Another Approach to Service Instance Migration

Nannette Liske[1], Niels Lohmann[2], Christian Stahl[3], and Karsten Wolf[2]

[1] Humboldt-Universtität zu Berlin, Institut für Informatik, Unter den Linden 6, 10099 Berlin, Germany

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

[3] Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, c.stahl@tue.nl

Abstract. Services change over time, be it for internal improvements, be it for external requirements such as new legal regulations. For long running services, it may even be necessary to change a service while instances are actually running and interacting with other services. This problem is referred to as *instance migration.* We present a novel approach to the behavioral (service protocol) aspects of instance migration. We apply techniques for finitely characterizing the set of all correctly interacting partners to a given service. The approach assures that migration does not introduce behavioral problems with *any* running partner of the original service. Our technique scales up to services with thousands of states, including models of real WS-BPEL processes.

1 Introduction

Service-oriented computing aims at creating complex systems by composing less complex systems called *services.* A service interacts with an environment consisting of other services. Such a complex system is subject to changes. To this end, individual services are substituted by other services. This becomes particularly challenging as services rely on each other and often nobody oversees the overall system—for example, if the individual services belong to different enterprises.

As a service is stateful rather than stateless, its exposed operations have to be invoked in a particular order, described by its business protocol. Throughout this paper we restrict ourselves to *business protocol changes* [1]; that is, we assume that nonfunctional properties (e.g., policies, quality of services) and semantical properties are not violated when changing a service S_{old} to a service S_{new}.

In our previous work we presented a procedure to decide for given services S_{old} and S_{new} whether S_{new} can substitute S_{old} [2]. The approach ensures that every service S that interacts properly with S_{old} also interacts properly with S_{new}. A properly interacting service is called a *partner.* In [3], we have applied these techniques to WS-BPEL processes.

However, this approach only covers the static and not the *dynamic* business protocol evolution. A service has *running instances.* In case a service is long running (e.g., an insurance), it is not feasible to wait until a running instance has terminated. Instead, instances have to be *migrated* to the new service definition. In this paper, we extend our previous work towards instance migration.

Given a running instance in a state q_{old} of S_{old}, instance migration is the task of finding some state q_{new} of S_{new} such that resuming the execution in state q_{new} does not affect any partner of S_{old}. We call the transition from q_{old} to q_{new} a *jumper transition*. Clearly, not for every state q_{old} may exist a jumper transition to a state q_{new}. Sometimes it might be necessary to continue the instance on S_{old} until a state is reached, where a migration is then possible. As a service may have arbitrary many running instances, we do not calculate suitable jumper transitions for each individual instance, but calculate them independently of actually running instances.

A jumper transition models that an engine is stopped, an instance is frozen and migrated to the new service definition. As our approach only guarantees behavioral correctness, a jumper transition may later disqualify for other reasons; for example, it may violate a data dependency or domain-specific restrictions. Hence, the set of jumper transitions can be seen as a safe overapproximation of possibilities to migrate an instance. That means, any *additional* jumper transition can introduce behavioral problems such as deadlocks in the interaction with some partner of S_{old}.

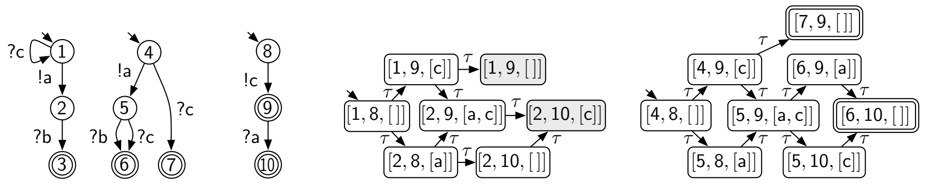
The contribution of this paper can be summarized as follows. We present an algorithm to compute the *maximal* set of jumper transitions. An implementation of this algorithm justifies the applicability of our approach to real-world WS-BPEL processes. In contrast to most existing approaches we assume an *asynchronous* communication model for services, because services are intended to communicate asynchronously rather than synchronously [4]; furthermore, we do not put restrictions on the structure of S_{old} and S_{new} and the way they are changed. We only require that every partner of S_{old} is a partner of S_{new}.

The necessary background from our previous work is introduced in Sect. 2. In Sect. 3, we formalize the problem of instance migration in terms of the introduced concepts. Our actual approach to migration is explained in Sect. 4. In Sect. 5, we report on an implementation and a case study. We compare our contribution to related work in Sect. 6 and, finally, we conclude the paper in Sect. 7.

2 Behavior of Services

We model a service as a *service automaton*. This model reflects the control flow and the business protocol while abstracting from semantics and nonfunctional properties. To a limited degree, data aspects may be coded within the states of a service automaton.

Definition 1 (Service automaton). *A* service automaton $S = [C_{in}, C_{out}, Q, q_0, \delta, \Omega]$ *consists of two disjoint sets* C_{in} *of inbound message channels and* C_{out} *of outbound message channels, a set of states* Q *including an initial state* q_0 *and a set of final states* Ω*, and a nondeterministic labeled transition relation* $\delta \subseteq Q \times (C_{in} \cup C_{out} \cup \{\tau\}) \times Q$.



(a) $\mathsf{S_A}$ (b) $\mathsf{S_B}$ (c) $\mathsf{S_C}$ (d) $\mathsf{S_A} \oplus \mathsf{S_C}$ (e) $\mathsf{S_B} \oplus \mathsf{S_C}$

Fig. 1: Running example: service automata and their composition.

