Fixing Deadlocking Service Choreographies Using a Simulation-based Graph Edit Distance

Niels Lohmann

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

Abstract. Many work has been conducted to analyze services and service choreographies to assert manyfold correctness criteria. While errors can be *detected* automatically, the *correction* of defective services is usually done manually. This paper introduces a graph-based approach to calculate the minimal edit distance between a given defective service and synthesized correct services. This edit distance helps to automatically fix found errors while keeping the rest of the service untouched.

1 Introduction

In service-oriented computing [1], the correct interplay of distributed services is crucial to achieve a common goal. Choreographies [2] are a means to document and model the complex global interactions between services of different partners. BPEL4Chor [3] has been introduced to use BPEL [4] to describe and execute choreographies. Recently, a formal semantics for BPEL4Chor was introduced [5], offering tools and techniques to verify BPEL-based choreographies. Whereas it is already possible to automatically *check* choreographies for deadlocks or to synthesize participant services [6], no work was conducted in supporting the *fixing* of existing choreographies. This is especially crucial since fixing incorrect services is usually cheaper and takes less time than re-designing and implemeting a correct service from scratch. In addition, information on how to adjust an existing services can help the designer understand the error more easily compared to confronting him with a whole new synthesized service.

As a running example for this paper, consider the example choreography[1] of Fig. 1. It describes the interplay of a travel agency, a customer service, and an airline reservation system. The travel agency sends an offer to the client which either rejects it or books a trip. In the latter case, the travel agency orders a ticket at the airline service which sends either a confirmation or a refusal message to the customer. The customer service, however, does not receive the refusal message which leads to a deadlock in case the airline refuses the ticket order.

This design flaw can be easily corrected by adjusting the customer service. But even for this simple choreography, there is a variety of possibilities to fix the customer's service. Figure 2 depicts two possible corrections. Though both services would avoid the choreography to deadlock, the service in Fig. 2(a) is to be preferred over that in Fig. 2(b) as it is "more similar" to the original service.

[1] To ease the presentation, we use BPMN to visualize services and choreographies.

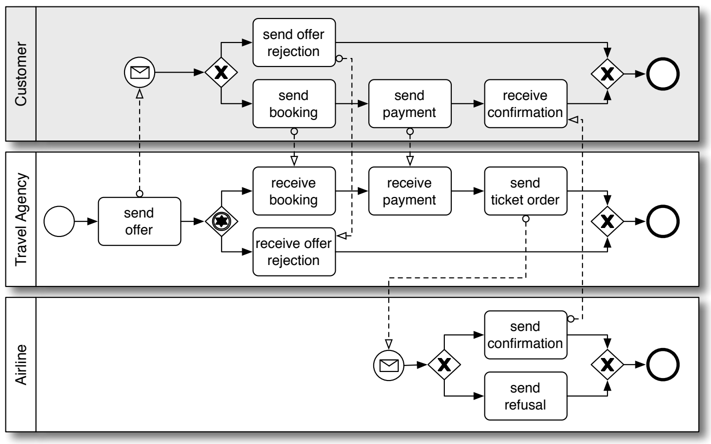
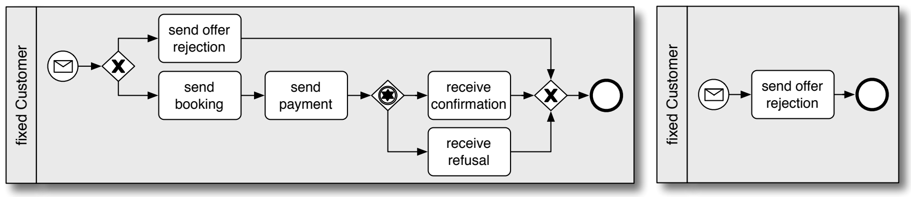


Fig. 1. Choreography between travel agency, airline, and customer. The choreography can deadlock, because the customer does not receive a refusal message from the airline.



(a) add receipt of refusal message (b) delete booking branch

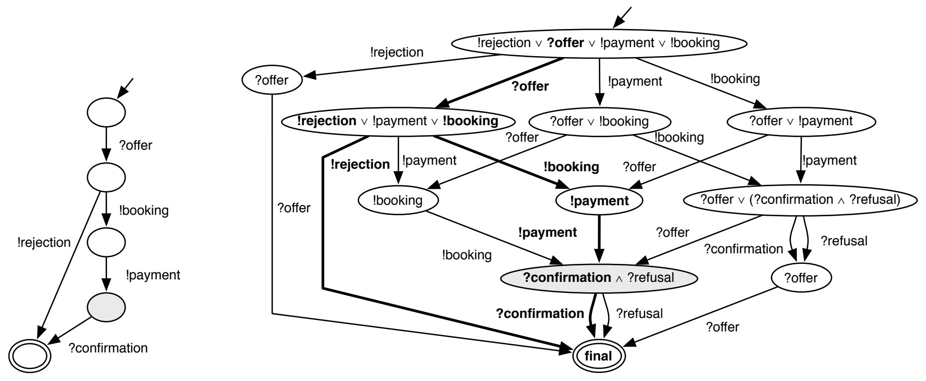
Fig. 2. Two possible corrections of the customer service to avoid deadlocks.

