and realizes data flow via (globally) shared variables. Additionally, assign activities can be used to copy (parts of) variables to other variables using techniques like XPath or XSLT. Although BPEL's built-in functionality is sufficient for simple data manipulation tasks, it becomes very cumbersome when dealing with more sophisticated data models, such as arrays. ECMAScript for XML (E4X) extends JavaScript with support for XML-based data manipulation by introducing new XPath-like language features. In this paper we show how E4X can help to significantly ease data manipulation tasks and propose a BPEL extension that allows employing JavaScript/E4X for implementing them. As E4X allows defining custom functions in terms of scripts, reusability with respect to data manipulation is improved. To verify the conceptual framework we present a proof-of-concept implementation based on Apache ODE.

1 Introduction

Business Process Management (BPM) and the workflow technology [1,2] in particular have enjoyed a great success and have a heavy impact on industry and research. The separation of business process logic and implementation of business functions enables programming on a higher, i.e. business process-oriented level [3], and renders the workflows flexible. Currently, the language for executable business processes is the Business Process Execution Language (BPEL) [4] which is standardized by OASIS[1]. BPEL is XML based and is a part of the Web Service standard stack [5]. It uses XML as data model and specifies activity implementations using the Web Service Description Language (WSDL) [6].

Data flow in BPEL is not explicitly specified but can be realized using (globally) shared variables. Assign activities can be used to copy (parts of) variables to other variables using XML data processing techniques. Although arbitrary expression languages (e.g. XPath [7] and XSLT [8]) can be used to specify expressions to select and copy data, specifying the data manipulation is still a cumbersome task. For instance, it is not possible with the conventional use of BPEL to add elements into a node set (i.e. array operations) or to modify a certain value of a filtered node set.

[1] http://www.oasis-open.org/

ECMAScript for XML (E4X) [9,10] extends JavaScript with support for XML-based data manipulation by introducing new XPath-like language features. The resulting language provides convinient access to XML data and intuitive scripting primitives with direct support for e.g. array operations. In this paper we use this extension to improve BPEL with respect to its data manipulation capabilities and therefore we propose an extension to WS-BPEL 2.0 to allow defining variable assignments in terms of JavaScript/E4X expressions. For this we employ the extensibility features of BPEL, in particular, `<extensionActivity>` and `<extensionAssignOperation>`. This approach contributes a significant enhancement to data manipulation.

The paper is structured as follows. Section 2 provides background information about ECMAScript for XML. Subsequently, The BPEL extensions for E4X are presented in Section 3 and are explained by example in Section 4. Section 5 presents a proof-of-concept implementation based on Apache ODE. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 ECMAScript for XML (E4X)

ECMAScript for XML (E4X) is a language extension that adds native support for XML to the ECMAScript family [11] (including JavaScript, ActionScript, JScript etc.). Unlike other programming languages (like Java) that allow accessing XML data either as event stream or in terms of the W3C DOM object model, E4X allows processing of XML data directly on the language level. The XML tree can be navigated using an object-like "dot" notation and allows for addressing XML child elements, attributes and node sets. Node sets can be filtered using parentheses. The example in Listing 1 illustrates how an E4X object is created and how subsequently the values of the quantity attribute for all items with the name "SOA book" are retrieved. The example also shows how E4X can be used in *for* loops to sum up the prices of all items in the example shopping cart.

```
var items = <items>
    <item name="SOA book" price="40" quantity="2"/>
    <item name="BPM book" price="35" quantity="3"/>
    <item name="EAI book" price="30" quantity="1"/>
  </items>;

alert( items.item.(@name == "SOA book").@quantity );

for each( var thisPrice in items..@price ) {
  sum += thisPrice;
}
```

Listing 1. E4X sample code

E4X allows assigning data values not only to single XML nodes but also to node sets. This enables batch-like modifications of multiple nodes with a single assignment expression (see [12] for further details about E4X).

3 E4X Extension for BPEL

BPEL 2.0 introduces effective extensibility mechanisms that allow for defining new activity types (extension activities) as well as using different mechanisms for data manipulation (extension assign operations). Since E4X provides powerful language extensions to directly address and modify XML data, it makes a good candidate for significantly improving BPEL's data manipulation capabilities. The E4X extension for BPEL is defined in terms of an extension namespace and an extension element for both `<extensionAssignOperation>` and `<extensionActivity>` elements. The extension namespace[2] must be declared as a `mustUnderstand` extension in the preamble of the BPEL process model to ensure that BPEL engines can understand and execute E4X expressions. Subsequently, the `<js:snippet>` element can be used within assign and extension activities respectively and can contain arbitrary JavaScript code.

