Conformance Testing for Asynchronously Communicating Services

Kathrin Kaschner

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

Abstract. We suggest a black box testing approach to examine conformance for stateful services. Here, we consider asynchronous communication in which messages can overtake each other during their transmission. For testing, we generate partner services that exchange messages with the implementation under test (IUT). From the observations made during testing, we are then able to infer whether the IUT conforms to its specification. We study how partner services need to be designed to serve conformance testing in an asynchronous setting and present an algorithm which generates a complete test suite.

1 Introduction

Modern software systems are more and more composed of a set of loosely-coupled *services*. Thereby, each service implements an encapsulated, self-contained functionality and communicates via message exchange with its *partner services*. For a proper interaction, the *behavior* of the involved services plays a central role; e.g., a whole composition of services may deadlock if a single service fails to send an expected message. To avoid failures, the behavior of a service should be tested thoroughly before it is deployed.

To face this issue, we propose a black box testing approach. That means, we examine the behavior of the implementation under test (IUT) from the partner's perspective without accessing the inner structure of the IUT. But usually, a service is not bound to a fixed set of partner services. Instead, partners may change frequently and can even be created after deployment of the IUT. Since they are unknown at testing time, real partners or their abstract behavioral description (public view) cannot be used for testing. Instead, we *synthesize* partner services and use them as test cases. Their intended purpose is to imitate the behavior of real partners with which the IUT will be potentially confronted in practise. Moreover, they observe the IUT's reactions during a test run such that we are able to conclude whether the interactions are conform to the IUT's specification.

Throughout the paper, we consider services with *stateful* interaction; i.e., the communication follows a more or less complex protocol in which several messages are exchanged between the services. In addition, we assume an *asynchronous* message passing, in which messages can overtake each other. This is motivated by the fact that services usually communicate over the internet, which does not preserve the message order during transmission. In contrast to *synchronous* communication, the sending of an asynchronous message cannot be blocked by the partner, but is executed independently from the receiving. Further, a message may be sent in advance if it is ensured that the partner (if it follows the protocol) will eventually receive it; i.e., between sending and receiving a message,

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 108–124, 2011.

other messages can be exchanged. All these non-trivial aspects need to be taken into account when test partners are synthesized for substituting real partner services.

To automate the test case generation procedure, a formal model of the specification is required. The test partners are then derived from the model. With our tests we focus on the specified behavior only (conformance testing) and do not examine whether the implementation is robust against undesired messages (robustness testing). To offer exhaustiveness, we design the test partners such that each possible behavior defined by the formal specification can be triggered during testing *and* thereby the asynchronous characteristics are taken into consideration (e.g., test partners may send messages in advance for imitating real partners adequately). Moreover, we present a selection algorithm to limit the number of test cases. The resulting test suite is complete in the sense that (1) each detected failure indicates an error in the implementation (soundness) and (2) each failure that can be discovered by any partner derivable from the formal specification can also be detected by a partner of the test suite (exhaustiveness). That way, the behavior of any potential real partner is considered best possible by the created test suite, and we can be confident that the IUT will interact correctly in practice – if it passes the test suite successfully.

This paper is organized as follows: In Sect. 2, we define when an observation made during testing is considered as correct. In Sect. 3, we study how the test partners for conformance testing need to be designed. Based on these considerations, Sect. 4 shows how a complete test suite can be generated. Section 5 summarizes related work and Sect. 6 concludes the paper.

2 Correct Behavior

While the specification can be assumed to be given in a formal description, the implementation is a physical, real object which is not amenable to formal reasoning. However, to deal with implementations in a formal way, we assume for our theory that for any implementation there is a formal model. But we only require its existence, not the model itself. This is common usage when the implementation is seen as a black box and formal testing is conducted [1]. Then, during conformance testing we infer from the observations made during testing, the (unknown) formal model of the IUT and decide with the help of the testing theory whether the implementation complies to its specification.

Formalizing Behavior. To formally reason about service behavior, we use *service automata* [2,3]. They are related to input output transitions systems (IOTS) [1] and I/O automata [4], but perform communication asynchronously via unidirectional *message channels*. When modeling behavior by service automata, we abstract from data. The set of the abstract messages is denoted by $\mathbb{M}$ and is assumed to be finite. The interface of a service is formed by a set of *pins* $\Pi \subseteq \mathbb{M} \times \{\mathbf{i}, \mathbf{o}\}$. Pin $\pi = [m, \mathbf{i}]$ is an *inbound pin* and $\pi = [m, \mathbf{o}]$ is an *outbound pin*. The *dual pin* of $\pi = [m, z]$ is defined as $\overline{\pi} = [m, z']$ with $z' \neq z$. From a set of pins Π we derive a set of *communicating events* $\mathbb{E}^{\Pi}$ such that $!m \in \mathbb{E}^{\Pi}$ iff there is a pin $\pi = [m, \mathbf{o}] \in \Pi$ (sending a message m) and $?m \in \mathbb{E}^{\Pi}$ iff there is a pin $\pi = [m, \mathbf{i}] \in \Pi$ (receiving a message m). *Internal events* (i.e., non-communicating events) are pooled in a set $\mathbb{E}^{\tau}$. The behavior itself is expressed by a finite state machine whose transitions are labeled with communicating events and internal events.

Definition 1 (service automaton [2,3]) *A* service automaton $A = [Q, q_0, \Omega, \Pi, \mathbb{E}^\tau, \delta]$ *consists of a finite set of* states Q*, an* initial state $q_0 \in Q$*, a set of* final states $\Omega \subseteq Q$*, a finite set of* pins $\Pi \subseteq \mathbb{M} \times \{\mathbf{i}, \mathbf{o}\}$ *(with* $\pi \in \Pi$ *implies* $\overline{\pi} \notin \Pi$*), a finite set of* internal events $\mathbb{E}^\tau$ $(\mathbb{E}^\tau \cap \mathbb{E}^\Pi = \emptyset)$ *and a* transition relation $\delta \subseteq Q \times (\mathbb{E}^\Pi \cup \mathbb{E}^\tau) \times Q$.

We write $q \xrightarrow{e} q'$ for $[q, e, q'] \in \delta$. For a state q, we express with $q \xrightarrow{e}$ that there is a state q' with $q \xrightarrow{e} q'$. If q has no successors, we write $q \not\rightarrow$. A state q' is *reachable from a state* q iff there are events $e_1, \ldots, e_n \in \mathbb{E}^\Pi \cup \mathbb{E}^\tau$ and states $q_1, \ldots, q_n \in Q$ with $q \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots \xrightarrow{e_{n-1}} q_{n-1} \xrightarrow{e_n} q_n = q'$ or $q = q'$. A state is *reachable* iff it is reachable from the initial state. A non-final state q ($q \notin \Omega$) is a *deadlock state* iff $q \not\rightarrow$ holds. Since specifications with deadlocks are seen as ill-designed, we consider deadlock-free specifications in our theory only.