We shall use indices for distinguishing the ingredients of different service automata. C_{in} and C_{out} establish the interface of S. Messages can be received from inbound channels and sent to outbound channels. In figures, we represent the interface implicitly by appending the symbol "?" to inbound channels and the symbol "!" to outbound channels. A transition with a label $a \in C_{in}$ receives a message from channel a. It is blocked if no message is available in the channel. A transition with a label $b \in C_{out}$ sends a message to channel b. We assume *asynchronous communication*, so sending transitions are never blocked. A transition with label τ ($\tau \notin (C_{in} \cup C_{out})$) represents any internal (i.e., non-communicating) activity. We shall write $q \xrightarrow{x}_S q'$ for $[q, x, q'] \in \delta$. Final states symbolize a successful completion of a service execution.

Example. As a running example, consider the service automata in Figs. 1(a)–(c). We use the standard graphical notations for automata and denote initial states by an inbound arrow and final states by double circles.

The interaction between services is defined through the concept of composition. For formalizing composition, we need to introduce *multisets*. A multiset is similar to a set, but permits multiple occurrences of elements. Formally, the number of occurrences of an element is represented as a mapping into the set $\mathbb{N}$ of natural numbers (including 0).

Definition 2 (Multiset). *A* multiset *A ranging over a set M is a mapping $A : M \to \mathbb{N}$. Multiset $A + B$ is defined by $(A + B)(x) = A(x) + B(x)$, for all x. A singleton multiset, written $[x]$ means $[x](x) = 1$ and $[x](y) = 0$, for $y \neq x$. The empty multiset $[\,]$ assigns 0 to all arguments. Let $Bags(M)$ be the set of all multisets ranging over set M.*

In the definition of composition, we use multisets in particular for representing the messages that are pending in channels. If, for some channel a, $M(a) = k$, then k messages are pending in channel a. Using multisets instead of queues, we assume asynchronous communication in which messages may overtake each other.

Definition 3 (Composition). *Services S_1 and S_2 are* composable *if $C_{in_1} = C_{out_2}$ and $C_{out_1} = C_{in_2}$. For composable services S_1 and S_2, the* composition *$S_c = S_1 \oplus S_2$ is the transition system (i.e., a service automaton with empty* *interface) S where $Q_c = Q_1 \times Bags(C_{in_1} \cup C_{in_2}) \times Q_2$, $q_{0c} = [q_{01}, [\,], q_{02}]$, $\Omega_c = \Omega_1 \times \{[\,]\} \times \Omega_2$, and the transition relation δ_c is determined as follows:*

send: *If $x \in C_{out_1}$, $q_1 \xrightarrow{x}_{S_1} q_1'$, $q_2 \in Q_2$, and $M \in Bags(C_{in_1} \cup C_{in_2})$, then $[q_1, M, q_2] \xrightarrow{\tau}_{S_c} [q_1', M + [x], q_2]$. Sending by S_2 is treated analogously.*

receive: *If $x \in C_{in_1}$, $q_1 \xrightarrow{x}_{S_1} q_1'$, $q_2 \in Q_2$, and $M \in Bags(C_{in_1} \cup C_{in_2})$, then $[q_1, M + [x], q_2] \xrightarrow{\tau}_{S_c} [q_1', M, q_2]$. Receiving by S_2 is treated analogously.*

internal: *If $q_1 \xrightarrow{\tau}_{S_1} q_1'$, $q_2 \in Q_2$, and $M \in Bags(C_{in_1} \cup C_{in_2})$, then $[q_1, M, q_2] \xrightarrow{\tau}_{S_c} [q_1', M, q_2]$. Internal transitions in S_2 are treated analogously.*

Example (cont.). The service automata $\mathsf{S_A}$ and $\mathsf{S_C}$ as well as $\mathsf{S_B}$ and $\mathsf{S_C}$ are composable (we assume all three services have three channels a, b, and c). Figures 1(d)–(e) depict the respective compositions.

Of course, only states reachable from the initial state are relevant. Using the notion of composition, we may define our correctness notion. We call an interaction correct if no *bad states* are reached in the composed system. We distinguish two kinds of bad states: deadlocks and overfull message channels. A deadlock is a non-final state where no transition is enabled. An overfull message channel is a state where some message channel contains more than k messages, for some given value k. As we treat the particular value of k as a parameter, we actually talk about *k-correctness.*

Definition 4 (k-correctness, k-partners). *Let $k > 0$ be a natural number. The interaction between composable services S_1 and S_2 is called k-*correct *if the composed system $S_1 \oplus S_2$ enables at least one transition in every non-final state $q \in Q_{S_1 \oplus S_2} \setminus \Omega_{S_1 \oplus S_2}$, and, for all states $[q_1, M, q_2]$ reachable from $q_{0_{S_1 \oplus S_2}}$ and all message channels x, $M(x) \leq k$. If the interaction between S_1 and S_2 is k-correct, we call S_1 a k-*partner *of S_2, and we call S_2 a k-*partner *of S_1. We write k-Partners(S) for the set of all k-partners of S.*

Example (cont.). The composition $\mathsf{S_A} \oplus \mathsf{S_C}$ contains two bad states (shaded gray). In contrast, the composition $\mathsf{S_B} \oplus \mathsf{S_C}$ does not contain any bad state, and in every reachable state at most one message is pending on each channel. Hence, $\mathsf{S_B}$ and $\mathsf{S_C}$ are 1-partners.