The E4X extension for BPEL comprises two main parts. First, it makes sure that all visible BPEL variables are injected into the JavaScript context so that they can be treated as normal E4X variables within the JavaScript code snippet. Second, it defines a number of functions that are necessary to glue both worlds together. These functions are listed in Table 1.

`load(string...)`	Allows importing reusable JavaScript libraries. That way, code snippets can be reused across JS/E4X extended activities.
`print(string...)`	Allows printing debug messages to the underlying engine's logging console.
`validate(BPELvariable)`	Makes sure that the given XML object complies with the variable declaration.
`throwFault(...)`	Creates a BPEL fault with a given QName and fault message.
`processName()`	Returns the name of the process model that is currently being executed.
`activityName()`	Returns the name of the activity that executes the JavaScript snipped that is currently being executed.
`piid()`	Returns the name of the activity that executes the JavaScript snipped that is currently being executed.

Table 1. E4X/BPEL built-in function list

[2] `http://ode.apache.org/extensions/js`

4 Example

As identified in Section 1 the most burdensome tasks are the (recurring) initialisation of variables and dealing with arrays (which always requires the use of external XSL scripts). In Listing 2 we demonstrate how E4X extension assign operations are utilized in BPEL. The first operation makes use of the string concatenation operator += and realises a typical *Hello World!* example. The second operation addresses the problems mentioned above. First, it loads an external, reusable JavaScript library, which contains helper methods to create and manipulate XML structures for our shopping cart example. Instead of manually assigning an XML skeleton to a BPEL variable and setting several values later on using XPath expressions, we can (re)use a shared method to create an empty shopping cart. In the last line we use a different JavaScript method to transform the values of the BPEL variable `item`, which was received from an external service, into the XML format prescribed by the shopping cart structure. Subsequently it is added to the virtual shopping cart (+= is the add operator on a node set).

```
<assign name="e4x-assign">
    <extensionAssignOperation>
        <js:snippet xmlns:js="http://ode.apache.org/extensions/js">
            myVar.TestPart += ' World';
        </js:snippet>
    </extensionAssignOperation>
    <extensionAssignOperation>
        <js:snippet xmlns:js="http://ode.apache.org/extensions/js">
            load('shoppingCartUtils.js');
            shoppingCart.parameters = createShoppingCartSkeleton();
            shoppingCart.parameters.items += createCartItem(item);
        </js:snippet>
    </extensionAssignOperation>
</assign>
```

Listing 2. JavaScript/E4X as extension assign operation implementation

Listing 3 demonstrates how to use JavaScript/E4X as extension activity implementation. We assume that the shopping cart has been transformed into a purchase order structure. Depending on the customer type (*gold*, *silver*, *besteffort*), we want to apply different discount ratios. After checking whether the ratios are within a reasonable range, the selected ratio is applied to all items, again by assigning values to a node set. In addition we set the shipping mode to a non-priority mode for best-effort customers.

5 Implementation

The concepts proposed above have been implemented as an extension to Apache ODE[3] and will be part of the upcoming ODE 2.0 release. It was originally intended to be a proof-of-concept implementation for the also newly introduced implementation of BPEL's

[3] `http://ode.apache.org`

```
<extensionActivity name="calculateDiscount">
  <js:snippet xmlns:js="http://ode.apache.org/extensions/js">
    if (goldRatio > 1.0 || silverRatio > 1.0) {
      throwFault('urn:myprocess', 'IllegalArgumentFault',
          'discount ratios must be <= 1.0');
    }
    if (customer.type == 'gold') {
      po.items.item.price *= goldRatio;
    } else if (customer.type == 'silver') {
      po.items.item.price *= silverRatio;
    } else if (customer.type == 'besteffort') {
      po.shippingMode = 'snailmail'
    }
  </js:snippet>
</extensionActivity>
```

Listing 3. JavaScript/E4X as extension activity implementation

extensibility mechanisms in ODE. It provides an *extension operation* implementation which integrates Apache ODE with Mozilla's Rhino[4] as the underlying JavaScript/E4X engine. Since the internal XML model of Rhino is not compatible with W3C's DOM model used by Apache ODE, it was necessary to implement a variable bridge that facilitates overlaying BPEL variables in the JavaScript context. This has been implemented in terms of a Rhino `Delegator`. The built-in functions are realized by overriding Rhino's `ImporterTopLevel`. The source code is available in Apache's Subversion repository[5].