As an example, service automaton A in Fig. 1(a) is initially waiting for message a or message c. In case c is received, message z is sent back. In case message a is received, A decides non-deterministically (modeled by a branch of two internal τ steps) whether it sends message y and is then waiting for message b, or it sends message x after receiving message b. The set of pins of B, C and D are dual to the set of pins of A. Some pins of C and D are not used by the internal process of C and D.

Formalizing Communication. To formalize the interplay of two services we use the concept of *composition*. In the model, communication is realized by connecting dual pins with a unidirectional channel. That way, a message of shape m is sent via an outbound pin $[m, \mathbf{o}]$ through the corresponding channel. There, m is pending until the service on the other channel's side receives it via its inbound pin $[m, \mathbf{i}]$. Throughout the paper, we consider those services as *composable*, whose pin sets are dual to each other. Note, that the criterion is merely syntactically and independent of the actual behavior. In particular, it does not guarantee that a sent message is indeed received on the other channel's side.

Two composable service automata are also called *partners*. In Fig. 1, service automata B, C and D are partners of A.

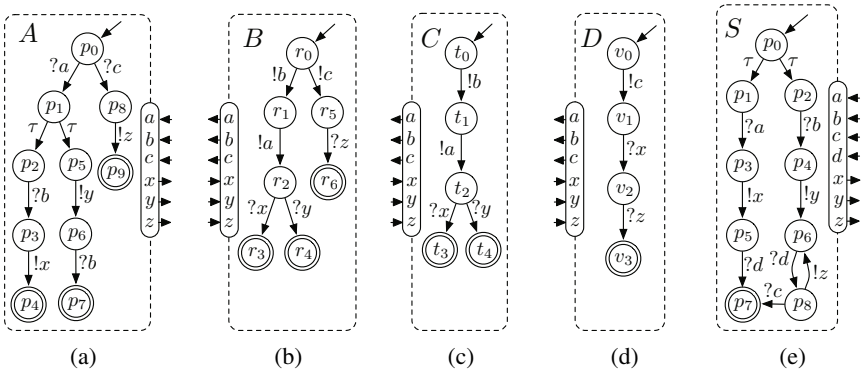


Fig. 1. Five service automata. In the graphical representation initial states are denoted by an incoming arc from nowhere and final states are circled twice. The interface with its inbound and outbound pins is depicted on the dashed box.

Formally, composition of services (see Def. 2) is a product automaton construction, adjusted to the characteristics of asynchronous message passing between the involved services. A state comprises the states of the involved services and the messages pending in the channels, represented by a multiset $\mathcal{B}$. A state $[q_A, q_B, \mathcal{B}]$ with $[q_A, q_B, \mathcal{B}] \not\rightarrow$ is declared as a final state iff $q_A \in \Omega_A$, $q_B \in \Omega_B$ and $\mathcal{B} = [\,]$. Whereas the first two conditions are obvious, the requirement for empty channels at termination is motivated by the fact that messages may contain important information such as payment details. Hence, the sender of such a message wants to ensure that it is actually received rather than ignored. In Def. 2, $Bags(\mathbb{M})$ denotes the set of all multisets over set $\mathbb{M}$, $[\,]$ denotes the empty multiset, and $+$ is the elementwise addition of multisets.

Definition 2 (composition of service automata [2,3]). *The* composition *of two partners A and B ($E_A^{\Pi}, E_B^{\Pi}, E_A^{\tau}$ and E_B^{τ} are pairwise disjoint) is the service automaton $A \oplus B = [Q, q_0, \Omega, \Pi, \mathbb{E}^{\tau}, \delta]$ consisting of $Q := Q_A \times Q_B \times Bags(\mathbb{M})$, $q_0 := [q_{0A}, q_{0B}, [\,]]$, $\Omega := \Omega_A \times \Omega_B \times \{[\,]\}$, $\Pi := \emptyset$, $E^{\tau} := E_A^{\Pi} \cup E_A^{\Pi} \cup E_B^{\tau} \cup E_B^{\tau}$, and δ containing exactly the following elements:*
for all $m \in \mathbb{M}$, $\tau \in E_A^{\tau} \cup E_B^{\tau}$ and $\mathcal{B} \in Bags(\mathbb{M})$,

- $[q_A, q_B, \mathcal{B}] \xrightarrow{!m} [q'_A, q_B, \mathcal{B} + [m]]$, *iff* $q_A \xrightarrow{!m}_A q'_A$ *(send event in A),*
- $[q_A, q_B, \mathcal{B}] \xrightarrow{!m} [q_A, q'_B, \mathcal{B} + [m]]$, *iff* $q_B \xrightarrow{!m}_B q'_B$ *(send event in B),*
- $[q_A, q_B, \mathcal{B} + [m]] \xrightarrow{?m} [q'_A, q_B, \mathcal{B}]$, *iff* $q_A \xrightarrow{?m}_A q'_A$ *(receive event in A),*
- $[q_A, q_B, \mathcal{B} + [m]] \xrightarrow{?m} [q_A, q'_B, \mathcal{B}]$, *iff* $q_B \xrightarrow{?m}_B q'_B$ *(receive event in B),*
- $[q_A, q_B, \mathcal{B}] \xrightarrow{\tau} [q'_A, q_B, \mathcal{B}]$, *iff* $q_A \xrightarrow{\tau}_A q'_A$ *(internal step in A),*
- $[q_A, q_B, \mathcal{B}] \xrightarrow{\tau} [q_A, q'_B, \mathcal{B}]$, *iff* $q_B \xrightarrow{\tau}_B q'_B$ *(internal step in B).*

Figure 2 shows the composition of service automata A and B of Fig. 1. It is free of deadlocks. In contrast, the composition of A and D deadlocks in state $[p_9, v_1, [z]]$ because A never sends message x after receiving message c.

A composition is *k-bounded* iff in all reachable states the number of identical messages in $\mathcal{B}$ does not exceed a value k. This property is motivated by the middleware, that realizes the message exchange between services. Since it has only finite storage available, we limit the capacity of messages in each channel to k, in our theory. The value of k is either known by the middleware's characteristics or chosen carefully. Then, a k-bounded composition does not overflow the channels. As an example, the composition of Fig. 2 is 1-bounded.

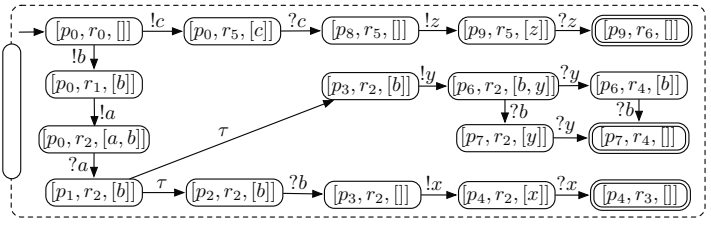


Fig. 2. Composition of A and B of Fig. 1 (only the reachable states are depicted)

Formalizing Observations. As mentioned earlier, we aim at generating partner services for the IUT. During the testing procedure, each generated partner service and IUT are executed in isolation within a test environment. Thereby, we assume that the tester can always observe the IUT's execution status; i.e., still in execution or terminated. A failure is detected as soon as a partner observes an unforeseen reaction of the IUT; i.e., the absence of specified message, sending wrong message or a wrong execution status. Due to the asynchronous setting it is not observable whether the IUT actually receives a message or not. In the following, we define when observations are classified as correct or incorrect. To this end, we give a formal definition of a *(test) run*.

