SOCA (2014) 8:299–322
DOI 10.1007/s11761-013-0146-2

SPECIAL ISSUE PAPER

On expanding protocol conformance checking to exception handling

Christian Heike · Wolf Zimmermann · Andreas Both

Received: 7 January 2013 / Revised: 16 October 2013 / Accepted: 4 November 2013 / Published online: 30 November 2013

Abstract Components or services must often be compliant to organizatorial or legal regulations. Furthermore, they should avoid unwanted behaviour such as abortion of the execution of a service without notification of the client. Violation of both might happen due to unintended uses of services. In general, the intention is specified by contracts. In this work, we consider a special form of contracts: service protocols. These specify for a service legal sequences of operation calls. We propose an approach for checking whether such protocols are obeyed in a service composition. For this, it is necessary to define a conservative abstraction of the behaviour of service-oriented systems and a contract based on interactions (named service protocol) to be verified. In our previous work, we have modelled unbound concurrency, unbound recursion, and synchronization. This article briefly presents the previous results and extends them by exception handling mechanisms. In particular, it takes into account that the execution of service may raise an exception and allows the clients to react on the exception.

Keywords Protocol conformance checking · Exceptions · Process rewrite systems · Model checking · Web services

C. Heike
Zuehlke Engineering AG, Wiesenstrasse 10a,
8952 Schlieren, Switzerland
e-mail: christian.heike@zuehlke.com

W. Zimmermann (✉)
Institut für Informatik, Universität Halle-Wittenberg,
06099 Halle/Saale, Germany
e-mail: wolf.zimmermann@informatik.uni-halle.de

A. Both
Research and Development, Unister GmbH,
Barfußgässchen 11, 04109 Leipzig, Germany
e-mail: andreas.both@unister.de

1 Introduction

Modern software development contains a big share of reusing previously developed software called services. Often these services are developed by third party companies and supplied as Web Service. Our work addresses the topic composability analysis for replaceability, compatibility, and process conformance, which was identified as a research challenge in [29].

While *stateless* services have no restrictions on the order of the call of operations of the interface, *stateful* services may restrict this order. For example, a file service may expect that a file is first opened for reading, then read operations may follow, and finally the file must be closed.

Such a required client behaviour needs to be obeyed during the execution of the service. If the client differs the service usage from the provided service protocol, the application might crash and cause a chain of events leading to unwished behaviour. In the worst-case scenario a problem within one service is appearing only for the current specific configuration may lead to the crash of the complete component-based software or service-oriented system.

Hence, our goal is to check automatically whether in a service composition each stateful service is used correctly. The correct usage of a stateful service must be specified by a service protocol and should be published together with the service interface. Therefore, it must be proven that the behaviour of clients of a service never violate the defined service protocol. A violation of a service protocol means that its operations are called not compliant to the protocol. Usually, these protocols for stateful services are specified as a finite state machine.

An automatic verification approach must consider the sequences of operation calls to a stateful services. Therefore, for a service composition, it must be checked whether the set of sequences of possible operation calls to a stateful service is a subset of the set of legal operation calls specified by the protocol. This kind of condition is called *trace inclusion*.

Remark 1 Note that in the literature on component-based and service-oriented architectures, there are also other kind of protocols, the interaction protocols, see e.g. [9]. *Interaction protocols* specify the behaviour of a service by request and reply messages, i.e., an abstraction service interaction interface is modelled. Thus, their primary focus is to model the receivable as well as the triggered interactions of a service. Interaction protocol checking examines whether for each request there is a reply message and vice versa, i.e., a bisimulation relation must be satisfied.

Service protocols consider a service interface as an application programming interface (API), i.e., the operations specified in an interface description are atomic units. Their primary focus is to model the set of legal sequences of operations calls. Service protocol conformance checking checks whether all sequences of operation calls to a service are legal, i.e. trace inclusion (or alternatively, a simulation relation) must be satisfied.