Treating overfull message channels as bad states has the advantage that a composed system has only finitely many reachable good states. This is essential for our approach. Besides, a crowded channel may indeed indicate a problem in the mutual interaction. In the real WS-BPEL processes we have analyzed so far, there is hardly any process in which more than a single message pending on a channel made sense. In practice, the value of k may stem from capacity considerations on the channels, from static analysis of the message transfer, or be chosen just sufficiently large. In the sequel, we shall assume that one particular value of k is fixed and we shall use the terms *correct* and *partner* without the preceding k.

In previous work, we were able to show that the (usually infinite) set *Partners*(S) can actually be finitely characterized. We provided an algorithm [5] and a tool for computing that characterization. The characterization exploits the fact that the set *Partners*(S) actually contains a top element in the simulation preorder (i.e., it can exhibit all behavior that any service in *Partners*(S) may exhibit).

Definition 5 (Simulation, most-permissive partner). *Let S_1 and S_2 be services with the same interface. A relation $\varrho \subseteq Q_1 \times Q_2$ is a* simulation relation *iff the following conditions are satisfied:*

Base: $[q_{01}, q_{02}] \in \varrho$;
Step: *If $[q_1, q_2] \in \varrho$ and, for some x, $[q_1, x, q_1'] \in \delta_1$ then there exists a state q_2' such that $[q_2, x, q_2'] \in \delta_2$ and $[q_1', q_2'] \in \varrho$.*

If there exists such a simulation relation, we say that S_2 simulates *S_1. An element $S^* \in$ Partners(S) is called* most-permissive partner of *S iff S^* simulates all elements of Partners(S).*

In [6], we showed that a (not necessarily unique) most-permissive partner exists for every service S unless *Partners*$(S) = \emptyset$.

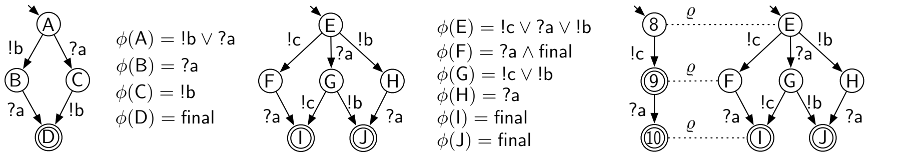
A simulation relation shows that the behavior of every partner of S is embedded in the behavior of a most-permissive partner. Hence, our finite characterization of *all* partners of S extends a most-permissive partner with Boolean annotations. They determine *which* embedded behaviors of the used most-permissive partner are actually in the set *Partners*(S). The formulas constrain the outgoing edges from states as well as the set of final states.

Definition 6 (Annotated automaton, matching). *An* annotated automaton *$A = [S_A, \phi]$ consists of a service automaton S_A and a mapping ϕ that assigns to each state of S_A a Boolean formula. The formulas use propositions from the set $C_{in_A} \cup C_{out_A} \cup \{\tau, \mathit{final}\}$.*
A service automaton S matches *with A if it uses the same interface as S_A and there is a simulation relation $\varrho \subseteq Q_S \times Q_{S_A}$ such that, for all $[q, q'] \in \varrho$, formula $\phi(q')$ is satisfied under the following assignment. Proposition x is true if there exists a state q_1 with $q \xrightarrow{x}_S q_1$. Proposition final is true if $q \in \Omega_S$.*
With Match(A), we denote the set of all services that match with A.

The main result of [5] is:

Proposition 1 (Operating guidelines). *For every service S with Partners$(S) \neq \emptyset$, there exists an annotated automaton $A = [S_A, \phi]$ (called* operating guidelines *of S or OG(S)) such that S_A is a particular most-permissive partner of S, subsequently referred to as MPP(S), and Partners(S) = Match(A).*

The most-permissive partner *MPP*(S) used as the underlying structure of operating guidelines has two important structural properties. First, it is deterministic (i.e., transitions leaving a state have different labels) no matter whether the service S is deterministic or nondeterministic. This fact makes the search for simulation relations rather efficient. Second, there exist transitions $[q, \tau, q]$ in every state. We are going to use this fact as an argument in subsequent proofs.



(a) $OG(\mathsf{S_A})$ (b) $OG(\mathsf{S_B})$ (c) matching $\mathsf{S_C}$ with $OG(\mathsf{S_B})$

Fig. 2: Running example: operating guidelines and matching.

Example (cont.). The operating guidelines of $\mathsf{S_A}$ and $\mathsf{S_B}$ (cf. Fig. 1) are depicted in Figs. 2(a)–(b). To increase legibility, we refrained from showing the τ-loops. The formula $\phi(\mathsf{A}) = \mathsf{!b} \vee \mathsf{?a}$ can be interpreted as a partner must send a message to channel b or receive a message from channel a; $\phi(\mathsf{F}) = \mathsf{?a} \wedge \mathsf{final}$ means that a partner must be in a final state, but still be able to receive a message from channel a. For the ease of presentation we also do not show the τ-disjunct in each annotation. For example, $\phi(\mathsf{F})$ is $\phi(\mathsf{F}) = (\mathsf{?a} \wedge \mathsf{final}) \vee \tau$; that is, a partner may also execute an internal step. As $\mathsf{S_C}$ is a partner of $\mathsf{S_B}$, it matches with $OG(\mathsf{S_B})$. The simulation relation ϱ is depicted in Fig. 2(c). It can be easily verified that the formulas are also satisfied. As $\mathsf{S_C}$ is not a partner of $\mathsf{S_A}$, there is no matching between $\mathsf{S_C}$ and $OG(\mathsf{S_A})$.