6 Related Work

BPELJ [13] is an extension to BPEL that enables the use of Java code within BPEL activities. While the focus of BPELJ and the approach presented here is similar, BPELJ does not comply with the extensibility features of BPEL 2.0 yet. Furthermore, selecting an XML node in a DOM representation is still cumbersome since Java does not enable selecting XML nodes directly. Hence E4X for BPEL becomes a valuable extension that addresses this deficiency.

[14] proposes an extension to BPEL enabling data manipulation based on ontological knowledge. Using the semantics of data used in a process allows abstracting away the actual implementation of the data manipulation task. It is sufficient to describe which ontology concepts will be provided as input and what is expected as output to discover appropriate data mediators using ontology reasoning. Abstracting away the actual implementation of the data manipulation in processes completely frees process modellers from defining data transformation in a process model and increases reusability of data manipulation tasks. The downside of this approach is that performance decreases

[4] `http://www.mozilla.org/rhino/`
[5] `http://svn.eu.apache.org/repos/asf/ode/trunk/extensions/e4x`

because appropriate transformation/mediation implementations have to be discovered every time data has to be copied from one variable to another. Using the custom E4X functions however, enables defining highly performant and reusable data manipulation functionality.

7 Conclusion

Data manipulation in BPEL is based on XML data processing which makes it a cumbersome task. In this paper we have proposed a BPEL extension that allows employing Javascript/E4X for data manipulation tasks in BPEL and we have shown how E4X can help to significantly ease their implementation. Moreover, reusability with respect to data manipulation is improved as E4X allows defining custom functions in terms of scripts. To verify the conceptual framework we have presented a proof-of-concept implementation based on Apache ODE.

Acknowledgements

The work published in this article was partially funded by the SUPER project[6] under the EU 6^{th} Framework Programme Information Society Technologies Objective (contract no. FP6-026850).

References

1. Leymann, F., Roller, D.: Production workflow. Prentice Hall (2000)
2. van der Aalst, W., van Hee, K.: Workflow management. MIT Press (2002)
3. Leymann, F., Roller, D.: Workflow-based applications. IBM Systems Journal **36**(1) (1997) 102–123
4. A. Alves et al.: Web Services Business Process Execution Language Version 2.0. Committee specification, OASIS (January 2007)
5. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR Upper Saddle River, NJ, USA (2005)
6. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001)
7. Clark, J., DeRose, S.J.: XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116 (November 1999)
8. Adler, S., Berglund, A., Caruso, J., Deach, S., Grosso, P., Gutentag, E., Milowski, R.A., Parnell, S., Richman, J., Zilles, S.: Extensible Stylesheet Language (XSL) Version 1.0. World Wide Web Consortium, Recommendation REC-xsl-20011015 (October 2001)
9. International Organization for Standardization: Information Technology — ECMAScript for XML (E4X) Specification. ISO/IEC 22537:2006 (February 2006)
10. Ecma International: ECMAScript for XML (E4X) Specification. Standard ECMA-357 (June 2004)

[6] `http://www.ip-super.org/`

11. Ecma International: ECMAScript Language Specification. Standard ECMA-262 (December 1999)
12. Tynjala, J.: E4X: Beginner to Advanced. http://developer.yahoo.com/flash/articles/e4x-beginner-to-advanced.html
13. Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., Rowley, M.: BPELJ; BPEL for Java. Joint white paper by BEA and IBM (March 2004)
14. Nitzsche, J., Norton, B.: Ontology based data mediation in BPEL(4SWS). In: Proceedings of the Workshop on Semantics for Web Services (semantics4WS 2008), Milano, Italy (September 2008)

A Method for Partitioning BPEL Processes for Decentralized Execution*

Daniel Wutke, Daniel Martin, and Frank Leymann

Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstrasse 38, 70569 Stuttgart, Germany
{wutke,martin,leymann}@iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de

Abstract Service orchestrations are a common means to compose individual services to either higher-level services or potentially complex composite applications. The Web Service Business Process Execution Language (WS-BPEL) is an example for a language that allows for defining automatically executable orchestrations of Web services. As of today, BPEL process are typically executed in a centralized manner; the process model is deployed on a single workflow management system which, during process instance execution, interprets the process definition and interacts with the orchestrated Web services on behalf of the user. In previous work, we have presented an approach which enables decentralized execution of BPEL processes based on a decentralized process model and supporting runtime infrastructure. In this paper we describe a method for automatic splitting of a process among the partners participating in its execution, referred to as *process partitioning*.