The goal of this paper is to formalize, systematize, and to some extent automatize the fixing of choreographies as above. We thereby combine existing work on characterizing all correctly interacting partners of a service with similarity measures and edit distances known in the field of graph correction. These approaches are recalled in Sect. 2. In Sect. 3, we define an edit distance that aims at finding the most similar service from the set of all fitting services. To support the modeler, we further derive the required edit actions needed to change the originally incorrect service. Finally, Sect. 5 is dedicated to a conclusion and gives directions for future research.

2 Background

2.1 Service Models and Partner Service Characterization

To formally analyze services, a sound mathematical model is needed. In the area of workflows and services, Petri nets are widely accepted formalism [7]. They combine a graphical notation with a variety of analysis methods. For real-life service description languages such as BPEL or BPEL4Chor exists a feature-complete Petri net semantics [8, 5]. To simplify the presentation, we abstract from the structure of a service and complex aspects such as data or fault handling, and focus on the external behavior of services in this paper. To this end, we use automata to model the external behavior services.[2]



(a) automaton (b) operating guideline (OG)

Fig. 3. An automaton modeling the customer service of Fig. 1 (a) and the operating guideline of the composition of travel agency and airline service (b).

Figure 3(a) depicts an automaton modeling the external behavior of the customer service of Fig. 1. The edges are labeled with messages sent to (preceded with “!”) or received from (preceded with “?”) the environment. As services are usually not considered in isolation, their interplay has to be taken into account in verification. An important correctness criterion is *controllability* [9]. A service is controllable if there exists a partner service such that their composition (i. e., the choreography of the service and the partner) is free of deadlocks.

Controllability can be decided constructively: if a partner service exists, it can be automatically synthesized [9, 6]. Furthermore, there exists one canonic partner service that is most permissive; that is, it simulates any other partner service. The converse however, does not hold; not every simulated service is a correct partner service itself. To this end, the most permissive partner service can be annotated with Boolean formulae expressing which states and transitions can be omitted and which parts are mandatory. This annotated most permissive partner service is called an *operating guideline* (OG) [10].

Figure 3(b) depicts the OG of the composition of the travel agency and the airline. The disjunction of the OG’s initial state means that a partner service must send a rejection, receive an offer, send a payment or send a booking in its initial state. Conjunctions in OG’s states represent the fact that some choices cannot be influenced by the partner of the travel agency and the airline. Instead, the customer has to react on any decision of the travel agency and the airline. For example, the formula “?confirmation $\wedge$?refusal” describes the fact that the

[2] Due to the close relationship between Petri nets and automata, there exist techniques to transform back and forth between the two formalisms.

airline decides whether to confirm *or* to refuse the booking and the traveler has to be ready to receive *both* resulting messages.

The automaton of Fig. 3(a) is simulated by the OG and fulfills all but one formula (satisfied literals are bold in Fig. 3(b)). It does not satisfy the formula "?confirmation $\wedge$?refusal" of the OG's state (depicted gray) reached after sending the booking and payment, because the automaton does not receive a refusal message in this state. Beside the corrected services of Fig. 2, the OG characterizes 1,021 additional partner services.

2.2 Graph Similarities and Edit Distances

Graph similarities are widely used in many fields of computer science, for example for pattern recognition or in bio informatics. Cost-based distance measures adapt the *edit distance* known from string comparison [11] to compare labeled graphs (e. g., [12]). They aim at finding the minimal number of modifications (i. e., adding, deleting, and modifying nodes and edges) needed to achieve a graph isomorphism. These distance measures have the drawback that they are solely based on the *structure* of the graphs. Thus, they focus on the syntax of the graphs rather than their semantics. When a graph (e. g., an automaton) models the *behavior* of a system, similarity of graphs should focus on simulation of behavior rather than on a small edit distance (see Fig. 4).

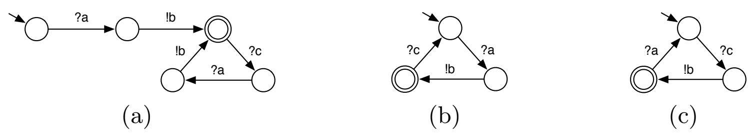


Fig. 4. Automaton (a) and (b) simulate each other, but have a high edit distance. Automaton (b) and (c) have a low edit distance, but rather unsimilar behavior.

