Toward Deciding the Existence of Adaptable Services

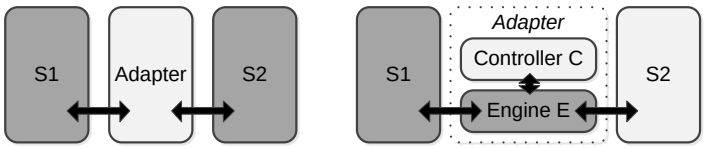
Christian Gierds

Humboldt-Universität zu Berlin, Department of Computer Science,
Unter den Linden 6, 10099 Berlin, gierds@informatik.hu-berlin.de

Abstract. *Service adaptation* allows two services to interact properly using a mediator or adapter. In service discovery one question is whether an *adaptable service* exists for a given service, i. e. whether a service exists which can be interacted with properly by using an adapter.
We look at a setting where this question boils down to deciding *distributed controllability*, and we present an idea for changing an algorithm for controller synthesis which answers this question.

1 Motivation

In recent years the idea of *service adaptation* gained momentum within the scientific community. Services already are a wide-spread paradigm also used in industry. Thus the pool of independently developed services is huge. As services are made to be coupled, the question of *service discovery* (viz. does a service exist that my service can interact with) plays an important role in Service-Oriented Architectures [6]. However, as a service might be developed without knowledge about existence of potential partners, it is likely that service discovery fails because of incompatibilities. In this case an *adapter* [8] might overcome these incompatibilities. As indicated in Fig. 1a, the adapter acts as mediator between two services S_1 and S_2 ensuring *semantically correct processing of messages* and *proper termination* of the services. In case service discovery fails, i. e. there does not exist any service for direct interaction with, *the question concerning service discovery now can be extended to the adapter setting.*



(a) Normal Adapter Setting (b) Discovery of an Adaptable Service

Fig. 1. Two different perspectives (dark-gray means given, light-gray means wanted)

Existence of an Adaptable Service: Is there another service and an adapter, such that my service can communicate correctly with this other service using the adapter?

This question actually is not easy to answer. It appears, that we may simply apply an algorithm for service selection in order to pick S_2 and an adapter. Service adaptation proposes to *create*, not to select a mediating service though. As it mainly works on a semantical rather than a functional level adapters are not meant to be stored and be publicly available. Thus the question above would suggest trying to create an adapter for each candidate S_2.

Setting. Many newer approaches [1–4] recommend to first describe the semantical constraints of an adapter and then to ensure also behavioral correctness (viz. proper termination by coordinated transformation of messages). Figure 1b shows one possible solution [4]: Let the services S_1 and S_2 be given, as well as a set of *message transformation rules* describing valid transformations of messages. These transformations are implemented in an *engine* service E ensuring the semantically correct exchange of messages. If we find a controller C triggering transformations as they are actually needed, then the composition of $E \oplus C$ is an adapter.

For this approach we may assume to have S_1 and at least the transformation rules and thus the engine E given, when looking for a service S_2. Checking the *Existence of an Adaptable Service* then asks for a service S_2 for which a controller C exists, such that the composition $S_1 \oplus E \oplus C \oplus S_2$ properly terminates.

Contribution. We provide an algorithmic idea to abstract from C and then ask for the existence of an S_2 using existing work on partner synthesis. As to the best of our knowledge the question for the *Existence of an Adaptable Service* has not been answered so far. The idea proposed in this paper is a first step in solving the problem.

The ultimate goal is the *characterization of all adaptable services*. Then, given a candidate in a repository, we could decide, whether an adapter can be computed for this candidate or not. However, this problem will remain for future work. So far, we simply want to be able to check, whether it is possible to find such a candidate, or if no such service exists as the transformation rules are not sufficient for adaptation.

In the following we first introduce service adaptation on a formal level (Sec. 2). Then we briefly discuss the problem of distributed controllability (Sec. 3)—our main question relates to this problem—before giving an idea for solving the problem in the special case of adapters (Sec. 4). Finally we give an outlook (Sec. 5) on how to extend the solution to partially solve the more general case of *distributed controllability*.

2 Service Adaptation

In the last couple of years many approaches [1–3] (among others) emerged for the adaptation of independently developed services. Many of these approaches

describe the semantical level of an adapter independently of its control structure. The semantical level is described by *message transformation rules* defining the transformation of a message in order to meet semantical constraints imposed by its content. Typical transformations range from simple renaming to the combination of several message into one message (or vice-versa), or the creation/deletion of protocol message (e. g., acknowledgments).