We have already described a number of applications of operating guidelines to problems related to service behavior, including test case generation [7], service correction [8], and service transformation [3]. One that we actually shall apply subsequently is related to *substitutability* (i.e., static business protocol evolution). Informally, substitutability states that every service that interacts correctly with S_1 will also interact correctly with S_2. This means that S_1 can be safely substituted by S_2 (this time assuming that there are no running instances).

Definition 7 (Substitutability). *Service S_1 is* substitutable *with service S_2 if $Partners(S_1) \subseteq Partners(S_2)$.*

Substitutability (which is an inclusion between infinite sets) can be checked using operating guidelines [2]. We need to check a simulation relation and implications between annotations:

Proposition 2 (Checking substitutability). *Let S_1 and S_2 be services with the same interface, $Partners(S_1) \neq \emptyset$, and $Partners(S_2) \neq \emptyset$. Let $OG(S_1) = [MPP(S_1), \phi_1]$ and $OG(S_2) = [MPP(S_2), \phi_2]$ be the corresponding operating guidelines. Then S_1 is substitutable with S_2 if and only if there is a simulation relation $\varrho \subseteq Q_{MPP(S_1)} \times Q_{MPP(S_2)}$ such that, for all $[q_1, q_2] \in \varrho$, the formula $(\phi(q_1) \implies \phi(q_2))$ is a tautology (i.e., true in all assignments).*

Implementations of all techniques referred to in this section are available at `http://www.service-technology.org/tools`.

Example (cont.). By checking Proposition 2, we can verify that $\mathit{Partners}(\mathsf{S_A}) \subseteq \mathit{Partners}(\mathsf{S_B})$; that is, $\mathsf{S_A}$ is substitutable with $\mathsf{S_B}$. As we saw earlier, the converse does not hold, because $\mathsf{S_C} \notin \mathit{Partners}(\mathsf{S_A})$ (cf. Fig. 1(d)).

3 Formalization of Instance Migration

Assume throughout this section that we want to migrate an instance of a service S_{old} to an instance of S_{new}. We generally assume that S_{old} and S_{new} have the same interface. Furthermore we require that S_{old} is substitutable with S_{new}. The assumption of substitutability is reasonable as it allows us immediately to migrate an instance of S_{old} being in its initial state to an instance of S_{new} being in its initial state. Furthermore, substitutability in connection with the assumption $S \in \mathit{Partners}(S_{old})$ gives us $S \in \mathit{Partners}(S_{new})$ which is also a desirable property.

An actual migration can be modeled as an internal transition from a state q_{old} of service S_{old} into a state q_{new} of service S_{new}. We call such a transition *jumper transition*. This kind of modeling abstracts from technical details like the process of freezing S_{old} (with all its parallel threads) in some intermediate state, transferring data to the new service and finally to start S_{new} in some non-initial state. In this sense, our approach considers behavior in isolation and abstracts from other aspects which are indeed relevant for instance migration.

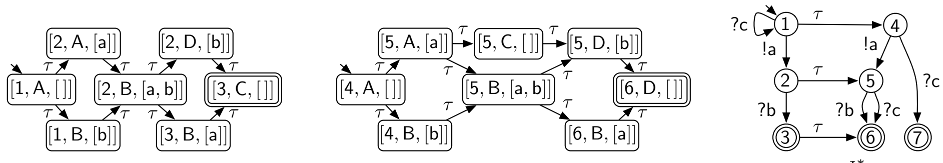
Formally, we are not just interested in one particular jumper transition. Instead, we would like to find *all* feasible jumper transitions. That is, we aim at the calculation of a largest possible set $J \subseteq Q_{old} \times Q_{new}$ of jumper transitions. This way, a single calculation of J may help in migrating *all* running instances of S_{old} regardless of how far the execution of instances has progressed.

It is worth mentioning that there may be states of S_{old} for which there is no corresponding state in S_{new}. In such a state, migration is not possible. Instead it is necessary to let S_{old} proceed to another state where a migration can take place.

Using relation J, the process of instance migration can be expressed in terms of a single model. In fact, we can place S_{old} next to S_{new} and insert all jumper transitions as internal transitions. This model captures all possible migration scenarios reflected in J. In the literature, the term *hybrid model* has been coined for this approach [9]. The following notation formalizes the idea and introduces a notation.

Definition 8 (Hybrid model). *Let S_1 and S_2 be services with disjoint sets of states ($Q_1 \cap Q_2 = \emptyset$) and equal interfaces. Let $J \subseteq Q_1 \times Q_2$. Then the* hybrid model *$S = \langle S_1 \overset{J}{\Longrightarrow} S_2 \rangle$ is a service automaton defined as follows. $Q_S = Q_1 \cup Q_2$, $C_{in_S} = C_{in_1} = C_{in_2}$, $C_{out_S} = C_{out_1} = C_{out_2}$, $q_{0S} = q_{01}$, $\Omega_S = \Omega_1 \cup \Omega_2$, $\delta_S = \delta_1 \cup \delta_2 \cup \{[q_1, \tau, q_2] \mid [q_1, q_2] \in J\}$.*

As the jumper transitions are internal to S_{old} and S_{new}, their occurrence is under full control of the provider of these service. For this reason, the hybrid model indeed reflects the process of migration of arbitrary instances of S_{old}.



(a) $\mathsf{S_A} \oplus \mathit{MPP}(\mathsf{S_A})$ (b) $\mathsf{S_B} \oplus \mathit{MPP}(\mathsf{S_A})$ (c) $\langle \mathsf{S_A} \xRightarrow{J^*} \mathsf{S_B} \rangle$

Fig. 3: Running example: constructing the hybrid model.