Service protocol conformance checking requires to know the behaviour of services. Often, service providers do not want to provide their code, business process etc., of the service's implementation. Thus, abstractions of the behaviour of the services are necessary, i.e., each real behaviour corresponds to an abstract behaviour but not necessarily vice versa. These abstractions should be automatically be derived and published together with the service description. Therefore, the basis for this abstraction is the source code of the service implementation, usually written in higher-level programming languages as e.g. *Java*, *C#* etc. Therefore, the important language concepts have to be considered since everything possibility in a programming language is used—whether it makes sense in a certain context or not. In our previous work [3,5], we modelled abstractions for the classical control structures such as loops, conditionals, sequential execution of statements etc. In particular, we considered recursion and concurrency without any restrictions on recursion depth or the number of concurrent threads. In this article, we consider in addition exception handling concepts.

Currently, abstractions are often specified using Petri-Nets (see e.g. [34,39]), pushdown systems (see e.g. [10,11]), finite state machines (see e.g. [28,38]), or process-algebras (see e.g. [9]). Finite state machines only allow an adequate modelling of bound concurrency and bound recursion. This means that the number of parallel threads and the recursion depth are bound by a constant, respectively. If recursion is present within the component or the application, the language of interactions is not regular but context-free [40]. Therefore, it requires a pushdown automaton to describe all sequences of interactions, i.e., finite state machine or those process-algebras not taking into account sequential recursion cannot model it. Furthermore, if recursive callbacks are present, protocol conformance checking based on finite state machine abstractions may lead to false positives [40]. Unbound recursion can be adequately modelled by pushdown systems, but there is no adequate modelling of unbound concurrency. Petri-Nets may model adequately unbound concurrency but not unbound recursion. However, recursive Petri-nets may deal (cf., [20]) with unbound concurrency and unbound recursion. With some process algebras, it is possible to model unbound recursion and unbound parallelism including synchronization. This leads to a model equivalent to a Turing Machine [20] (or a coloured Petri-Net with an infinite number of colours) and is therefore not well-suited for analysis tools. In order to provide safe protocol conformance checking, a conservative representation of the actual component behaviour is required. Consequently the abstraction layer has to be capable of representing unbound recursion, unbound parallelism as well as exception handling, such that it can still be checked.

Protocol Conformance Checking verifies the protocol w.r.t. these abstractions. Protocol conformance implies that there is no protocol violation. However, it is possible that protocol conformance cannot be proven although there is no protocol violation in the real behaviour (*false alarms*). Furthermore, service may call operations of other services. Hence, protocol violations may occur in services not directly used by the client.

In order to take into account indirect uses of services, it is necessary to consider the behaviour of services. On the one hand, services implementations fully specify this behaviour. On the other hand, services providers may wish to keep secret their implementation, e.g., because it may contain business secrets. Thus, an abstract behaviour of services that hide implementation decisions and business secrets should be published. A protocol conformance checker composes the abstract behaviour of each service according to the architecture of the system such that it keeps track of the calls to operations provided by all services [5].

If this abstraction is too coarse-grained, a large number of false alarms may be produced. In our work, the abstract behaviour completely abstracts from data, but under this restriction the control flow should be modelled precisely. In particular, if recursive callbacks are present in the composed application it should be present in the abstraction, otherwise some protocol violations might not be discovered [40]. This approach follows the idea that everything that is possible should be considered. Often, it is argued that recursive callbacks are not present in the service-oriented context. However, there are candidates for recursive callbacks such as for example a map-reduce service [1].

It is not necessary to specify manually the abstract behaviour of a service implementation, since it can be automatically derived from the source code of the implementation using standard compiler technology [3,5].

Both and Zimmermann [5] show that using Mayr's process rewrite systems [26], both, recursion and parallelism, can be adequately modelled. However, the protocol conformance checking problem becomes undecidable. An approximation for this protocol conformance checking problem is shown in [5] and its feasibility was demonstrated. None of these works consider exception handling. However, in modern programming languages interactions can also be initiated by exceptions. Hence, their characteristic behaviour triggered by conditional execution of blocks as well as the execution of the finally-block might lead to service protocol violations. In particular, in the case of chains of interface calls, a protocol violation might be raised somewhere in the component-based system at the direct interfaces of the component raising the exception.

Hence, simplified abstractions are not acceptable for application in component systems. Therefore, it is important to tackle these exceptions to ensure a rugged composition of services. Our main contributions are:

(i) showing that the previous concepts cannot deal with exceptions adequately
(ii) providing an (automatic) abstraction of exception handling that can be combined with abstractions of other programming language concepts such as procedure call and return, forking and synchronizing parallel processes, loops, conditional statements, statement sequences,
(iii) and showing an approach for protocol conformance checking based on this abstraction.