Key words: Process partitioning, decentralized process enactment, BPEL

1 Introduction

One of the key aspects of the *Service-oriented Architecture (SOA)* are service compositions following the so-called *two-level programming* paradigm where individual reusable services are composed into high-level services or potentially complex service orchestrations which can executed automatically using workflow management systems (WfMS). The means for defining the "wiring" between the compound services is provided by workflow definition languages such as the *Web Service Business Process Execution Language (WS-BPEL)* [1]. As of today, the automatic execution of a BPEL process typically comprises the deployment of the process model to a single WfMS and – after process instantiation triggered by incoming messages sent by clients to the WfMS – continuous evaluation of (i) the process' control flow defined in the process model and (ii) the current state of

* This work is supported by the EU funded project *TripCom* (FP6-027324), http://www.tripcom.org.

the process' instance data by the WfMS's *navigator* component; hence we refer to process navigation being a *centralized* process.

However, a number of reasons, ranging from outsourcing of process fragments to runtime performance optimizations without the need for process model changes, motivate the need for a decentralized execution environment for BPEL processes. Hence, in previous work [2,3,4], we have presented an approach that allows for (nearly) arbitrary process splitting by enabling distributed navigation among the partners participating in a process' execution. Following this approach, once can realized different deployment of the same process model within the spectrum from ranging from centralized execution to fully decentralized execution where each "step" (i.e. activity) being carried out by the process is executed by a different process participant. Finding an "adequate" distribution, also referred to as *partitioning*, of the process is dependent on a number of influential factors. Subject of this paper is the presentation and discussion of these factors and a high-level description of an approach to the process partitioning problem that addresses each of the identified influential factors.

The remainder of the paper is structured as follows. In Section 2 the EWFN process model provides the basis for the proposed approach is described to the extend necessary for the discussion of the partitioning procedure. Based on this foundation, Section 3 introduces the general idea of process partitioning, discusses various parameters that influence process partitioning and outlines a procedure that addresses each of the defined partitioning parameters. In Section 4 a brief overview of a few related approaches which are either particularly relevant to the problem discussed in the paper due to either addressing the concrete problem of partitioning BPEL processes or similar parameters influencing process partitioning are presented.

2 Decentralized Enactment of BPEL Processes

Coordinating a number of distributed clients, where each of those clients realizes a defined part of an overall process requires communication of both *process control flow* and *process instance data* among the clients participating in the process' execution. In this context, control flow refers to the individual client's execution being started according to the order defined in the process model; the term instance data characterizes both data being "visible" in process models such as BPEL variables, partner links (which can be source or destination of assignment operations) or correlation sets as well as "invisible" instance data such as the state of a scope or the state of incoming message activities. While this information is provided in the WS-BPEL specification through a description of the language's operational semantics, this description is – due to its informal textual nature – neither suitable for automatic execution by WfMSs nor may it serve as input for process partitioning.

As a result, the formalism of *Executable Workflow Networks (EWFN)* has been developed on the basis of colored, non-hierarchical Petri nets [5] and Boolean networks [6]; it allows for explicitly describing the data communicated

during process instance execution using the communication primitives of the the Linda Tuplespace model [7] – *read* for non-destructive and *take* for destructive consumption of data, *write* for production of data – plus a number of extensions that address process execution-specific requirements such as the *sync* operation for synchronizing join operations [8].