Using the notion of a hybrid model, we may state the correctness requirement on J. Essentially, we would like that every partner S of S_{old} interacts correctly with the hybrid model. In other words, interaction does not lead to bad states before, during, or after the migration.

Definition 9 (Feasible migration). *Let S_{old} and S_{new} be services. The migration relation $J \subseteq Q_{old} \times Q_{new}$ is* feasible *if $Partners(S_{old}) \subseteq Partners(\langle S_{old} \xRightarrow{J} S_{new} \rangle)$.*

4 Migration Approach

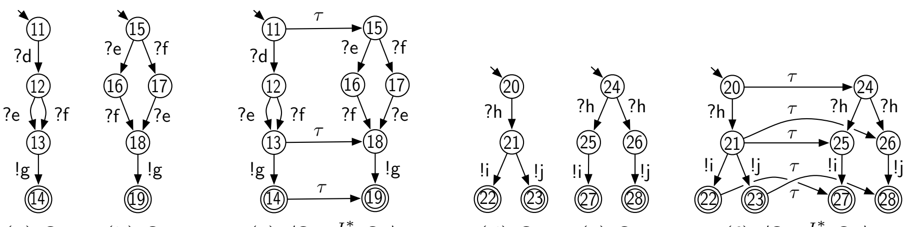
In this section, we first exhibit a particular migration relation J^*. Then we show that J^* is feasible. We continue with a discussion on the maximality of J^*.

The next definition shall determine J^*. To this end, remember that a migration must be correct independently of the interacting partner S of S_{old} and the state of S. A jumper transition $[q_{old}, q_{new}]$ means that we switch from a reachable state $[q_{old}, M, q]$ of $S_{old} \oplus S$ into state $[q_{new}, M, q]$. Of course, we are on the safe side if $[q_{new}, M, q]$ is a reachable state of $S_{new} \oplus S$. This is due to the fact that S is a partner of S_{old} and, by substitutability, a partner of S_{new}, too. Being a partner, no bad states can be reached from $[q_{new}, M, q]$ which is all we desire.

This observation leads straight to the definition of J^*, with just one modification. Instead of considering an arbitrary service S, we consider the particular service $MPP(S_{old})$ (which we can compute from S_{old}). This is a reasonable choice as $MPP(S_{old})$ embeds the behavior of all partners of S_{old}.

Definition 10 (Migration relation J*). *Let S_{old} and S_{new} be substitutable services. Then $J^* = \{[q_{old}, q_{new}] \mid$ for all $[q_{old}, M, q] \in Q_{S_{old} \oplus MPP(S_{old})}$ holds: $[q_{new}, M, q] \in Q_{S_{new} \oplus MPP(S_{old})}\}$.*

Example (cont.). As $\mathsf{S_A}$ is substitutable with $\mathsf{S_B}$, we can calculate the migration relation to migrate states from instances from $\mathsf{S_A}$ to $\mathsf{S_B}$. The compositions with the most-permissive partner of $\mathsf{S_A}$ are depicted in Figs. 3(a)–(b). Among the states, we have $\{[2, \mathsf{A}, [\mathsf{a}]], [2, \mathsf{B}, [\mathsf{a}, \mathsf{b}]], [2, \mathsf{D}, [\mathsf{b}]]\} \subseteq Q_{\mathsf{S_A} \oplus MPP(\mathsf{S_A})}$ and $\{[5, \mathsf{A}, [\mathsf{a}]], [5, \mathsf{B}, [\mathsf{a}, \mathsf{b}]], [5, \mathsf{C}, [\,]], [5, \mathsf{D}, [\mathsf{b}]]\} \subseteq Q_{\mathsf{S_B} \oplus MPP(\mathsf{S_A})}$. From Definition 10, we can conclude that $[2, 5] \in J^*$: We can safely migrate state 2 to state 5 without jeopardizing correctness. The resulting hybrid model is depicted in Fig. 3(c).



(a) $\mathsf{S_D}$ (b) $\mathsf{S_E}$ (c) $\langle \mathsf{S_D} \xRightarrow{J^*} \mathsf{S_E}\rangle$ (d) $\mathsf{S_F}$ (e) $\mathsf{S_G}$ (f) $\langle \mathsf{S_F} \xRightarrow{J^*} \mathsf{S_G}\rangle$

Fig. 4: Further migration examples.

Figures 4(a)–(c) show an example in which migration is not possible in every state of the old service: state 12 of service $\mathsf{S_D}$ cannot be migrated to a state of $\mathsf{S_E}$. Only after $\mathsf{S_D}$ proceeds to state 13, migration to $\mathsf{S_E}$ is again possible.

In the remainder of this section, we will focus on feasibility and maximality of the migration relation.

Feasability of the migration relation

We will first show that the migration relation J^* is indeed feasible; that is, the jumper transitions induced by J^* do not introduce bad states in the interaction with running partners of S_{old}.

Theorem 1. *J^* is feasible.*

Proof. Assume the contrary. Then there exists a service $S \in \mathit{Partners}(S_{old}) \setminus \mathit{Partners}(\langle S_{old} \xRightarrow{J^*} S_{new}\rangle)$. For not being a partner of $\langle S_{old} \xRightarrow{J^*} S_{new}\rangle$, there must be an execution in $\langle S_{old} \xRightarrow{J^*} S_{new}\rangle \oplus S$ that leads to a bad state. Consider first the case that the sequence does not contain any jumper transition. Then the sequence is actually a sequence in $S_1 \oplus S$ which contradicts the assumption $S \in \mathit{Partners}(S_{old})$.