Section 2 defines process rewrite systems, protocols and provides a running example. In Sect. 3, the abstraction of exception semantics to process rewrite systems is demonstrated. Section 4 shows how to check protocol conformance. Section 5 discusses related work.

2 Preliminaries

This section introduces our service model, its execution semantics including exception handling, and summarizes [3–5]. In particular, it is shown that the execution semantics naturally corresponds to Mayr's process rewrite systems [26].

A service s *provides* an interface I_s where an interface is a set of type descriptions and procedure signatures with exceptions that may be raised during execution. The implementation of s may call procedures of other services. The *required* interface R_s of s is the set of procedures of other services called by s. A service-oriented system is a directed graph $S \triangleq (WS, C)$ where WS is a set of services such that each service $s \in WS$, $p \in R_s$ there is an edge $(s, s') \in C$ with $p \in I_{s'}$. Hence, any call leaving existing service $s \in WS$ calls a procedure of another service $s' \in WS$. Consequently, if a service implementation might lead to a call of another procedure, it is represented within S.
There are two kinds of procedures in interfaces, *asynchronous* and *synchronous* procedures. If a synchronous procedure is called, the caller waits until the callee is completed. If an asynchronous procedure is called, the caller and the callee concurrently continue their execution. A synchronize statement **sync** f is a barrier, i.e. the execution waits until the last asynchronous call of f is completed. Before a (synchronous or asynchronous) procedure p returns, all asynchronous procedures called by p must be completed.

For the implementation of the services, programming languages such as e.g. *Java*, *C#*, or *BPEL* can be used. For the purpose of this article, the complete consideration of all programming language concepts would be too much. Instead we consider the most important concepts such as loops (with the classical semantics of while-loops), conditionals, sequential execution of statements, a synchronization statement, synchronous and asynchronous procedures, and exception handling. Loops, conditional statements, and sequential execution of statements have the standard semantics. Synchronous and asynchronous procedure calls and the synchronization have a semantics as described above.

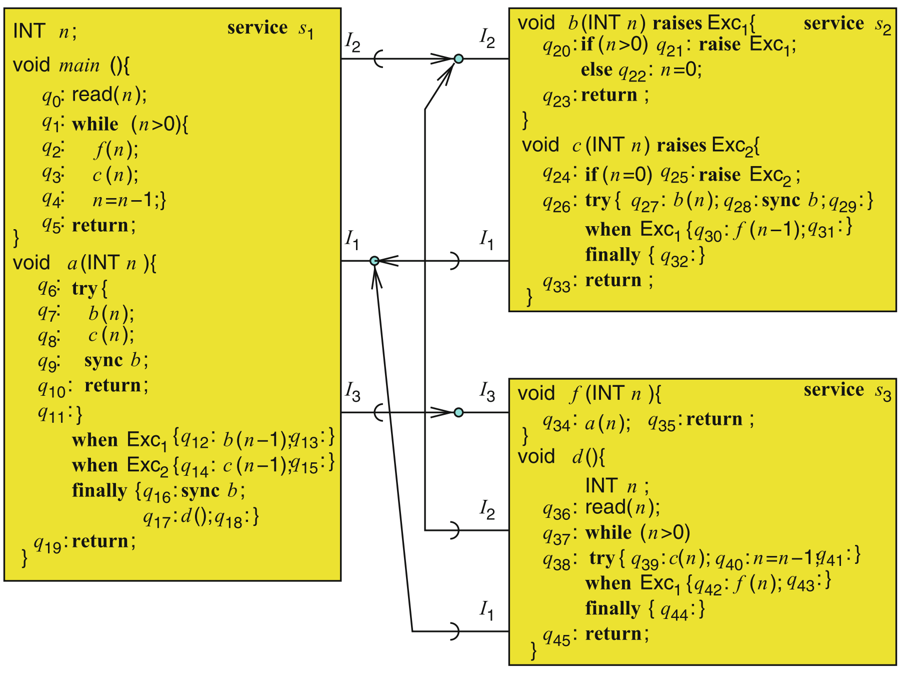
In contrast to these concepts, exception handling has a rather complex semantics: The statement **raise** E raises the exception E. This means the execution is being interrupted, i.e., it is not being continued by the execution of the next statement. If the exception E has been raised outside of a **try**-block, then the current procedure stops with the exception E, i.e., the corresponding call raises E. A try statement

```
try { ··· }
when E_1 { ··· }
···
when E_n { ··· }
finally { ··· }
```