In [13], this problem is addressed and motivated by finding computer viruses by comparing a program with a library of control flow graphs. In that setting, classical simulation is too strict because two systems that are equal in all but one edge label *behave* very similar, but there exists no simulation relation between them. To this end, the authors introduce a *weighted weak quantitative simulation* function to compare states of two graphs. Whenever the two graphs cannot perform a transition with same labels, one graph performs a special stuttering step similar to τ-steps in stuttering bisimulation [14]. Stuttering is penalized using a cost function that assigns any pair of labels a cost. Minimizing the costs for stuttering then yields in maximal similarity.

3 Combining Edit Distance and Service Matching

The algorithm to calculate weighted weak quantitative simulation can be used as a similarity measure for automata or OGs, but has two drawbacks: Firstly, it is not an edit distance, as it does not give the modification actions needed to achieve simulation. Secondly, it does not take formulae of the OG into account. Therefore, a high similarity between an automaton and a OG would not guarantee deadlock freedom as the example of Fig. 3 demonstrates.

3.1 Simulation-based Edit Distance

Given a state q_A of an automaton and q_O of an OG, the algorithm of [13] determines the best matching between the transitions of q_A and q_O. In addition, the automaton or the OG can stutter (i. e., remain in the same state). From these transition pairs that are used to calculated the similarity measure, we can derive the according edit actions (see Table 1).

Table 1. Deriving edit actions from transition pairs of [13].

automaton transition	OG transition	resulting edit action
a	a	keep transition a
a	b	modify transition a to b
a	*stutter*	delete transition a
stutter	a	insert transition a

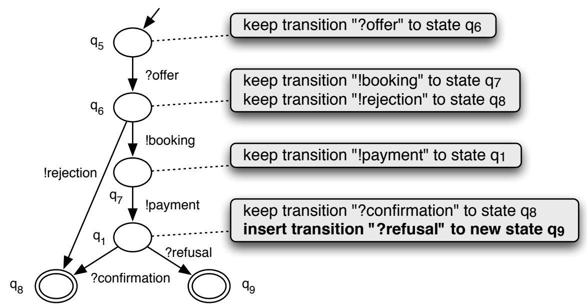
3.2 Adding Formula-checking to the Edit Distance

As described earlier, a partner is represented by the OG if (i) the OG simulates the partner and (ii) the OG's formulae are fulfilled. The simulation-based edit distance does not respect the OG's formulae. Therefore, a partner created by this edit distance is not guaranteed to interact without deadlocks. Experiments showed that fixing these partner services by adding edges to fulfill the OG's annotations yield suboptimal results. To this end, we extend the algorithm of [13] not to statically take all outgoing transitions of an OG's state into account, but also check any formula-fulfilling subset of outgoing transitions. The algorithm now adds all edges necessary to fulfill the OG's formulae while trying to find the best matching assignment.

When comparing the gray states of Fig. 3, the "?confirmation" transitions match, but the OG's "?refusal" transition can only be matched to a stuttering step of the automaton. Thus, "insert transition ?refusal" would be the resulting edit action of the automaton's state. Figure 5 illustrates the result of the algorithm.

4 Complexity Considerations and Experimental Results

The original simulation algorithm of [13] to calculate a simulation between two service automata A_1 and A_2 needs to check $\mathcal{O}(|Q_{A_1}| \cdot |Q_{A_2}|)$ state pairs (Q_{A_i} is the set of states of A_i). The extension to respect the OG's formulae to calculate the edit distance between a service automaton A and an OG O takes the OG's formulae and the resulting label permutations (i. e., how the edges



q5
q6
q7
q1
q8
q9
?offer
!booking
!rejection
!payment
?refusal
?confirmation
keep transition "?offer" to state q6
keep transition "!booking" to state q7
keep transition "!rejection" to state q8
keep transition "!payment" to state q1
keep transition "?confirmation" to state q8
insert transition "?refusal" to new state q9

Fig. 5. Matching-based edit distance applied to the customer's service.

of a satisfying assignment are mapped to the edges of the service automaton) into consideration. The length of the OG's formulae is limited by the maximal degree of the nodes which again is limited by the interface I. Thus, for each state pair, at most $2^{|I|}$ satisfying assignments have to be considered. The number of permutations is again limited by the maximal node degree such that at most $|I|!$ permutations have to be considered for each state pair and assignment. This results in $\mathcal{O}(|Q_A| \cdot |Q_O| \cdot 2^{|I|} \cdot |I|!)$ comparisons.

Though the extension towards a formula-checking edit distance yields a high worst-case complexity, OGs of real-life services tend to have quite simple formulae, a rather small interface (compared to the number of states), and a low node degree. As a proof of concept, we implemented the edit distance in a prototype.[3] It takes an acyclic deterministic service automaton and an acyclic OG as input and calculates the edit actions necessary to achieve a matching with the OG. The prototype exploits the fact that a lot of subproblems overlap, and uses dynamic programming techniques [15] to cache and reuse intermediate results which significantly accelerates the runtime. We evaluated the prototype with models of some real-life services. In most cases, the edit distance could be calculated within few seconds. Table 2 summarizes the results.