Definition 3 (run). *For a service automaton A, a finite or infinite sequence $\sigma = q_0 e_1 q_1 e_2 q_2 \ldots$ of states and events is called* run *iff q_0 is the initial state of A and $q_{i-1} \xrightarrow{e_i} q_i$ for all $i \in \{1, 2, ...\}$. A run is* maximal *iff it is infinite or it ends in a state q_n with $q_n \not\rightarrow$. The set of all possible runs in A is denoted by Σ_A.*

The notion of a run can be applied to both, a single service automaton and a composition. For example, $\sigma_1 = p_0\, ?a\, p_1\, \tau\, p_5\, !y\, p_6\, ?b\, p_7$ and $\sigma_2 = p_0\, ?a\, p_1$ are runs of service automaton A in Fig. 1(a), and $\varphi_1 = [p_0, r_0, []]\, !c\, [p_0, r_5, [c]]\, ?c\, [p_8, r_5, []]\, !z [p_9, p_5, [z]]\, ?z\, [p_9, r_6, []]$ and $\varphi_2 = [p_0, r_0, []]\, !b\, [p_0, r_1, [b]]\, !a\, [p_0, r_2, [a, b]]$ are runs in the composition $A \oplus B$ of Fig. 2. Thereby, σ_1 and φ_1 are maximal runs.

For a finite run $\sigma = [q_{0_A}, q_{0_B}, []] \ldots [q'_A, q'_B, \mathcal{B}]$ in a composition $A \oplus B$, we define $msg(\sigma)$ as the function that returns the messages pending in the channels after σ; i.e., $msg(\sigma) = \mathcal{B}$. The pending messages after φ_2 are $msg(\varphi_2) = [a, b]$.

During the interaction between an implementation and a test partner, a run is executed in their composition. But, the part proceeded in the implementation is hidden since we cannot access the implementation's structure in black box testing. Consequently, only the part executed in the test partner can be observed and used for the judgment of correctness. Therefore, we establish the notion of a *projected run*. Assume σ is a run in a composition $A \oplus B$. Then, σ *projected* on A, denoted by $\sigma_{\downarrow A}$, reflects the events and states of A during σ. It can easily be seen that $\sigma_{\downarrow A}$ is a run in A. As an example, consider composition $A \oplus B$ in Fig. 2 and run φ_1, mentioned above. Then $\varphi_{1_{\downarrow A}} = p_0\, ?c\, p_8\, !z\, p_9$ and $\varphi_{1_{\downarrow B}} = r_0\, !c\, r_5\, ?z\, r_6$.

With the help of an observed projected run in a test partner, we are able to make assumptions about the implementation's behavior. Thereby, we exploit that (1) each message received by the test partner has been sent by the implementation and (2) the implementation can only receive messages of the test partner. Whether a sent message is indeed received by the implementation cannot be observed in black box testing.

When judging correctness, not only the observed (projected) runs are considered but also the implementation's execution *status*. Thereby, we distinguish three values: `fi`, for terminating in a final state; `ex`, still in execution (i.e., waiting for a message, being before sending or in deadlock) and `inf`, for infinite runs. The latter is only for theoretical considerations because infinite runs cannot be identified in practise. We consciously do not define separate statuses for "receiving message(s)" and "running internal step(s)", since these statuses are transient and will eventually lead to the status `ex` or `fi`. We assume that it is waited long enough, when querying the implementation's execution status.

The function γ (see Def. 4) returns the statuses that can be reached after a run σ without sending a message. Thereby, the messages pending in the channels after σ can be used to proceed σ. Thus, the status after a run may vary depending on the messages that are already sent by the environment.

Definition 4 (status). *Let A be a* service automaton*, let $\sigma \in \Sigma_A$ and let $\mathcal{B}$ be a multiset of messages. In case σ is a finite run, let it end in state q. The* status *after σ is defined by the function $\gamma_A^{\mathcal{B}} : \Sigma_A \to 2^{\{\texttt{fi},\texttt{ex},\texttt{inf}\}}$ as follows:*

- $\texttt{fi} \in \gamma_A^{\mathcal{B}}(\sigma)$*, iff σ is finite and there is a final state q' reachable from q via internal steps and receiving steps $?a$ with $a \in \mathcal{B}$,*
- $\texttt{ex} \in \gamma_A^{\mathcal{B}}(\sigma)$*, iff σ is finite and there is a state q' reachable from q via internal steps or receiving steps $?a$ with $a \in \mathcal{B}$ such that $q' \xrightarrow{!x}$ or $q' \xrightarrow{?b}$ $(b \notin \mathcal{B})$,*
- $\texttt{inf} \in \gamma_A^{\mathcal{B}}(\sigma)$*, iff σ is infinite.*

For example, $\gamma_B^{\mathcal{B}}(\varphi_{1_{\downarrow B}}) = \{\texttt{fi}\}$ and $\gamma_B^{\mathcal{B}}(\varphi_{2_{\downarrow B}}) = \{\texttt{ex}\}$ regardless of the content of $\mathcal{B}$. Now, we are ready to define *observations about an implementation I*.

Definition 5 (recognizable behavior, observation). *For an implementation I and a test partner P, the* behavior of I recognizable by P *is defined by the set $\mathcal{O}_P^I := \{[\sigma, t] \mid$ there is a run $\varphi \in \Sigma_{I \oplus P}$ with $\varphi_{\downarrow P} = \sigma$ and $t = \gamma_I^{msg(\varphi)}(\varphi_{\downarrow I})\}$. The elements of $\mathcal{O}_P^I$ are* observations *about I recognizable by P.*

As an example, we consider A of Fig. 1(a) as an implementation and B of Fig. 1(c) as a test partner. Two possible observations about A recognizable by B are $\text{obs}_1 = [r_0\,!b\,r_1\,!a\,r_2\,?x\,r_3, \texttt{fi}]$ and $\text{obs}_2 = [r_0\,!b\,r_1\,!a\,r_2\,?y\,r_4, \texttt{fi}]$.

An implementation may own decisions that cannot be influenced by the test partner. This means, a test partner is not able to enforce a certain decision. Consequently, during testing not all observations recognizable by a test partner indeed occur. In the example above, B may induce only obs_1 when sending message b - even though the test is repeated. For this purpose, we distinguish two classes of correctness: For *weak correctness* it is sufficient that any observation made of the implementation can be explained by the specification. For *strong correctness* we additionally claim that every possible observation (regarding a given specification) has indeed occurred during testing.