is executed as follows (this is according to *Java*, *C#*, *BPEL*): The statements in the try-block are being executed as usual. If an exception E is raised, then the block of the first exception handler **when** $E_i\{\cdots\}$ with $E_i = E$ is executed.[1] If E is different from any exceptions in the exception handlers, the try statement terminates with exception E. Note that asynchronous procedure calls within a try-block, an exception handler, or a finally-block must be synchronized when the block is left. Without loss of generality, we assume that the last statement of try-block synchronizes asynchronous procedure calls within the block—either by a synchronize or by a return statement.

However, the execution of the try statement definitely finishes with the execution of the finally-block, no matter what

[1] This can be easily extended to a subtype hierarchy of exception types.



```
INT n;                                   service s1
void main (){
    q0: read(n);
    q1: while (n>0){
    q2:     f(n);
    q3:     c(n);
    q4:     n=n−1;}
    q5: return;
}
void a(INT n){
   q6: try{
   q7:    b(n);
   q8:    c(n);
   q9:   sync b;
   q10:  return;
   q11:}
        when Exc1 {q12: b(n−1);q13:}
        when Exc2 {q14: c(n−1);q15:}
        finally {q16: sync b;
                     q17: d();q18:}
  } q19: return;
```

```
void b(INT n) raises Exc1{            service s2
   q20: if (n>0) q21: raise Exc1;
        else q22: n=0;
   q23: return ;
}
void c(INT n) raises Exc2{
   q24: if (n=0) q25: raise Exc2 ;
   q26: try{ q27: b(n); q28: sync b; q29:}
        when Exc1 {q30: f(n−1);q31:}
        finally { q32:}
 } q33: return ;
```

```
void f(INT n){                        service s3
 } q34: a(n);  q35: return ;
void d(){
        INT n ;
   q36: read(n);
   q37: while (n>0)
   q38:  try{ q39: c(n); q40: n=n−1;q41:}
        when Exc1 {q42: f(n); q43:}
        finally { q44:}
  } q45: return;
```

```
interface I1 {
    sync void a(INT n);
}
```

```
interface I2 {
    exceptions Exc1, Exc2;
    async void b(INT n) raises Exc1;
    sync  void c(INT n) raises Exc2;
}
```

```
interface I3 {
    sync void f(INT n);
    sync void d();
}
```

Fig. 1 A service-oriented system

happens inside the try-block or the exception handler. Hence, a return statement or raising an unhandled exception within a try-block or an exception handler is earliest being executed after the finally-block has been executed. However, if the finally-block executes a return statement or raises an exception, then it returns from the current procedure or ends with an exceptional state without executing any open return or raise statement.

Example 1 Figure 1 shows a service-oriented system consisting of three services s_1, s_2, and s_3 with provided interfaces I_1, I_2, and I_3, respectively. The provided interfaces are shown by circles, the required interfaces are visualized by opened circles and service bindings are visualized by arrows. The symbols q_i represent program points indicating each statement. Procedure b of interface I_2 is the sole asynchronous procedure. Every other procedure is synchronous. The execution starts with calling *main* of service s_1. The return statement at program point q_{10} in Fig. 1 would not be executed if the finally block would be replaced by

```
finally  {  q16 : sync b;
            q17 : d();
            q18 : return;
         }
```

The reason is that before returning by the return statement at q_{10}, the finally block is being executed and this execution executes q_{18}. If there would be a return statement, then this would be the return from procedure a.

Remark 2 The semantics of loops, conditional statements, sequential execution, and the synchronous procedure call (and return) is surprisingly uniform across different programming languages. The semantics of exception handling – including the finally-statement– is according to *Java*, cf. Chapter 11.3 of [19]. The try statement with a finally clause of *C#* and *.NET* has an analogous semantics. We do not consider function calls as they could be transformed into procedure calls with result parameters. Similarly, other control structures such as different loops, switch statements can be transformed into while-loops and conditionals, respectively. Such transformations are often applied in compilers for intermediate code generation.