Consider now the case that a jumper transition $[q_{old}, q_{new}]$ occurs in the considered execution. By our construction, only one such transition can occur. We now produce a contradiction by exhibiting a partner S_{bad} of S_{old} which is not a partner of S_{new}. This contradicts the assumed substitutability of the involved services.

As $S \in \mathit{Partners}(S_{old})$ there is a simulation relation $\varrho \subseteq Q_S \times Q_{\mathit{MPP}(S_{old})}$ such that the conditions of Definition 6 are met. Let $S_{bad} = \langle \mathit{MPP}(S_{old}) \xRightarrow{\varrho^{-1}} S\rangle$. We first show $S_{bad} \in \mathit{Partners}(S_{old})$. By Proposition 1, it is sufficient to show that S_{bad} matches with the operating guidelines $\mathit{OG}(S_{old}) = [\mathit{MPP}(S_{old}), \phi]$. To this end, consider the relation $\varrho_{bad} = \varrho \cup id_{Q_{\mathit{MPP}(S_{old})}}$ between S_{bad} and $\mathit{MPP}(S_{old})$.[4] ϱ_{bad} is actually a simulation. For states in $\mathit{MPP}(S_{old})$ this is easily verified as the

[4] By construction of S_{bad}, we have $Q_{\mathit{MPP}(S_{old})} \subset Q_{S_{bad}}$, so $id_{Q_{\mathit{MPP}(S_{old})}}$ can be seen as a relation between $Q_{S_{bad}}$ and $Q_{\mathit{MPP}(S_{old})}$.

identity is indeed a simulation between a service and itself. Consider a jumper transition $[q_1, q_2] \in Q_{MPP(S_{old})} \times Q_S$. We arrive with $[q_1, q_1] \in id_{Q_{MPP(S_{old})}}$. As $MPP(S_{old})$ has τ-loops in every state, the jumper transition can be matched, leading to the pair $[q_2, q_1]$. The jumper transition $[q_1, q_2]$ has been introduced only if $[q_2, q_1] \in \varrho$. So, $[q_2, q_1]$ is indeed in the simulation relation. For the remaining transitions, simulation follows from the fact the ϱ has been chosen as a simulation between S and $MPP(S_{old})$. For completing the matching procedure, we have to show that the assignments determined by S_{bad} satisfy the related annotations in $MPP(S_{old})$. For those states of S_{bad} which are in $MPP(S_{old})$, this is obvious as $MPP(S_{old})$ is indeed a partner of S_{old} and the identity is a valid simulation relation. For those states of S_{bad} which are in S, satisfaction of the annotations follows from the choice of ϱ.

We conclude our proof by showing that S_{bad} is not a partner of S_{new} which contradicts the assumed substitutability. For this purpose, return to the assumed execution sequence that brings $\langle S_{old} \overset{J^*}{\Longrightarrow} S_{new} \rangle \oplus S$ into a bad state. We replay this sequence in $S_{bad} \oplus S_{new}$. Assume that the jumper transition $[q_{old}, q_{new}]$ occurred in the context of state q of S and a bag M of pending messages. In other words, the composed system $\langle S_{old} \overset{J^*}{\Longrightarrow} S_{new} \rangle \oplus S$ contained the transition $[q_{old}, M, q] \xrightarrow{\tau} [q_{new}, M, q]$. As $MPP(S_{old})$ embeds the behavior of S, we can find a corresponding sequence in $S_{old} \oplus MPP(S_{old})$ that reaches a state $[q_{old}, M, q^*]$ such that $[q, q^*]$ are in the simulation relation between S and $MPP(S_{old})$. The latter sequence is also executable in S_{bad}. Now, let the jumper transition $[q_{old}, q_{new}]$ occur, followed by the jumper transition $[q^*, q]$ in S_{bad}. The resulting state is $[q_{new}, M, q]$. This is exactly the state reached by the jumper transition in the originally considered sequence. Hence, the remainder of the original sequence may be appended and shows that $S_{bad} \oplus S_{new}$ may reach a bad state. $\square$

Maximality of the migration relation

Now we turn to the question of maximality of J^*. For this purpose, consider a transition $[q_{old}, q_{new}] \in Q_{old} \times Q_{new}$ which is not contained in J^*. By Definition 10, this means that there exists at least one service (e.g., $MPP(S_{old})$) and a reachable state $[q_{old}, M, q]$, from where the migration leads to a state $[q_{new}, M, q]$, is not reachable in the composition $S_{new} \oplus MPP(S_{old})$. As $MPP(S_{old})$ is most-permissive, this means that actually *no* service in $Partners(S_{old})$ is able to reach $[q_{new}, M, q]$. That is, migration would bring us into a part of S_{new} which is not intended to be reached by *any* partner of S_{old}. Though continuation from such a state may or may not lead to bad states, we believe that it is very unplausible to continue interaction there. In this light, we may claim that our migration relation is the largest possible set of jumper transitions.