Definition 6 (correctness of recognizable behavior). *Let I be an implementation and S a specification. The behavior of I recognizable by a partner P*

- *is* weak correct *regarding S iff $\mathcal{O}_P^I \subseteq \mathcal{O}_P^S$, and*
- *is* strong correct *regarding S iff $\mathcal{O}_P^I = \mathcal{O}_P^S$.*

The behavior of I recognizable by a (possibly infinite) set $\mathbb{P}$ of partners

- *is* weak correct *regarding S iff for every $P \in \mathbb{P}$ holds: $\mathcal{O}_P^I \subseteq \mathcal{O}_P^S$, and*
- *is* strong correct *regarding S iff for every $P \in \mathbb{P}$ holds: $\mathcal{O}_P^I = \mathcal{O}_P^S$.*

Definition 6 gives a correctness criterion depending on a given set of partners. In the next section, we define a set of partners that is suitable for conformance testing for a given specification S.

3 Conformance Partner

As mentioned above, a deadlock-free composition is fundamental to guarantee a proper interaction between services. It makes no sense to bind services if their composition may deadlock. Consequently, we can assume that in practise the IUT will only be confronted with partners with which a deadlock-free interaction is ensured a priori. In another context, we already proposed deadlock-freely interacting partners as test cases [5]. These partners communicate deadlock-freely per construction with the IUT (supposed it is implemented correctly). However, if a deadlock occurs during testing, a failure is detected. Deadlock-freely interacting partners can be generated automatically from the formal specification.

But in general, this approach is not applicable for conformance testing without further ado. Due to the abstraction, the formal specification may own non-communicated decisions. Since deadlock-freedom after non-communicated decisions cannot be guaranteed, deadlock-freely interacting partners never cover such decisions and the behavior after them. Consequently, the implementation would be tested insufficiently.

This issue is illustrated by specification S in Fig. 1(e): There is a non-communicated decision in state p_0. Thus, a partner does not know whether S is expecting message a or b. Whatever the partner assumes, its communication with S can deadlock. Even though both messages are sent, deadlock-freedom is not guaranteed. For example, partner P_1 (P_2) in Fig. 3 guesses message a (b) is expected. Consequently, the communication can deadlock in state $[p_2, v_1, a]$ ($[p_1, t_1, b]$). In contrast, P_0 in Fig. 3 sends both, a and b. Here, the composition with S deadlocks in state $[p_7, r_5, a]$ and $[p_7, r_7, b]$ since for deadlock-freedom empty channels are required at termination.

As it can easily be seen, each possible partner can deadlock with S. Thus, no test cases could be generated for S using the existing approach [5]. In general, if non-communicated decisions belong to the model, parts of the implementation are not tested using deadlock-freely interacting partners.

Nevertheless, such specifications make sense. Non-communicated decisions are usually caused by a too coarse abstraction when creating the formal model from an

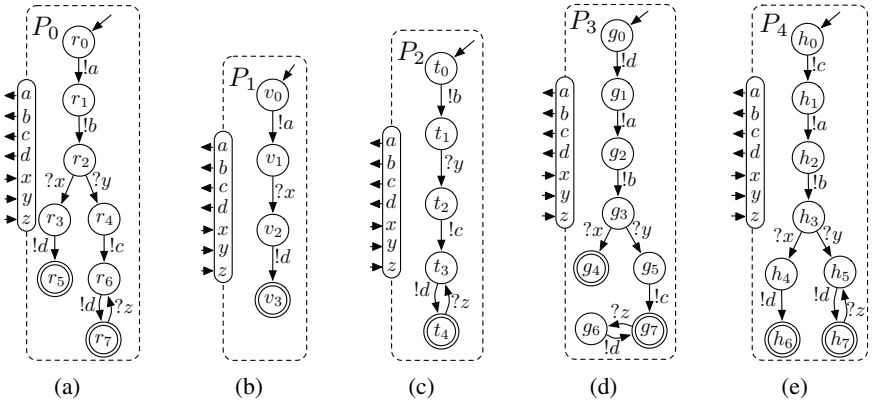


Fig. 3. Partners of the specification S in Fig. 1(e). P_0 and P_3 are conformance partners of S.

(informal) specification. Abstraction is essential to obtain a manageable model. The model in turn is required for automated test case generation. Indeed, non-communicated decisions can be eliminated by refinement. But finding the right level of abstraction is a non-trivial task: If the refinement is too high, the size of the model increases dramatically and makes testing inefficient. Moreover, non-communicated decisions are not as obvious as in the example above. Usually, an extensive analysis is required.

To obviate the problem of refinement, we introduce a new class of partners - the *conformance partners*. With them, non-communicated decisions in the formal specification can be handled and need not be eliminated before test case generation. Thus, thorough testing is possible even though there are non-communicated decisions in the formal specification. In contrast to a deadlock freely interacting partner, a conformance partner is allowed to risk pending messages in the channels at termination time if they are necessary to guarantee the continuation after a non-communicated decision and this non-communicated decision is reached for sure at the point of sending. That means, when a non-communicated decision is inevitable, a conformance partner sends the messages required for all alternatives until it is clear (by receiving a message) how the decision was made. For example, P_0 in Fig. 3(a) is a conformance partner for S in Fig. 1(e). It sends both, a message a and b. Thus, continuation in q_0 is guaranteed, regardless of how S decides. After P_0 has received x or y it knows how the decision was made and behaves appropriately. Since not both alternatives can be executed, either a or b will stay in the channels at termination. That is now permitted thanks to the relaxation of the termination criterion. P_3 is a conformance partner for S, too. It sends message d in advance. That is possible, because d is received independently of the decision and stays in the channel until S reaches state p_5 or p_6 where d is consumed. It is essential to include such partners to test the sending of messages in advance adequately. Similar to P_3, partner P_4 sends message c before it knows which alternative is chosen by S. But in contrast to message d, the receiving of message c is only possible if the right-hand side is chosen. Otherwise, c stays in the channel. That can be avoided, if c is sent after the result of the decision is communicated by message x or y. That is why, c must not stay in the channels at termination. In contrast to a and b, it is not necessary to risk that c is pending in the channels for securing continuation after a non-communicated decision. Thus, we do not classify P_4 as a conformance partner.

These considerations are formalized in the following two definitions. By the notion of *weak receivable* messages we determine whether messages of a channel are allowed to be pending at termination time. In contrast, a channel m is empty after a test run σ if all messages that have been transmitted via m during σ are *strong receivable*. Finally, Def. 8 constitutes the conformance partner.

Definition 7 (weak receivable, strong receivable). *Let A and B be partners. Let σ be a maximal run in B and $\pi = [m, o]$ an outbound pin of B. The messages sent during σ via channel m are* weak receivable *by A iff for each m-event in σ there is a run σ^* with:*

- *σ^* is a prefix of σ containing (at least) the m-event currently considered, and*
- *there is a run φ^* in $A \oplus B$ with $\varphi^*_{\downarrow B} = \sigma^*$ ending in state $[q_A, q_B, \mathcal{B}]$ with $\mathcal{B}(m) = 0$.*

The messages sent during σ via channel m are strong receivable *by A iff after all φ with $\varphi_{\downarrow B} = \sigma$ a state $[q_A, q_B, \mathcal{B}]$ in $A \oplus B$ is reachable such that $\mathcal{B}(m) = 0$.*

As an example: for the run $h_0\,!c\,h_1!a\,h_2\,!b\,h_3\,?x\,h_4\,!d\,h_6$ of P_4 the messages in channel a and d are strong receivable by S, the messages in channel b are weak receivable and the messages in channel c are neither strong receivable nor weak receivable. For the infinite run $h_0\,!c\,h_1!a\,h_2\,!b\,h_3\,?y\,h_5\,!d\,h_7\,?zh_5\,!d\,h_7$ ... of P_4 the messages in channel b, c and d are strong receivable by S and the messages in channel a are weak receivable.