The first seven servies are derived from BPEL processes of a German consulting company; the last four services are taken from the current BPEL specification [4]. Column "search space" of Table 2 lists the number of acyclic deterministic services characterized by the OG. All these services are correct partner services and have to be considered when finding the most similar service. The presented algorithm exploits the compact representation of the OG and allows to efficiently find the most similar service from more than 10^{2000} candidates.

For most services, the calculation only takes a few seconds.[4] The "Internal Order" and "Purchase Order" services are exceptions. The OGs of these services have long formulae with a large number of satisfying assignments (about ten times larger than those of the other services) yielding a significantly larger search space. Notwithstanding the larger calculation time, the service fixed by the calculated

[3] Available at `http://service-technology.org/rachel`.

[4] The experiments were conducted on a 2 GHz computer.

Table 2. Experimental results.

service	interface	states SA	states OG	search space	time (s)
Online Shop	16	222	153	10^{2033}	4
Supply Order	7	7	96	10^{733}	1
Customer Service	9	104	59	10^{108}	3
Internal Order	9	14	512	$> 10^{4932}$	195
Credit Preparation	5	63	32	10^{36}	2
Register Request	6	19	24	10^{25}	0
Car Rental	7	49	50	10^{144}	6
Order Process	8	27	44	10^{222}	0
Auction Service	6	13	395	10^{12}	0
Loan Approval	6	15	20	10^{17}	0
Purchase Order	10	137	168	$> 10^{4932}$	391

edit actions is correct by design, the difference between the erroneous and the fixed service is minimal, and the calculation time is surely an improvement compared to iterative manual correction.

5 Conclusion and Future Work

We presented an edit distance to find the necessary edit actions to correct a faulty automaton to interact without deadlock in a choreography. The edit distance (i. e., the actions needed to fix the automaton) can be automatically calculated using a prototypic implementation. Together with translations from and to [8, 16] BPEL processes and the generation of the OG [6, 10], a complete tool chain[5] to analyze and correct BPEL-based choreographies is available.

However, a lot of questions remain open. First of all, the choice *which* service to fix is not always obvious and needs further investigation. For instance, the choreography of Fig. 1 could also have been fixed by adjusting the airline service. In contrast, the service of the travel agency cannot be fixed with the presented approach: the composition of the customer and the airline service is not controllable and thus no OG exists. The diagnosis of uncontrollable services is subject of future work.

Another aspect to be considered in future research is the choice of the *cost functions* used in the algorithm, as it is possible to set different values for any transition pairs. Semantic information on message contents and relationships between messages can be incorporated to refine the correction. For example, the insertion of the receipt of a confirmation message can be penalized less than the insertion of sending an additional payment message. Beside the presented application of fixing choreographies, the edit distance might also be used in the context of service-oriented architectures (SOA): given a service requester as automaton and a service repository consisting of provider services' OGs, the

[5] See `http://service-technology.org/tools`.

edit distance might help to find the smallest adaptation of the requester to find a fitting provider. Finally, heuristics may help to increase the performance of the implementation by omitting suboptimal edit actions. For instance, guidance metrics such as used in the A* algorithm [17] may greatly improve runtime.

References

1. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Commun. ACM **44**(4) (2001) 71–77
2. Dijkman, R., Dumas, M.: Service-oriented design: A multi-viewpoint approach. IJCIS **13**(4) (2004) 337–368
3. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS 2007, IEEE (2007) 296–303
4. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. OASIS Standard, 11 April 2007, OASIS (2007)
5. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: WS-FM 2007. Volume 4937 of LNCS., Springer (2008) 46–60
6. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006. Volume 4102 of LNCS., Springer (2006) 17–32
7. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
8. 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
9. Schmidt, K.: Controllability of open workflow nets. In: EMISA 2005. Volume P-75 of LNI., GI (2005) 236–249
10. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341
11. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Dokl. **10**(8) (1966) 707–710
12. Tsai, W., Fu, K.: Error-correcting isomorphisms of attributed relational graphs for pattern analysis. IEEE Trans. on SMC **9**(12) (1979) 757–768
13. Sokolsky, O., Kannan, S., Lee, I.: Simulation-based graph similarity. In: TACAS 2006. Volume 3920 of LNCS., Springer (2006) 426–440
14. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: FSTTCS 1997. Volume 1346 of LNCS., Springer (1997) 284–296
15. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
16. 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)
17. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths in graphs. IEEE Trans. Syst. Sci. and Cybernetics **SSC-4**(2) (1968) 100–107