5 Case Study and Implementation

For evaluating our proposed approach, we have implemented the computation of the migration relation J^* of Definition 10. The algorithm takes the two service

Table 1: Numbers on the services used in the case study.

service S	$\lvert C_{in_S} \cup C_{out_S} \rvert$	$\lvert Q_S \rvert$	$\lvert Q_{MPP(S)} \rvert$	$\lvert Q_{PV(S)} \rvert$
Travel Service	10	34	192	202
Purchase Order	10	402	168	176
Ticket Reservation	9	304	110	118
Internal Order	7	1,516	96	104
Contract Negotiation	11	784	576	588
Deliver Finished Goods	14	182	1,632	1,394
Passport Application	11	14,569	1,536	1,540

Table 2: Numbers on the calculation of the maximal migration relation.

migration $S \Rightarrow PV(S)$	$\lvert Q_{MPP(S) \oplus PV(S)} \rvert$	search space	$\lvert J^* \rvert$	time (sec)
Travel Service	2,976	3,333,120	49	2.1
Ticket Reservation	1,031	4,886,940	359	0.6
Purchase Order	2,545	19,851,000	429	1.3
Internal Order	1,455	34,460,220	1,613	0.9
Contract Negotiation	17,331	856,844,640	866	12.9
Deliver Finished Goods	60,753	1,050,783,888	197	123.1
Passport Application	100,975	990,199,624,400	22,382	518.1

automata S_{old} and S_{new} as its input. First, it computes the most-permissive partner $MPP(S_{old})$. According to the technique used in [6], this calculation returns not only $MPP(S_{old})$, but also the set of states $Q_{S_{old} \oplus MPP(S_{old})}$. Consequently, a second calculation is only required for producing $Q_{S_{new} \oplus MPP(S_{old})}$. The two sets of states are then sorted according to a criterion that enables an efficient verification of the implications in Definition 10.

The services used in the case study were anonymized real WS-BPEL processes provided by a small German consulting company. They implement several business processes from different domains such as government administration, industrial production, and customer services. To apply our formal framework, we first translated these WS-BPEL processes into service automata [10].

Table 1 lists the size of the interface (i.e., the number of inbound and outbound channels) and the number of states of the service automata. Due to complex internal behavior such as fault and compensation handling, the services have up to 14,569 states. The forth column contains the number of states of the most-permissive partner. For the considered services, the most-permissive partner usually has less states, because it only describes the interaction behavior and does not contain internal behavior other than the τ-loops mentioned in the remarks below Proposition 1.

In the case study, we migrated each service to its *public view*. The public view of a service S is a service $PV(S)$ that can be canonically derived from the operating guidelines $OG(S)$ such that holds: $OG(PV(S)) = OG(S)$. Hence, the public view $PV(S)$ is (1) by design substitutable with the original service

S. Being constructed from the operating guidelines, however, it (2) abstracts from internal behavior and usually has no structural relationship to S. For these reasons, we chose the public view to benchmark the migration approach. The last column of Table 1 lists the number of states of the public views.

Table 2 lists information about the migration. To calculate the migration relation J^*, the composition of the most-permissive partner of the original service S (called "S_{old}" before) and the public view of S ("S_{new}") has to be considered. At maximum, this composition contained more than 100,000 states. The third column ("search space") lists the number of states to check in Definition 10. As this number depends on several state spaces, it heavily suffers from state explosion and nearly reaches 10^{12} states for the Identity Card service. Nevertheless, this number is only a theoretical bound, because (1) only two generator sets are kept in memory (the state spaces of the compositions with the most-permissive partner), and (2) these generators are sorted and represented to quickly detect violation of the criterion of Definition 10.

The forth column of Table 2 lists the size of the maximal migration relation J^* (i.e., the number of jumper transitions). Compared to the states of the involved services and the search spaces, this relation is rather small. The last column shows that most results were available in a few seconds. The maximal calculation took a bit more than eight minutes.[5] Though the implementation is only a prototype to prove the concept, we claim that these numbers are acceptable: The whole setting of instance migration is motivated by long-running services in which a few minutes of calculation is negligible. Furthermore, once the jumper transitions have been calculated, they can be applied to any number of running instances.

The case study of this paper can be replayed using the Web-based implementation of the tools available at `http://service-technology.org/live/migration`. At the same URL, the tools and the examples of the case study can be downloaded.

6 Related work

Instance migration (or dynamic business protocol evolution) is a hot topic which has been studied by many researchers. Our proposed approach is inspired by the notion of state replaceability in [11], where all pairs of states (q_{old}, q_{new}) of S_{old} and S_{new} are determined such that S_{old} and S_{new} are forward and backward compatible. Backward compatible means that every path from the initial state of S_{old} to q_{old} is a valid path from the initial state of S_{new} to q_{new}. In contrast, forward compatible means that every path from q_{old} to a final state in S_{old} is also a valid path from q_{new} to a final state in S_{new}. Besides state replaceability, several weaker notions are presented in [11].

We identify the following differences to our approach: In [11] it is guaranteed that a service can always reach a final state, whereas our approach only guarantees deadlock freedom. As a restriction, synchronous communication is assumed in [11].

[5] The reported experiments were conducted on an Apple MacBook with a 2.16 GHz Intel Core 2 Duo processor. No calculation required more than 1 GB of memory.

In contrast, service automata model asynchronous communication, as services are intended to communicate asynchronously rather than synchronously [4]. Although not explicitly mentioned, the approach in [11] is restricted to deterministic services, as forward and backward compatibility only relies on trace inclusion. For example, if we assume *synchronous communication*, then services $\mathsf{S_F}$ and $\mathsf{S_G}$ in Figs. 4(d) and 4(e) cannot be migrated. A service that first executes h and then expects i is a partner for $\mathsf{S_F}$ but not for $\mathsf{S_G}$ ($\mathsf{S_G}$ may enter the right branch causing a deadlock). However, by looking at traces, this counterexample cannot be detected. Moreover, the states $\mathsf{20}$ and $\mathsf{24}$ are forward and backward compatible. In contrast, our proposed method works for deterministic and non-deterministic services. As the crucial difference, we do not compare the structures of S_{old} and S_{new} but use information about all partners of S_{old} to compute the jumper transitions.