Definition 8 (conformance partner). *Let A and B be partners (A is free of deadlocks). B is a* conformance partner *of A iff*

1. *for all maximal runs σ in B and all m with $[m, \mathbf{o}] \in \Pi_B$ holds: The messages sent during σ via channel m are strong or weak receivable by A,*
2. *for all maximal runs ψ in A and all n with $[n, \mathbf{o}] \in \Pi_A$ holds: The messages sent during ψ via channel n are strong receivable by B,*
3. *for every state $[q_A, q_B, \mathcal{B}]$ in $A \oplus B$ with $[q_A, q_B, \mathcal{B}] \not\rightarrow$ holds: $q_A \in \Omega_A$ and $q_B \in \Omega_B$.*

The set of all conformance partner of A is denoted by Conf(A).

Note, if B is a conformance partner for A then A is not necessarily a conformance partner for B. This asymmetry can be easily derived from the definition by comparing the requirement (1) and (2).

A run σ in a service automaton A is *covered* by a partner B iff there is a run φ in $A \oplus B$ such that $\varphi_{\downarrow A} = \sigma$. We then also say, σ is covered by $\varphi_{\downarrow B}$. Finally, we constitute in Thm. 1 that a specification is fully covered by conformance partners.

Theorem 1. *Every run σ in a service automaton A is* covered *by (at least) one conformance partner of A.*

Proof: Assuming $\sigma \in \Sigma_A$ is the shortest path not covered by any conformance partner; i.e., $\sigma = \sigma^* e q'$ with σ^* is still covered. Consequently, there is a partner $P \in Conf(A)$ such that there is a run $\varphi \in A \oplus P$ with $\varphi_{\downarrow P} \in \Sigma_P$ and $\varphi_{\downarrow A} = \sigma^*$. If e is an internal event or a sending event then σ is also covered by $\varphi_{\downarrow P}$. If e is a receiving event and $e \in msg(\varphi)$ then σ is also covered by $\varphi_{\downarrow P}$. Finally, if e is a receiving event and possibly $e \notin msg(\varphi)$ then we can extend P to a partner P' such that P' sends a message e after σ^* is covered. This does not violate Def. 8 since e is obviously receivable by A. Let ψ be the run in P' that covers σ. As it can be easily seen there is a continuation for ψ such that P' does not violate the requirements of Def. 8. Thus, σ is covered by the conformance partner P'. *q.e.d.*

By requirement (1) in Def. 8 we ensure that the sending of messages in advance is adequately considered by the conformance partners - even if non-communicated decisions are covered. Requirement (2) guarantees that no messages from the specification are pending in the channels at termination time. Thus, specified messages are not ignored by a conformance partner. That is essential for the evaluation of a test run. Requirement (3) ensures, that a finite interaction with a conformance partner always terminates in a final state - if the implementation acts correctly. Based on these considerations together with Thm. 1 we can define *conformance* as follows:

Definition 9 (conformance). *Let S be a specification and I be an implementation. I is* weak (strong) conform *to S iff for every $P \in Conf(S)$ holds: $\mathcal{O}_P^I \subseteq \mathcal{O}_P^S$ ($\mathcal{O}_P^I = \mathcal{O}_P^S$).*

As already mentioned, each channel is able to buffer only a limited number of k messages. For this reason, we restrict ourself to specifications whose composition with each existing conformance partner is k-bounded. Other specifications are not reasonable since the channels can overflow and then the following behavior is not defined.

4 Test Case Generation

As stated in Def. 9, we propose conformance partners as test cases. In this section, we describe how conformance partners are derived systematically from a formal specification S. Therefore, we create a representation - we call it *test guidelines* - that characterizes the set of *all* conformance partners of S. Its construction follows the ideas of the operating guidelines [2] that characterize deadlock-freely interacting partners.

Using all conformance partners as test cases would be too time consuming. Thus, we show in the second part of this section, how a limited number of partners can be extracted from the test guidelines. The resulting test suite is exhaustive in the sense that it is still able to detect each failure that could be discovered by any conformance partner.

Synthesizing Conformance Partners. To generate the test guidelines for a specification S, we first construct a partner that overapproximates the behavior of the conformance partners. Intuitively, a service B has *more behavior* than another service A if it is able to imitate A's behavior such that the environment cannot observe differences between the two services. We formalize this property by the notion of *weak simulation*.

Definition 10 (weak simulation). *Let A and B be service automata. A relation $\varrho \subseteq Q_A \times Q_B$ is a* weak simulation *relation iff*

- $[q_{0_A}, q_{0_B}] \in \varrho$,
- *for all $q_A, q'_A \in Q_A$, $q_B \in Q_B$, and $m \in \mathbb{E}_A^\Pi$ holds: if $[q_A, q_B] \in \varrho$ and $q_A \xrightarrow{m} q'_A$, then there exists a state $q'_B \in Q_B$ with $q_B \xrightarrow{m} q'_B$ and $[q'_A, q'_B] \in \varrho$,*
- *for all $q_A, q'_A \in Q_A$, $q_B \in Q_B$ and $e \in \mathbb{E}_A^\tau$ holds: if $[q_A, q_B] \in \varrho$ and $q_A \xrightarrow{e} q'_A$, then $[q'_A, q_B] \in \varrho$,*
- *for all $q_A, q'_A \in Q_A$, $q_B \in Q_B$ and $e \in \mathbb{E}_B^\tau$ holds: if $[q_A, q_B] \in \varrho$ and $q_B \xrightarrow{e} q'_B$, then $[q_A, q'_B] \in \varrho$,*
- *$[q_A, q_B] \in \varrho$ implies that $q_A \in \Omega_A$ iff $q_B \in \Omega_B$.*

B weakly simulates A *iff there exists a weak simulation relation $\varrho \subseteq Q_A \times Q_B$.*

As an example, the service automaton depicted in Fig. 4(a) weakly simulates (or has more behavior than) P_0, P_1, P_2 and P_3 of Fig 3. The weak simulation relation with P_1 is $\varrho = \{[v_0, q_0], [v_1, q_2], [v_2, q_9], [v_3, q_{19}]\}$.

To synthesize a partner that has more behavior than *any* conformance partner, we consider specification S and the message channels as a black box. That means, their state is only estimated by considering the communication with the partner. Usually, there are several possibilities. They are calculated by the $closure$.