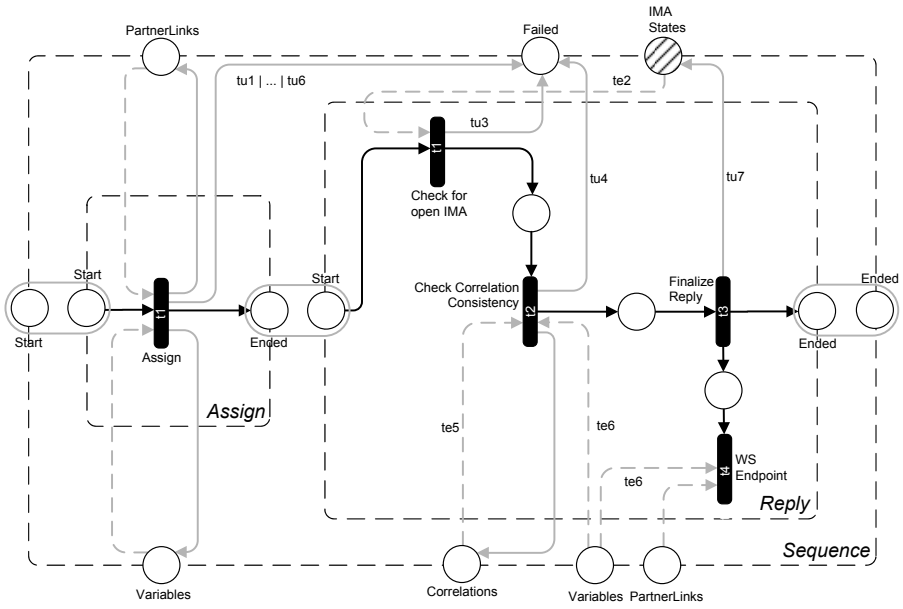


Figure 1. Example of the EWFN representation of a BPEL SEQUENCE activity with two contained `assign` and `reply` activities.

Figure 1 depicts the EWFN representation of a BPEL `sequence` activity with two contained activities. Activities are depicted as dashed rectangles and comprise (similar to Petri nets) transitions and places. Following the EWFN formalism, transitions represent a piece of application logic and carry out the actual processing. Transitions coordinate themselves by consuming tokens from and producing tokens to places which provide passive buffers for tokens; the tokens are self-contained in the sense that they provide enough information to uniquely identify each token communicated in an EWFN. On the level of the infrastructure supporting execution of EWFNs, places are realized by tuplespaces, transitions by tuplespace clients. The arcs between transitions and places represent the individual operations supported by the tuplespace interface and may be annotated with a weight representing the operation cost, resulting e.g. from the volume of the data being communicated as part of that operation.

As an example, transition $t1$ of the `assign` activity represents an assignment of a value to either a BPEL variable or a partner link. To represent process control flow $t1$ is activated by a token becoming available in its *Start* place; once $t1$ has finished its execution is signals its successful completion to the

subsequent activity by producing a corresponding token to its *Ended* activity. Consuming and producing process control flow information is depicted as directed black arcs between places and transitions and transitions and places respectively. During execution of their application logic, transitions may consume and produce further tokens representing instance data; in case of the depicted `assign` activity this might include the modification of the value of a variable or the endpoint reference assigned to a partner link. Similar to BPEL activities, EWFNs can be nested as presented in the example with the `sequence` activity surrounding the `assign` and `receive` activities. The operational semantics of the `sequence` activity is defined as each contained activity becoming ready to execute once its preceding activity has completed its execution. In the EWFN this is represented by collapsing the *Ended* place of the `assign` activity and the *Start* place of the `receive` activity into one place.

3 Process Partitioning

The term *process partitioning* refers to the procedure of assigning information about the partner the corresponding node of the EWFN is executed by during instance runtime to each transition and place of an EWFN. This process is influenced by a number of parameters which can be classified in three groups.

Process model As outlined before, the EWFN of a BPEL process provides a formal description of (i) the steps carried out during process instance execution and (ii) the data being communicated along the way. As a result it is one of the major parameters of the process partitioning procedure.

Service infrastructure Through BPEL's interaction activities (e.g. `invoke` and RECEIVE) the BPEL process may interact with Web service clients and the Web services it orchestrates. As the partners providing a service used by the process are in any event process participants they are potentially suitable candidates for executing a part of the processes orchestration logic as well.

Organizational factors Organizational factors reflect parameters that are not necessarily a result of the process structure or its service landscape, but are defined manually by users. It might e.g. be desired to manually define the partition of a certain place that contains BPEL variable data for data ownership reasons.

The proposed process partitioning approach comprises three phases and is an extension of the procedure presented in [9] in the sense that it also relies on the notion of different kinds of nodes – *fixed*, *heavy*, and *light* – whose partition assignment is addressed in consecutive phases of the partitioning algorithm as depicted in Table 1; once a partition of a been determined in on of the phases the node is not considered in the further phases of the algorithm.