Remark 3 The interface I_s of a service can be specified using WSDL. The exceptions are specified by the `fault`-part in the operations. There are several approaches on Web Service Composition with exception handling, see e.g. [18,23]. In particular, the stub generated from a WSDL interface description of a service s might raise exceptions that can be handled by the client using s.

This article assumes that a protocol of a service s is given by a finite state machine $A_s \triangleq (\Sigma_s, R_s, \rightarrow_s, r_0^s, F_s)$ where

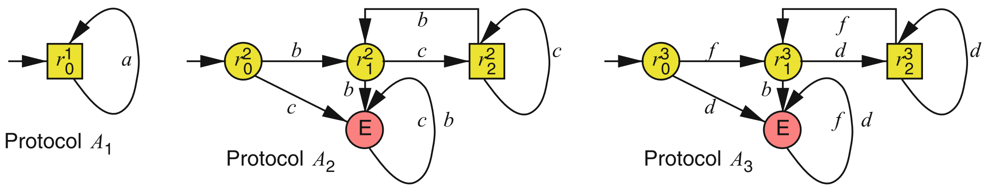
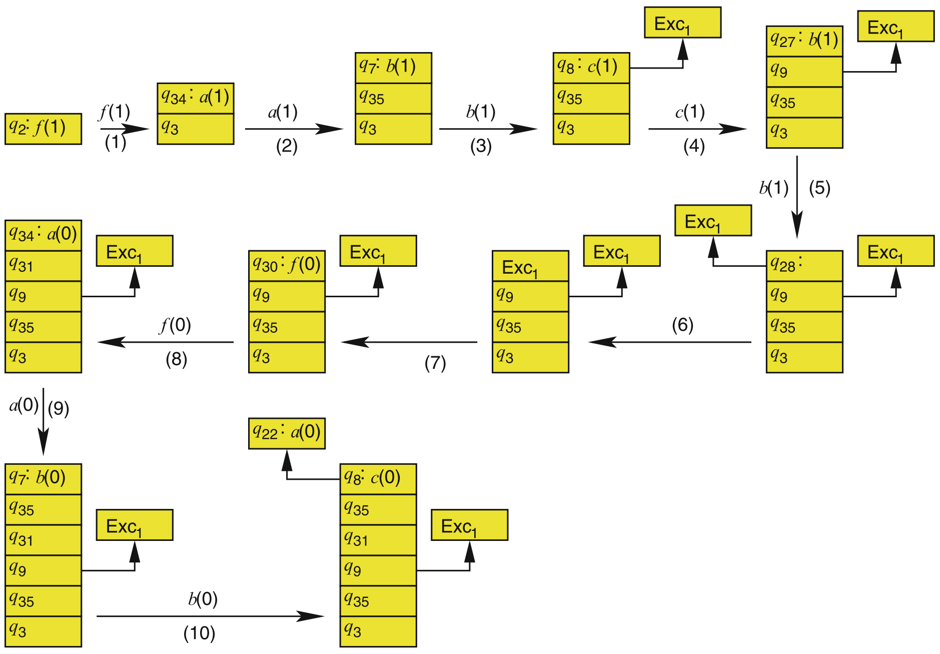


Fig. 2 Protocols of the services in Fig. 1

Fig. 3 A protocol violation in Fig. 1



Σ_s denotes the set of operations symbols in the interface of service s, R_s are the states of s, r_0^s is the state when s is started by a client, and F_s is the set of final states. A final state must be reached when the client finishes the use of s. Thus, the language $L(A_s)$ accepted by the finite state machine A_s defines the set of legal sequences of operation calls to s. Hence, a service protocol contains only operation calls available in the service description. In particular, a software architect is enabled here to define a required behaviour without knowing (or specifying) the service implementation. This is a main distinction in comparison with interaction protocols [9]. Figure 2 shows the state diagrams of protocols of the services in Fig. 1 which are used as an example throughout this article (final states are indicated by squares and initial states by an arrow without source). If the protocols A_2 or A_3 reach state E, then a protocol violation occurs. Here, A_2 permits only calls of b followed by a least one call of c; all other sequences lead to an error state.