Definition 11 (situation, closure [2,3]). *Let $S = [Q, q_0, \Omega, \Pi, \mathbb{E}^\tau, \delta]$ be a specification and $X \subseteq (Q \times Bags(\mathbb{M}))$. The elements of X are called* situations*. The set $closure_S(X)$ is the smallest set satisfying:*

1. $X \subseteq closure_S(X)$.
2. *If* $[q, \mathcal{B}] \in closure_S(X)$ *and* $q \xrightarrow{e} q'$ *with* $e \in \mathbb{E}^\tau$, *then* $[q', \mathcal{B}] \in closure_S(X)$.
3. *If* $[q, \mathcal{B}] \in closure_S(X)$ *and* $q \xrightarrow{!m} q'$ *with* $!m \in \mathbb{E}^\Pi$, *then* $[q', \mathcal{B} + [m]] \in closure_S(X)$.
4. *If* $[q, \mathcal{B} + [m]] \in closure_S(X)$ *and* $q \xrightarrow{?m} q'$ *with* $?m \in \mathbb{E}^\Pi$, *then* $[q', \mathcal{B}] \in closure_S(X)$.

For a set X of situations, $closure_S(X)$ comprises all situations that can be reached by S without the help of the partner; i.e., by doing internal steps, sending messages or receiving messages already pending in the channels. For example, for S in Fig. 1(e) and $X = \{[p_0, [a]]\}$ we obtain $closure_S(X) = \{[p_0, [a]], [p_1, [a]], [p_2, [a]], [p_3, [\,]], [p_5, x]\}$.

In the following, we define the partner $TS^0(S)$ that has more behavior than any conformance partner of S. Each state q of $TS^0(S)$ consists of those situations that are possible after the events leading to q have occurred.

Definition 12 (conformance partner overapproximation). *Let $S = [Q_S, q_{0_S}, \Omega_S, \Pi_S, \mathbb{E}^\tau_S, \delta_S]$ be a specification. We define the service automaton $TS^0(S) = [Q, q_0, \Omega, \Pi, \mathbb{E}^\tau, \delta]$ with $\Pi = \overline{\Pi_S}$, $E^\tau = \emptyset$, and Q, q_0 and δ inductively as follows:*

base: $q_0 := closure_S(\{[q_{0_S}, [\,]] \mid q_{0_S} \in Q_S\})$ *and* $q_0 \in Q$.
step: For all $q \in Q$ *and* $m \in \mathbb{M}$*:*

1. *There is a transition* $q \xrightarrow{!m} q'$ *and* $q' \in Q$ *iff* $!m \in \mathbb{E}^\Pi$ *and* $q' := closure_S(\{[q_S, \mathcal{B} + [m]] \mid [q_S, \mathcal{B}] \in q\})$ *and* $[q^*, \mathcal{B}] \in q'$ *implies* $\mathcal{B}(m) \leq k$.
2. *There is a transition* $q \xrightarrow{?m} q'$ *and* $q' \in Q$ *iff* $?m \in \mathbb{E}^\Pi$ *and* $q' := closure_S(\{[q_S, \mathcal{B}] \mid [q_S, \mathcal{B} + [m]] \in q\})$.
3. $q \in \Omega$ *iff* *there is* $[q_S, \mathcal{B}] \in q$ *with* $q_S \in \Omega_S$ *and* $m \in \mathcal{B}$ *implies there is* $[m, \mathbf{i}] \in \Pi_S$.

A state q in $TS^0(S)$ has (1) a leaving edge labeled with a sending event $!m$ iff the given message-bound k is not exceeded by the sending of message m and (2) a leaving edge for any receiving event $?m$ (possibly the closure of the successor state q' is empty). Further, state q is defined as a final state (3) iff it contains a situation where S is in a final state and the input channels of $TS^0(S)$ are empty. That means, if $TS^0(S)$ reaches q, S could be terminated and no more messages can be received by $TS^0(S)$.

$TS^0(S)$ overapproximates the behavior of any conformance partner.This can be directly proofed by comparing the conditions of Def. 12 and Def. 8 (conformance partner). To fulfill condition (1) in Def. 12, $TS^0(S)$ sends in every state all possible messages $m \in \mathbb{M}$ (as long as the message-bound is not exceeded). In contrast, a conformance partner is more restricted: Its sent messages must be strong or weak receivable by S (see (1) in Def. 8). Caused by condition (2) in Def. 12, $TS^0(S)$ is *input complete*; i.e., every state has a leaving edge for any receive event. Thus, a conformance partner cannot

take more receive events into account. Finally, the last conditions of Def. 12 and Def. 8 coincide each other. Consequently, $TS^0(S)$ weakly simulates any each conformance partner of S.

To actually construct the test guidelines, we now create a conformance partner from $TS^0(S)$. Currently, $TS^0(S)$ only violates condition (1) of Def. 8 whereas condition (2) and (3) are already fulfilled (see the argumentation above). The behavior of $TS^0(S)$ is complete in the sense that due to the message bound no more edges can be added. Thus, edges must be removed to obtain a conformance partner of S. Since only condition (1) of Def. 8 is violated, $TS^0(S)$ simply owns some send events that are forbidden for conformance partners. They can be identified by the following *labeling function*. Thereby, a *strongly connected component* (SCC) is a maximal set of mutually reachable states, and a *terminal* strongly connected component (TSCC) is an SCC with no leaving eges.

Definition 13 (labeling function). *For $TS^i(S)$ $(i = \{0, 1, \ldots\})$ of a service automaton S we define a labeling function l that assigns to each SCC C in $TS^i(S)$ a set $\mathcal{M} \subseteq Bags(\mathbb{M})$ $(l(C) = \mathcal{M})$ such that $\mathcal{M}$ is maximal according to the following two conditions:*

1. *for each state $q \in C$ and each $[p, \mathcal{B}] \in q$: $\mathcal{M} \leq \mathcal{B}$, and*
2. *for all messages m: $\mathcal{M}(m) \leq max\{l(C')(m) \mid C' \text{ is successor of } C\}$.*

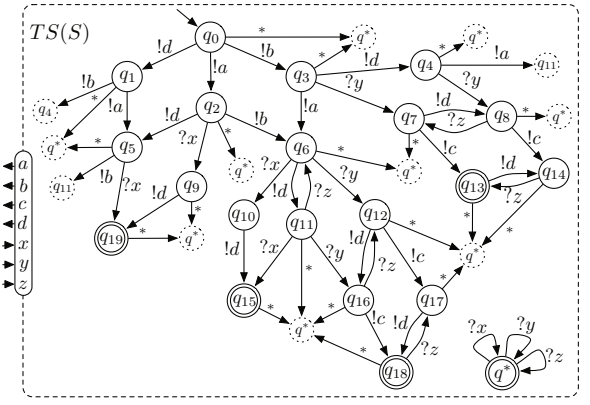
We use the labeling function to analyze which messages sent by $TS^i(S)$ are neither weak nor strong receivable by S. Then, the corresponding edges have to be eliminated to fulfill condition (1) of Def. 8. Thereby, condition (2) and (3) remain fulfilled since only edges labeled with send events are removed. In particular, input completeness is preserved. The procedure is formalized in Def. 14.