Fixed nodes represent nodes with partitioning information defined a priori by manual user input and reflect organizational partitioning parameters. To allow for maximum flexibility, each node of an EWFN – transitions and places – can become a fixed node. Interaction activities, i.e. those points of a BPEL process

Phase	1. Fixed Nodes	2. Heavy Nodes	3. Light Nodes
Examined Objects	Arbitrary nodes	Interaction activities: `invoke`, `receive`, `pick`, `reply`	Non-interaction activities; instance data
Partitioning Method	Manual assignment by user; rules	Automatic *service discovery* and *service selection*; rules	Graph partitioning applied to the process' EWFN

Table 1. Phases of the proposed method to BPEL process partitioning.

where interaction with its service landscape occurs, are referred to as *heavy nodes*. Their partitioning information is determined automatically using means for *service discovery* (based on the service's functional characteristics through its WSDL description) and *service selection* (based on the service's non-functional properties such as service invocation cost, service response time, etc. reflected through WS-Policy descriptions). In addition, partitioning information of *heavy nodes* might be dependent on deployment information (e.g. for defining on which endpoint the process can receive incoming messages) or other *heavy nodes* (e.g. in case of `receive-reply` pairs to support synchronous Web service bindings). *Light nodes* reflect non-interaction activities as well as process instance data for which no partitioning information has been defined until this point. Their partitioning is performed by migrating them to the partition of adjacent *fixed* or *heavy nodes* as defined in the process' EWFN. Since an EWFN is a directed weighted graph and the problem is a variant of the well-known *graph partitioning* problem with the optimization criteria of minimizing the cost of inter-partition interactions, existing optimization algorithms such as *Simulated Annealing* [10] are used to realize this phase of the partitioning algorithm.

4 Related Work

In [11], the authors introduce a workflow system architecture for supporting large-scale distributed applications called *Mentor* based on TP monitors and object request brokers. Decentralized workflow execution in Mentor is achieved by rule-based partitioning of a workflow based on activity and state charts into a set of sub-workflows which are then enacted by a number of distributed workflow engines that are synchronized using Mentor. In [12], a BPEL process model is manually split by a user (e.g. for reasons of process outsourcing) in a number of fragments and a corresponding BPEL process (along with necessary deployment information) is created for each fragment. The BPEL processes are then deployed and executed at the partners participating in the process' execution. For supporting BPEL's `scope` and `while` activities, a central coordinator is required. In [9] a similar approach to distributed execution of BPEL processes is presented supporting automatic process partitioning based on an analysis of a program dependence graph generated for the process and a corresponding cost model.

5 Conclusions

In this paper we have outlined a process model that enables decentralized execution of BPEL processes and allows for nearly arbitrarily fragmented process execution. Thereby we have stressed the need for an algorithm for defining process partitions based on a number of influential factors and have presented a high-level overview of the proposed algorithm and how it addresses the individual process partitioning parameters.

References

1. Organization for the Advancement of Structured Information Standards: Web Services Business Process Execution Language Version 2.0 – OASIS Standard (March 2007)
2. Wutke, D., Martin, D., Leymann, F.: Model and Infrastructure for Decentralized Workflow Enactment. In: SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing, New York, NY, USA, ACM (2008) 90–94
3. Daniel Martin and Daniel Wutke and Frank Leymann: EWFN – a Petri net dialect for tuplespace-based workflow enactment. Volume 380 of CEUR Workshop Proceedings., CEUR-WS.org (September 2008) 7–14
4. Martin, D., Wutke, D., Leymann, F.: A novel approach to decentralized workflow enactment. Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE (Sept. 2008) 127–136
5. Jensen, K.: Coloured Petri Nets, Vol. 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag (1992)
6. Langner, P., Schneider, C., Wehler, J.: Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen. Wirtschaftsinformatik **39**(5) (1997) 479–489
7. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
8. Martin, D., Wutke, D., Leymann, F.: Synchronizing control flow in a tuplespace-based, distributed workflow management system. In: ICEC '08: Proceedings of the 10th international conference on Electronic commerce, New York, NY, USA, ACM (2008) 1–9
9. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing Execution of Composite Web Services. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems,Languages, and Applications (2004) 170–187
10. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by Simulated Annealing. Science **220**(4598) (1983) 671–680
11. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A., Weikum, G.: From Centralized Workflow Specification to Distributed Workflow Execution. Journal of Intelligent Information Systems **10**(2) (1998) 159–184
12. Khalaf, R., Leymann, F.: Role-based decomposition of business processes using bpel. In: ICWS '06: Proceedings of the IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2006) 770–780

Autorenverzeichnis