Example 2 Figure 3 shows a possible execution of the service-oriented system in Fig. 1 when the value 1 is read. Steps (1) and (2) are synchronous calls. The program point after the call is pushed onto the runtime stack. Step (3) is an asynchronous call. In this case, the stack forks, i.e., it has now two branches: one for the caller and one for the callee. This kind of runtime structure is called a *cactus stack*. [13,21] showed that a cactus stack can be used as a runtime system for concurrent processes. Note that each stack in a cactus stack might be maintained on a different processor. Hence, cactus stacks are a well-suited as a logical runtime model for distributed systems such as service compositions. Thus, in Fig. 3, the cactus stacks after steps (5) and (10) could be maintained by three processors.

The call of $b(1)$ raises exception Exc$_1$. In a step, all interleavings are possible, i.e. any top element of a stack in the cactus stack can be taken for the next step. Suppose the call $c(1)$ (Step 4) is taken. Then, in (5), $b(1)$ is called, which also raises Exc$_1$. Since this exception is not handled, the synchronize statement results in Exc$_1$ (Step (6)). Step (7) shows that now the exception handler in the body of c is being executed and this calls $f(0)$. Then – as above – $a(0)$ is called. Note that the branch from q_9 cannot be removed until q_9 is on the top of the (main-) stack. This execution demonstrates that protocol A_2 is violated because service s_2 receives the calls $b(1)$, $c(1)$, $b(1)$, $b(0)$, i.e., A_2 will be in state E after this sequence. Furthermore, this sequence stems from the execution of an exception handler.

Thus, an abstract state can be represented as a cactus stack of program points and exception states and an abstract semantics transforms cactus stacks into cactus stacks. The transformations are (i) changing a state on one of the top stack elements, (ii) pushing a state on one of the stacks, (iii) poping a state from one of stacks, (iv) forking to a new stack, (v) synchronizing two stacks (i.e., waiting for a forked stack to be emptied).

According to [3], there is a one-to-one correspondence between cactus stacks and process-algebraic expressions. The set $PEX(Q)$ of process-algebraic expressions over a finite set Q (*atomic processes*) is the smallest set satisfying:

(i) $Q \subseteq PEX(Q)$
(ii) If $e, e' \in PEX(Q)$, then $e.e' \in PEX(Q)$ and $e \parallel e' \in PEX(Q)$ (*sequential* and *parallel composition*, respectively).
(iii) $\varepsilon \in PEX(Q)$

The *empty process*, denoted by ε, is the identity w.r.t. sequential and parallel composition.

For example, the cactus stack in Fig. 3 after step (10) can be represented by the process-algebraic expression $(((q_8 \| q_{22}).q_{35}.q_{31}.q_9) \| q_{\mathsf{Exc}_1}).q_{35}.q_3$.

Hence, the transformations of cactus stacks can be represented by rewrite rules for process-algebraic expressions, as demonstrated in Fig. 4. These rules are a short summary of our previous work [4,5]. The rewrite rules are labelled with the name of the operation provided by a service if an external service is being called while internal calls or control logic (e.g., thread operations) are labelled with λ. The reason is that protocol conformance checking only considers interaction sequences between services (w.r.t. the considered interaction protocol) but not internal procedure calls within a service.

A *process rewrite system* (short: PRS) is a tuple $\Pi \triangleq (\Sigma, Q, \rightarrow, q_0, F)$ where

(i) Q is a finite set (*atomic processes*),
(ii) Σ is a finite alphabet disjoint from Q (*actions*),
(iii) $q_0 \in Q$ (the *initial state*),
(iv) $\rightarrow \subseteq PEX(Q) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q)$ is a set of *process rewrite rules* ($\lambda \in \Sigma^*$ is the empty word),
(v) $F \subseteq Q \cup \{\varepsilon\}$ (the set of *final processes*).

The PRS Π defines a derivation relation $\Rightarrow \subseteq PEX(Q) \times \Sigma^* \times PEX(Q)$ (Σ^* is the set of all finite words over Σ) by the inference rules in Fig. 5. The set $L(\Pi) \triangleq \{w \in \Sigma^* : \exists f \in F \bullet q_0 \overset{w}{\Rightarrow} f\}$ is the *language accepted by the PRS* Π.

Remark 4 The second inference rules implies that rewrite rules can only applied to the top of one of the stacks in a cactus stack.