Definition 14 (conformance partner synthesis). *Given $TS^i(S)$ $(i \geq 0)$, the service automaton $TS^{i+1}(S)$ is obtained by removing an arc labeled with $!m$ and leading to an SCC C with $l(C)(m) > 0$. Thereby, the removal of an arc includes the removal of all states that become unreachable from the initial state q_0.*

Let $TS(S)$ be $TS^j(S)$ for the smallest j with $TS^j(S) = TS^{j+1}(S)$.

Fig. 4(a) shows $TS(S)$ of the specification S in Fig. 1(e). From the argumentation above follows directly that $TS(S)$ is a conformance partner of S *and* it is the one with the most behavior; i.e., $TS(S)$ weakly simulates any other conformance partner of S. But, not every weakly simulated service automaton is a conformance partner. For example, P_1 and P_2 are simulated by $TS(S)$ in Fig. 4(a). But they are not conformance partners of S as they violate condition (3) of Def. 8. For example, in the composition $S \oplus P_1$, state $[p_2, v_1, [a]]$ has no successors, but neither p_2 nor v_1 is a final state.

To determine whether a weakly simulated automaton P is really a conformance partner, we annotate $TS(S)$ with Boolean formulae. They express which states and transitions of $TS(S)$ must be present in the simulation relation and which parts are allowed to be absent. By the resulting *annotated service automaton* the set of all conformance partners of S is characterized. We call it the *test guidelines* of specification S ($TG(S)$ for short) as it represents all eligible test cases. Due to page limit, we omit the details of



(a)

$\phi(q_0) = !a \vee !b \vee !d$
$\phi(q_1) = !a \vee !b$
$\phi(q_2) = !b \vee !d$
$\phi(q_3) = !a \vee !d$
$\phi(q_4) = !a$
$\phi(q_5) = !b$
$\phi(q_6) = ?x \wedge ?y \vee !d$
$\phi(q_7) = !c \vee !d$
$\phi(q_8) = !c$
$\phi(q_9) = !d$
$\phi(q_{10}) = !d$
$\phi(q_{11}) = ?x \wedge ?y$
$\phi(q_{12}) = !d \vee !c$
$\phi(q_{13}) = !d$
$\phi(q_{14}) = ?z$
$\phi(q_{15}) = true$
$\phi(q_{16}) = !c$
$\phi(q_{17}) = !d$
$\phi(q_{18}) = ?z$
$\phi(q_{19}) = true$
$\phi(q^*) = true$

(b)

Fig. 4. Test guidelines $TG(S)$ of specification S in Fig. 1(e) consisting of (a) conformance partner $TS(S)$ and (b) Boolean formulae. In the graphical representation, dotted states are placeholders for states with the same number. $TS(S)$ is input complete. For not depicted receiving events there is implicitly an edge to state q^*, indicated by an edge labeled with "*". The $closure$ of some states is as follows: $q_0 = \{[p_0, []], [p_1, []], [p_2, []]\}$, $q_1 = \{[p_0, [a]], [p_1, [a]], [p_2, [a]], [p_3, []], [p_5, x]\}$ and $q^* = \emptyset$.

the construction of the Boolean formulae. It follows exactly the ideas of generating operating guidelines [2]. There, Boolean formulae are used to characterize the set of the deadlock-freely interacting partners for a given service.

Finally, a service automaton P is a conformance partner for S iff it is weakly simulated by $TG(S)$ *and* in every state pair $[q_P, q_{TG(S)}]$ of the weak simulation relation state q_P evaluates the boolean formula of $q_{TG(S)}$ to true. The latter means, $\phi(q_{TG(S)})$ is satisfied in the assignment β_{q_P} where $\beta_{q_P}(e) = true$ iff there exists $q'_P \in Q_P$ with $[q_P, e, q'_P] \in \delta_P$.

As an illustrating example, Fig. 4(b) shows the Boolean formulae for $TS(S)$ of the specification S in Fig. 1(e). The formula $\phi(q_0) = !a \vee !b \vee !c$ of the initial state q_0 determines that a conformance partner must have a leaving edge labeled with $!a$, $!b$ or $!c$ (or combinations thereof) at the beginning. Due to the structure of $TS(S)$ a conformance partner is also allowed to have additionally a leaving edge labeled with $?x$, $?y$ or $?z$ (or combinations thereof).

As already mentioned, $[v_0, q_0]$ and $[v_1, q_2]$ are elements of the weak simulation relation between P_1 and $TS(S)$. State v_0 has a leaving edge labeled with $!a$. Thus, v_0 evaluates the formula $!a \vee !b \vee !c$ of q_0 to true. In contrast, v_1 does not fulfill the formula $!b \vee !d$ of state q_2 as it has neither a leaving edge labeled with $!b$ nor a leaving edge labeled with $!d$. Thus, P_1 is not characterized by $TG(S)$. This coincides with the statement above where P_1 was identified as non-conformance partner of S using Def. 8.

Note, the literals in the Boolean formulae of the test guidelines are not always connected by disjunctions. Conjunctions are also possible. For example, the Boolean formula $\phi(q_{11}) = ?x \wedge ?y$ of $TS(S)$ in Fig. 4(a) indicates that a conformance partner must provide both, a leaving edge labeled with $?x$ *and* a leaving edge labeled with $?y$ if the

weak simulation relation "touches" state q_{11}. A conjunction in a Boolean formula is caused by a decision of the specification that cannot be influenced by the partner. Thus, a conformance partner is required to deal with the messages of all possible alternatives to avoid deadlocks in the interaction. In the example, a partner cannot influence whether S in Fig. 1(e) sends x or y.

We are aware that the complexity of the generation of $TG(S)$ is more or less moderate. But due to the strong links, it is comparable with the construction of operating guidelines. With the tool Wendy[1] [6] it was shown that operating guidelines can be calculated in a reasonable time.

Test Case Selection. The test guidelines $TG(S)$ of a specification S characterize all conformance partners of S. Thus, $TG(S)$ can already be seen as a complete test suite for S. However, there are some redundant partners which can be left out without reducing the quality of the test suite.

The behavior of the IUT is evaluated based on the observations made during testing. Consequently, to imitate each conformance partner, we only need to ensure that each observation that is possible by any conformance partner is also possible by one partner of the test suite. This is already fulfilled when selecting the conformance partner $TS(S)$ exclusively as it weakly simulates any conformance partner of S. But $TS(S)$ contains many branches. Thus, it needs to be executed repeatedly for thorough testing. To enforce the different paths, manual adjustments are required before every execution. This is not desirable in automated testing. Instead, we split $TS(S)$ into several small partners $T_1, ..., T_n$ such that

(1) each T_i is a conformance partner,
(2) each run of $TS(S)$ is also considered by (at least) one T_i,
(3) each T_i contains a run that is not considered by any T_j ($i \neq j$), and
(4) each T_i contains as less branches as possible.

Condition (1) and (2) guarantee that the resulting test suite contains only sound test cases (i.e., conformance partners) and is exhaustive regarding Def. 9. By Condition (3), we exclude redundancies among the test cases in the sense that each test partner could make an observation that cannot be made by any other test partner in the test suite. Definition (4) ensures that test partners do not need to be adjusted before execution. Note, to preserve the conformance partner properties not all branches can be eliminated by splitting. However, the remaining branches do not require adjustments. Basically, they are caused by decisions of the implementation that cannot be influenced by the test partner. Thus, a certain alternative cannot be enforced during testing and manually adjustments can be omitted.