We use previous work [4] by colleagues and myself based on Petri nets to formally describe the setting (as shown in Fig. 1b). Using Petri nets gives some advantages: they allow to easily describe distributed models, and message transformation rules can be directly translated into a net structure.[1] We use the typical definition of a *Petri net* $N = (P, T, F, m_0)$ with finite sets of places P, transitions T, a flow relation $F \subseteq (P \times T) \cup (T \times P)$, and an initial marking m_0. A marking $m : P \to \mathbb{N}$ assigns to every place a number of tokens. The firing semantics are as usual: a transition $t \in T$ is enabled in a marking m, when all places in the preset are marked appropriatly ($(p,t) \in F \Rightarrow m(p) > 0$), and t may fire only, if it is enabled and thus changes the marking to $m'(p) = m(p) - F(p,t) + F(t,p)\ \forall p \in P$.

An *open net* additionally uses transition labels to express communication via some channel c: a transition may asynchronously send a message ($!c$), receive a message ($?c$)—thus restricting firing if c contains no message—, or it may synchronize ($\#c$). Further we provide a set Ω of *final markings*, indicating in which state a service is allowed to terminate.

The approach for adapter synthesis then can be summarized as follows: let us assume to have given service models S_1 and S_2 (as open nets) as well as message transformation rules, which can be canonically translated into an open net E (the *engine*). Each transition of E is synchronously connected to a controller port, thus allowing full control about the application of transformation rules. We then use existing algorithms for controller synthesis for constructing a controller and thus an adapter, if any exists. The controller's role is to ensure proper termination as transformation rules may not be applied arbitrarily. In the following *proper termination* is used synonymously to *weak termination* (viz. it is always possible to reach a final state).

Figure 2 shows our (technical) running example. We use the typical graphical notation for Petri nets. Additionally the dashed line shows the boundary of one open net, places indicating a proper final state are depicted using a double line. Communication labels are written inside a transition (omitting the synchronous labels used for communication in the adapter between the lower engine and the upper controller part). Service S_1 (Fig. 2a) receives an initial message ($?\mathsf{e}$), waits for an external choice (either $?\mathsf{b_1}$ or $?\mathsf{b_2}$), sends an appropriate answer (either $!\mathsf{t}$ or $!\mathsf{c}$), and returns to its initial state. Service S_2 (Fig. 2c) simply sends a message ($!\mathsf{m}$) and waits for a reply ($?\mathsf{k}$); or it may decide to terminate. Within the engine part of the adapter (Fig. 2b) we see the communication with S_1 and S_2 as well as the transformation rules ($\mathsf{r_1}$ to $\mathsf{r_5}$). In more detail the rules are: $r_1 : m \mapsto e$

[1] N. B.: The whole theory could be canonically translated into state machines. However we decided to use Petri nets.