A service-oriented system S is abstracted to a process rewrite system $\Pi_S \triangleq (\Sigma, Q, \rightarrow, q_0, F)$ where $\Sigma = \bigcup_{s \in S} \Sigma_s$ is the set of all procedures in the interface descriptions of the services of S, $Q \triangleq PP \cup Exceptions$, PP is the set of program points, $Exceptions \triangleq \{q_E : E \text{ is an exception}\}$, $\rightarrow$ is defined by as in Fig. 4, q_0 is the program point where S starts, $F \triangleq Q_F \cup Exceptions$, and Q_F is the set of program points where the main program returns (i.e., the execution of the program terminates). Exceptional states are final because a program may terminate in an exceptional state.

Remark 5 In Fig. 4 there is slight difference to [4,5]: it uses $q \overset{\lambda}{\rightarrow} \varepsilon$ for a procedure. Furthermore, [5] shows a composi-

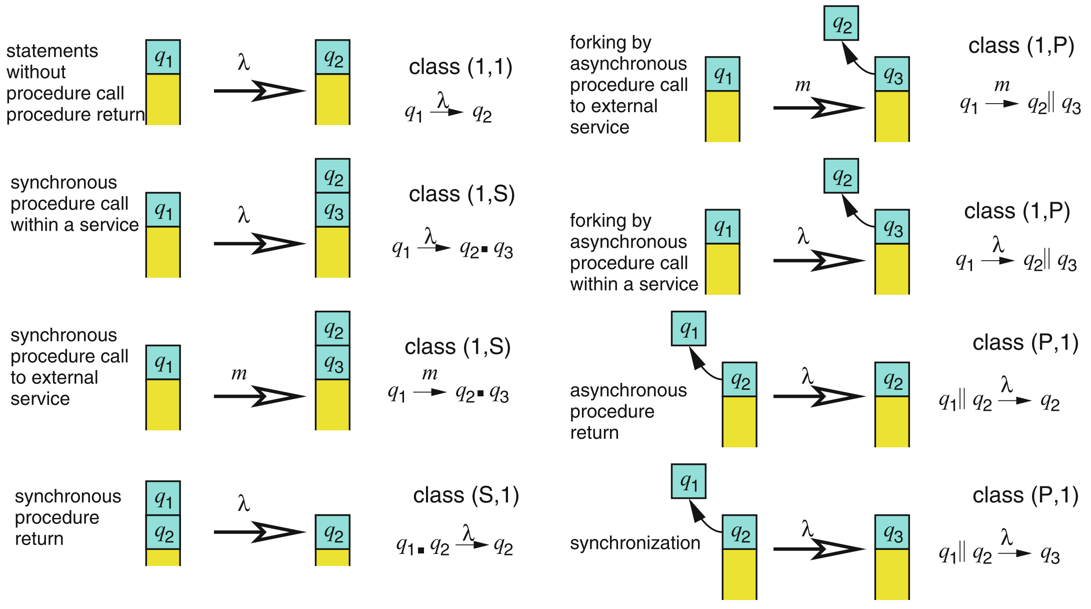

statements without procedure call procedure return
λ
class (1,1)
$q_1 \xrightarrow{\lambda} q_2$
synchronous procedure call within a service
class (1,S)
$q_1 \xrightarrow{\lambda} q_2 . q_3$
synchronous procedure call to external service
m
class (1,S)
$q_1 \xrightarrow{m} q_2 . q_3$
synchronous procedure return
class (S,1)
$q_1 . q_2 \xrightarrow{\lambda} q_2$
forking by asynchronous procedure call to external service
class (1,P)
$q_1 \xrightarrow{m} q_2 \| q_3$
forking by asynchronous procedure call within a service
class (1,P)
$q_1 \xrightarrow{\lambda} q_2 \| q_3$
asynchronous procedure return
class (P,1)
$q_1 \| q_2 \xrightarrow{\lambda} q_2$
synchronization
class (P,1)
$q_1 \| q_2 \xrightarrow{\lambda} q_3$