The generation of the test suite can be realized by one depth-first search through the test guidelines. Basically, the splitting of the branches is triggered by the Boolean formulae. Due to page limit, we sketch the procedure by the running example only. Consider the test guidelines in Fig. 4(a). The formula "$!a \vee !b \vee !d$" of the initial state demands that any conformance partner has to start with an $!a$, $!b$ or $!d$ event - or combinations thereof. Thereby, the partners containing more than one action can choose which message to send on (test) runtime. In such a case, we can move this runtime decision to the

[1] `http://service-technology.org/wendy`

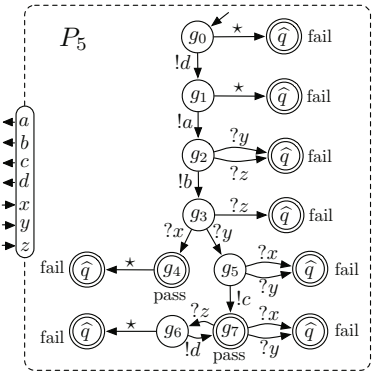


Fig. 5. A test partner synthesized from the test guidelines in Fig. 4. For better readability state $\widehat{q}$ is depicted multiply. An edge labeled with "$\star$" is a placeholder for all receiving events of P_5.

design time of the test partners and only consider those partners consisting of one of the respective events. Consequently, we can split and have three kinds of test cases: either starting with $!a$ or $!b$ or $!d$. By following the $!d$ edge, we reach a state related to the formula "$!a \vee !b$". Thus, we can refine the last kind of test partners to those which either start with $!d$ followed by $!a$ or start with $!d$ followed by $!b$. With the remaining states we proceed in the same manner. Note, we do not split conjunctions. Otherwise, the conformance partner property is not preserved. As an example, see again conformance partner P_3 in Fig. 3(d).

State q^* in $TS(S)$ is a special state as its closure is the empty set (cf. Fig.4(a)). That is, a conformance partner P may contain a state $\widehat{q}$ that is related to q^* in the weak simulation relation, but the specification cannot cover transitions in P leading to $\widehat{q}$. If, however, such a state is reached during testing, the implementation has sent an unspecified message. To observe the implementation best possible, we add such a state $\widehat{q}$ to the test partners and equip each "regular" state with all possible transitions leading to $\widehat{q}$. As soon as $\widehat{q}$ is reached during testing, a failure is detected and we can stop the test run. Thus, we classify $\widehat{q}$ as a final state. During testing, in each state, a test partner tries to execute a transition to $\widehat{q}$ first. If no unspecified message could be received it continues with a "regular" transition. As an example, Fig. 5 depicts a test partner. It is the conformance partner P_3 of Fig. 3(d) enriched with state $\widehat{q}$ and the respective transitions to discover unspecified messages.

To distinguish the intended behavior from unspecified messages easily, we introduce the labels `pass` and `fail` into the test partner. State $\widehat{q}$ is labeled with `fail`. The other final states q_Ω indicate that the specification could reach a final state if a test partner is in q_Ω. Consequently, we label each q_Ω with `pass`. These are the only states where the implementation is allowed to terminate.

For testing conformance we execute the selected partners one after another together with the implementation. A failure is detected if during a test run the implementation behaves unforeseen in any point of time; i.e., a test partner reaches a state labeled with `fail`, the implementation terminates but the test partner is not in a `pass` state, or the

interaction deadlocks (i.e., the test partner gets stuck in a state not labeled with `pass`). For weak conformance it is sufficient to execute each test partner once. In contrast, to establish strong conformance, the test partners are executed repeatedly until every expected observation is recognized. The implementation passes the test suite successfully, if every test run completes with `pass`. On success, we can be confident that it will interact correctly in practice as well.

5 Related Work

There exists a variety of approaches for testing services. A detailed overview is given by Bozkurt et al. [7] and Baresi and Nitto [8]. Test case generation for *synchronous* communicating components was studied in detail by Tretmans (e.g., [9,1]). But due to the different characteristics of message passing, these approaches are not applicable for our asynchronous setting. Others consider asynchronous communication, but with limitations: The approaches of Dranidis et al. [10] and Keum et al. [11] are restricted to services with a communication exclusively consisting of request-response pairs; that is, the sending of a message is directly followed by receiving a message. In our approach, we are more liberal; that is, messages can be sent and received in an arbitrary order. Other approaches (e.g., [12,13,9]) assume asynchronous message passing through an input and an output queue. Here, the sending and receiving is independent, but the messages cannot overtake each other during their transmission. Consequently, these approaches are also not applicable for our setting. To our knowledge, there exists no related work that considers the overtaking of messages adequately when generating test cases for asynchronous communicating components.

6 Conclusion

In this paper, we presented a black-box testing approach for stateful asynchronously communicating services. We formalized correct (asynchronous) communication and introduced conformance partners as test cases. Further, we studied how a limited number of conformance partners can be selected such that the resulting test suite is still complete. That way, we avoid testing with all possible conformance partners without reducing the quality of the test suite.

The introduced theory is independently from a specific language since we use automata as underlying formalism. Thus, the presented test case generation approach is not restricted to services, but also applicable for asynchronous communicating components in other domains; e.g., components of reactive systems or participants in telecommunication systems.

References

1. Tretmans, G.J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer networks and ISDN systems 29, 49–79 (1996)
2. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)

3. Lohmann, N.: Correctness of services and their composition. PhD thesis, Universität Rostock / Technische Universiteit Eindhoven, Rostock, Germany / Eindhoven, The Netherlands (2010)
4. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2 (1989)
5. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 66–78. Springer, Heidelberg (2009)
6. Lohmann, N., Weinberg, D.: Wendy: A tool to synthesize partners for services. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 297–307. Springer, Heidelberg (2010)
7. Bozkurt, M., Harman, M., Hassoun, Y.: Testing web services: A survey. Technical report (2010)
8. Baresi, L., Nitto, E.D. (eds.): Test and Analysis of Web Services. Springer, Heidelberg (2007)
9. Tretmans, G.J.: A Formal Approach to Conformance Testing. PhD thesis, University of Twente, Enschede (1992)
10. Dranidis, D., Kourtesis, D., Ramollari, E.: Formal verification of web service behavioural conformance through testing (2007)
11. Keum, C., Kang, S., Ko, I.-Y., Baik, J., Choi, Y.-I.: Generating Test Cases for Web Services Using Extended Finite State Machine. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 103–117. Springer, Heidelberg (2006)
12. Weiglhofer, M., Wotawa, F.: Asynchronous input-output conformance testing. In: COMPSAC (1), pp. 154–159 (2009)
13. Simao, A., Petrenko, A.: From test purposes to asynchronous test cases. In: ICSTW 2010. IEEE Computer Society (2010)