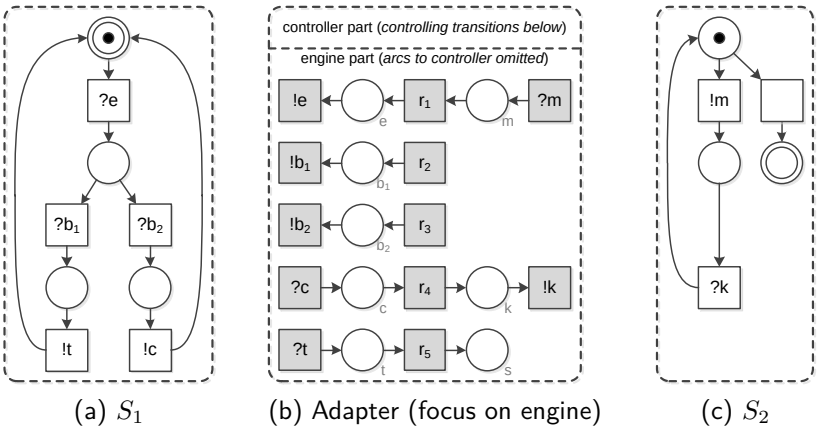


Fig. 2. Running example: Adapter for two services S_1 and S_2

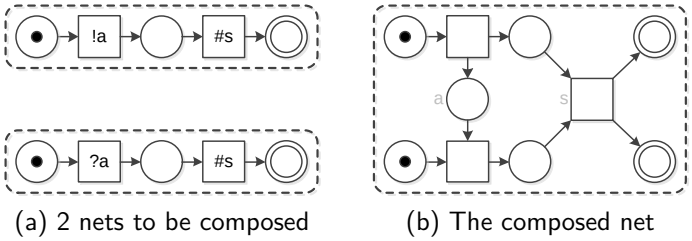


Fig. 3. Composition of open nets

(*rename m to e*); $r_2 : \mapsto b_1$ (*create* b_1); $r_3 : \mapsto b_2$ (*create* b_2); $r_4 : c \mapsto k$ (*rename c to k*); and $r_5 : t \mapsto s$ (*rename t to s*). As we can see, rules canonically translate into Petri net transitions, where the channel names translate into places. *Please note*, that there are additional arcs between the engine and controller part that we omitted for sake of simplicity.

Composition of two open nets A and B is realized by introducing a buffer place p_c for each asynchronous channel c and adding an arc (t, p_c) for each transition t with label $!c$, an arc (p_c, t) for each transition t with label $?c$, and each pair of transitions with the same synchronous label $\#c$ is merged while preserving their individual presets and postsets (Figure 3 shows an example).

We now want to change the perspective in order to check the *Existence of an Adaptable Service*: let us assume we only know about service S_1 and some transformation rules. *Does any service* S_2 *exist, such that* S_1 *and* S_2 *are adaptable given the transformation rules?* Regarding Fig. 1a we have given services S_1 and E and are looking for services S_2 and C (where the latter is of minor importance). Nevertheless in our setting we are actually looking for *two* services that are not

allowed to communicate with each other directly—as S_2 and C only communicate with the engine E—but that we can match during construction.

Let us rephrase the problem a bit: *given $S_1 \oplus E$, we are looking for a service S_2, such that we can be sure, an appropriate C also exists.*

3 The Problem of Distributed Controllability

The problem we want to solve in the adapter setting—checking for the *Existence of an Adaptable Service*—relates to the problem of *distributed controllability* [7]. Given a service A ($S_1 \oplus E$) with two distinguished ports for communication, do two services B_1 (service S_2) and B_2 (controller C) exist, such that the composition of $B_1 \oplus A \oplus B_2$ describes a proper system. In this setting, B_1 communicates with A over one port, and B_2 over the other. However, B_1 and B_2 are not allowed to directly communicate with each other.

The described problem is known to be solvable for acyclic services [7]; however, for cyclic services there exists strong suspicion, that the problem is in fact undecidable.[2]

Although checking the *Existence of an Adaptable Service* thus relates to a problem suspected to be undecidable, we present an idea for tackling this problem in the special case of adapters. This is possible as the engine of an adapter has a very special structure—every transitions within the engine is under control. The decisions of any controller are very determined, which helps us in predicting a controllers decisions. Thus even in case of cyclic services we can actually decide, whether an adaptable service exists.

In the next section we exploit this fact by actually foretelling some of the controllers decisions and then checking for the *Existence of an Adaptable Service* using existing algorithms for partner synthesis.

4 Existence of Adaptable Services

In this section we shortly sketch the idea for answering the question, whether a service S_2 exists for a given service S_1 and a set of transformation rules, such that S_1 and S_2 are adaptable with respect to the transformation rules.

First of all we have to fix an interface for S_2. Otherwise it is not clear, which messages used within the transformation rules are actually meant to be sent or received—which actually is not always clear from the rules. However, in many cases the rules suggest a certain direction and we leave it to future work to find heuristics for allowing more flexibility in choosing the interface.

When trying to find S_2 we have to take care of certain points: first, building a central controller (one serving both ports) can be misleading. There exists S_1 and E which have a central controller, but no distributed one (e. g., when S_2 would need to react on messages exchanged between E and C). An algorithm working on the central controller must be aware of this. Second, as we are

[2] Personal communication with Karsten Wolf. Unpublished result.

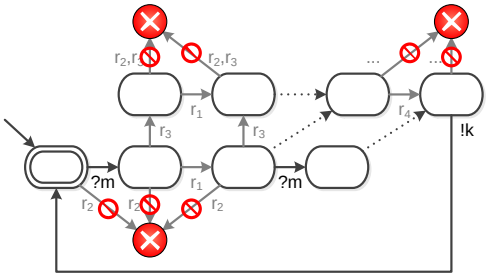


Fig. 4. Automaton derived from $S_1 \oplus E$ with branches leading to deadlocks removed

mainly interested in S_2 we could abstract from the interface (i. e. remove all communication) between engine and controller. This would respect somehow the independence of S_2 and controller C. However, this does not take into account the possibility for design-time coordination of S_2 and C, thus failing in finding any S_2 in many cases, when actually one exists.