Dynamic evolution has been in particular studied in the field of workflows; see [12, 13] for an overview. Some approaches [14, 15] calculate the part of the workflow definition that is affected by the change (i.e., the change region). If an instance of S_{old} is not in the change region, it can be safely replaced by S_{new}. Other approaches like [16] and [17] are restricted to acyclic workflow models. In addition, [17] and also [18] take only the history into account to decide migration. Hence, the migrated instance may deadlock.

In [19] inheritance (i.e., branching bisimulation) is proposed for relating two workflows S_{old} and S_{new}. Transfer rules are presented to map a state of S_{old} to a state of S_{new}. The transfer rules ensure proper termination of an instance in S_{new}. The approach can also be combined with dynamic change regions in [15] to widen the applicability. However, branching bisimulation is too strict; for example, services $\mathsf{S_F}$ and $\mathsf{S_G}$ in Figs. 4(d) and 4(e) are not branching bisimular, and hence could not be migrated. In contrast, using our approach a migration can be computed (see Fig. 4(f)).

The ADEPT2 framework [13] offers support to dynamically change a workflow definition and to migrate running instances of the old workflow definition to the new one. The approach guarantees that no deadlocks or livelocks are introduced. Furthermore, the history of the migrated instance can be replayed on the new workflow definition. Thereby ADEPT2 also takes the data flow into consideration and ensures data consistency. However, the approach is restricted to workflows, whereas we consider services.

7 Conclusion

We provided an approach to the automated calculation of the maximal set of jumper transitions which model the possible migration of service instances. We addressed the behavioral aspect and took care that migration does not introduce reachable bad states. Other than this, the set of jumper transitions is reasonably large. The calculation of the set is possible within seconds to few minutes, considering real WS-BPEL processes. As instance migration is typically relevant for long-running services, this amount of time negligible. Though the results base on service automata, they can be easily applied to other service description languages once a translation to automata is specified. As such a translation is usually straightforward, the choice of service automata as formal model poses no intrinsic restrictions.

We are of course aware that our approach only considers behavior while it is necessary to obey several restrictions in several other aspects. Therefore, it is very well possible that some of our jumper transitions disqualify for reasons of data integrity or domain specific reasons. However, these issues can hardly avoid problems if the service runs into a bad state. Hence, our approach can be understood as a first overapproximation which reasonably reduces the combinatorics for subsequent consideration of other aspects for correct migration. Furthermore, data dependencies can be detected by techniques used in the area of static program analysis [20]. For WS-BPEL there exist such techniques already [21–23].

Due to the lack of tools and the fact that usually thousands of running instances have to be migrated, we think that our approach is a significant step towards supporting instance migration.

An interesting line of further research is to investigate the data aspect in more detail. As a result, we may get a smaller overapproximation. Furthermore, as a service composition may run on different servers another interesting line of further work is to migrate the state of each service separately rather than migrating the whole composition at once.

Acknowledgements Niels Lohmann and Karsten Wolf are funded by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

References

1. Papazoglou, M.P.: The challenges of service evolution. In: CAiSE 2008. LNCS 5074, Springer (2008) 1–15
2. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. LNCS ToPNoC **5460**(II) (2009) 172–191
3. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract WS-BPEL processes. In: WWW 2008, ACM (2008) 785–794
4. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (2007)
5. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: PETRI NETS 2007. LNCS 4546, Springer (2007) 321–341
6. Wolf, K.: Does my service have partners? LNCS ToPNoC **5460**(II) (2009) 152–171
7. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC 2008 Workshops. LNCS 5472, Springer (2009) 66–78
8. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: BPM 2008. LNCS 5240, Springer (2008) 132–147
9. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. In: ER 1996. LNCS 1157, Springer (1996) 438–455
10. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data Knowledge Engineering **64**(1) (2008) 38–54

11. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the dynamic evolution of web service protocols in service-oriented architectures. TWEB **2**(2) (2008)
12. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems - a survey. Data Knowl. Eng. **50**(1) (2004) 9–34
13. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. LNCS ToPNoC **5460**(II) (2009) 115–135
14. Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: COOCS 1995, ACM (1995) 10–21
15. Aalst, W.M.P.v.d.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers **3**(3) (2001) 297–317
16. Agostini, A., Michelis, G.D.: Improving flexibility of workflow management systems. In: Business Process Management, Models, Techniques, and Empirical Studies. LNCS 1806, Springer (2000) 218–234
17. Sadiq, S.W.: Handling dynamic schema change in process models. In: Australasian Database Conference. (2000) 120–126
18. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. **24**(3) (1998) 211–238
19. Aalst, W.M.P.v.d., Basten, T.: Inheritance of Workflows: An Approach to Tackling Problems Related to Change. Theoretical Computer Science **270**(1-2) (2002) 125–203
20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. 2nd edn. Springer-Verlag, Berlin (2005)
21. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In Dumas, M., Heckel, R., eds.: WS-FM 2007. Volume 4937 of LNCS., Springer (2008) 77–91
22. Moser, S., Martens, A., Gorlach, K., Amme, W., Godlinski, A.: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: SCC 2007, IEEE Computer Society (2007) 98–105
23. Heinze, T.S., Amme, W., Moser, S.: Generic CSSA-based pattern over Boolean data for an improved WS-BPEL to Petri net mappping. In: ICIW 2008, IEEE Computer Society (2008) 590–595