The approach we want to follow is a mixture of both ideas: abstract from the communication between engine and controller, but use information about transitions being part of the engine to decide, whether bad states could be avoided by an yet unknown controller.

The algorithm [5] for constructing a controller in the adapter setting presented above is based on communicating automata which can be canonically derived from the reachability graph of $S_1 \oplus E$. In our setting there might be many states, where avoiding a deadlock or livelock seems impossible for S_2 alone. For a transition of the engine a controller can however decide *not* to fire it if it leads unavoidably to bad states.

Our algorithm for finding an adaptable service S_2 (without generating a corresponding controller part C) can be summarized as follows: Let us have given a service S_1 and an engine E representing message transformation rules. Translate the reachability graph of $S_1 \oplus E$ into a communicating automaton. If there is a transition with a controller label—a label for communication between engine E and controller C—which results in a state, where reaching a bad state is unavoidable for any S_2, then remove this transition and all states becoming unreachable. If no such further transitions exist, remove all controller labels (making corresponding transitions communicating with the controller part of an adapter internal) and run the algorithm for partner synthesis.

This way we exploit the possibility to rely on a correct decision of C in case it is necessary. As we are only removing transitions where reaching a final state cannot be enforced anymore—neither by S_2 or any controller C—communication between engine and controller is not determined and a level of uncertainty remains, which S_2 has to cope with (viz. S_2 is still not aware of communication between E and C).

Running example: Let us consider the example in Fig. 2. The transition r_3 (creating b_2) should fire once for every m received, but never a second time. As we know that r_3 is part of the engine, we know that a C can decide to fire r_3 one time (as a final state is reachable in the example), but never a second time, when no further m was received (as a final state might not be reachable anymore in the example). Thus we remove all transitions related to the second firing of r_3. We can see the (partial) result of this operation in Fig. 4. Arcs and labels related to engine transitions are gray, an unavoidable deadlock is indicated by a cross, the removal of arcs by prohibition signs. When we start partner synthesis on the artifact depicted in Fig. 4, then we get as result a service corresponding to the net initially depicted in Fig. 2c. Thus we are able to compute a witness to show that S_1 is adaptable provided the given transformation rules.

5 Conclusion and Outlook

The use of adapters extends the setting of Service-Oriented Architectures by some challenging questions. In the case of service discovery we may not only be interested a (compatible) partner service, but also a partner service usable through an adapter would serve our purposes. Thus the question arises if any *adaptable service does exist*. In case we regard an adapter as a union of semantical constraints and control flow we have provided an algorithmic idea to answer this question. We have omitted a proof for the correctness of this approach. Surely it highly relies on the very special structure of the engine (unique communication labeling, controller always has complete knowledge about the state of the engine, thus decisions are determined, etc.).

If we want to lift this approach to decide distributed controllability in a more general setting, certain pitfalls are ahead that do not allow to directly apply the idea on arbitrary services. Some major issues are transitions with equal communication labels, a higher degree of uncertainty, and asynchronous communication labels on both ports (asynchronous communication normally needs to be restricted due to undecidability results, what the algorithm is not yet aware of).

Nevertheless we want to refine the algorithm in a way, that if we abstract an *arbitrary* two-port service S from one port and find a controlling service for the second port, then only because S is distributed controllable. For the adapter setting we want to show on a formal level, that the algorithm actually decides the problem. Thus if $S_1 \oplus E$ is distributed controllable, then the algorithm finds some S_2.

Further, we would like to characterize *all* adaptable service. This would allow us to actually do Service Discovery more efficiently without synthesizing an adapter for each candidate service S_2. We could simply *match* S_2 against the characterization.

References

1. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R., Toumani, F.: Developing adapters for web services integration. In: CAiSE. pp. 415–429 (2005)
2. Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: ICSOC. pp. 27–39 (2006)
3. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Business Process Management. pp. 65–80 (2006)
4. Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE Transactions on Services Computing 99(PrePrints) (2010)
5. Lohmann, N., Wolf, K.: Compact representations and efficient algorithms for operating guidelines. Fundam. Inform. 108(1-2), 43–62 (2011)
6. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (Jul 2007)
7. Wolf, K.: Does my service have partners? T. Petri Nets and Other Models of Concurrency 2, 152–171 (2009)
8. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2), 292–333 (1997)