Fig. 4 Cactus stack transformations/process rewrite rules for abstract semantics

$$\frac{q \xrightarrow{a} q'}{q \overset{a}{\Rightarrow} q'} \ (\mathrm{R}) \qquad \frac{u \overset{a}{\Rightarrow} v}{u.w \overset{a}{\Rightarrow} v.w} \ (\mathrm{S}) \qquad \frac{u \overset{a}{\Rightarrow} v \quad v \overset{x}{\Rightarrow} w}{u \overset{ax}{\Rightarrow} w} \ (\mathrm{T})$$

$$\frac{u \overset{a}{\Rightarrow} v}{u \parallel w \overset{a}{\Rightarrow} v \parallel w} \ (\mathrm{P1}) \qquad \frac{u \overset{a}{\Rightarrow} v}{w \parallel u \overset{a}{\Rightarrow} w \parallel v} \ (\mathrm{P2}) \qquad \frac{}{u \overset{\lambda}{\Rightarrow} u} \ (\mathrm{L})$$

$$a \in \Sigma \cup \{\lambda\}, x \in \Sigma^*, u, v, w \in PEX(Q)$$

Fig. 5 Inference rules for the definition of the derivation relation in PRSs

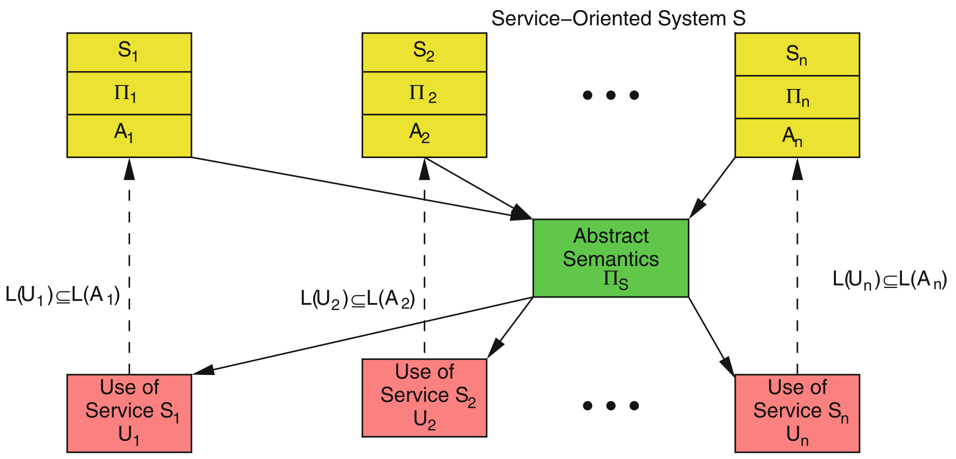


Fig. 6 Protocol conformance checking of a service-oriented system

tional construction of Π_S. For each $s \in S$, a PRS Π_s is automatically derived from the implementation of s by using classical compiler technology, and these PRSs are glued together to obtain Π_S. For reasons of space, this construction has been omitted. $L(\Pi_S)$ contains all possible interaction sequences between services that may happen during execution. Note that programming language concepts expressing fork–join parallelism (as e.g., in BPEL) can be abstracted analogously to calling asynchronous procedures and synchronize with them, respectively.

The use of a service s in service-oriented system S is defined as a PRS $U_s \triangleq (\Sigma_s, Q, \rightarrow_s, q_0, F)$ where Σ_s is the set of operations in the provided interface of s, Q, q_0, and F are defined as above, and $\rightarrow_s$ is defined as $\rightarrow$ except that all labels $l \notin \Sigma_s$ are replaced by λ. The protocol conformance checking problem checks for each service s of S whether $L(U_s) \subseteq L(A_s)$.

Mayr [26] classified the process rewrite rules according to the class of process-algebraic expressions on the left-hand side and right-hand side, respectively: Class 1 allows only single states, class S only single states or sequential expressions, class P only single states parallel expressions, and class G (general) allows arbitrary process-algebraic expressions. Figure 4 shows the class for each rule. An (x, y)-*PRS*, $x, y \in \{1, S, P, G\}$ allows only rules whose left-hand sides belong to class x and whose right-hand sides belong to class y.

Remark 6 (1, 1)-PRSs correspond to non-deterministic finite state machines (with λ-transitions), $(1, S)$-PRSs correspond to context-free systems, (S, S)-PRSs correspond to push-down machines, (P, P)-PRSs correspond to Petri-Nets, and $(1, G)$-PRSs correspond to process-algebras, cf. [26].

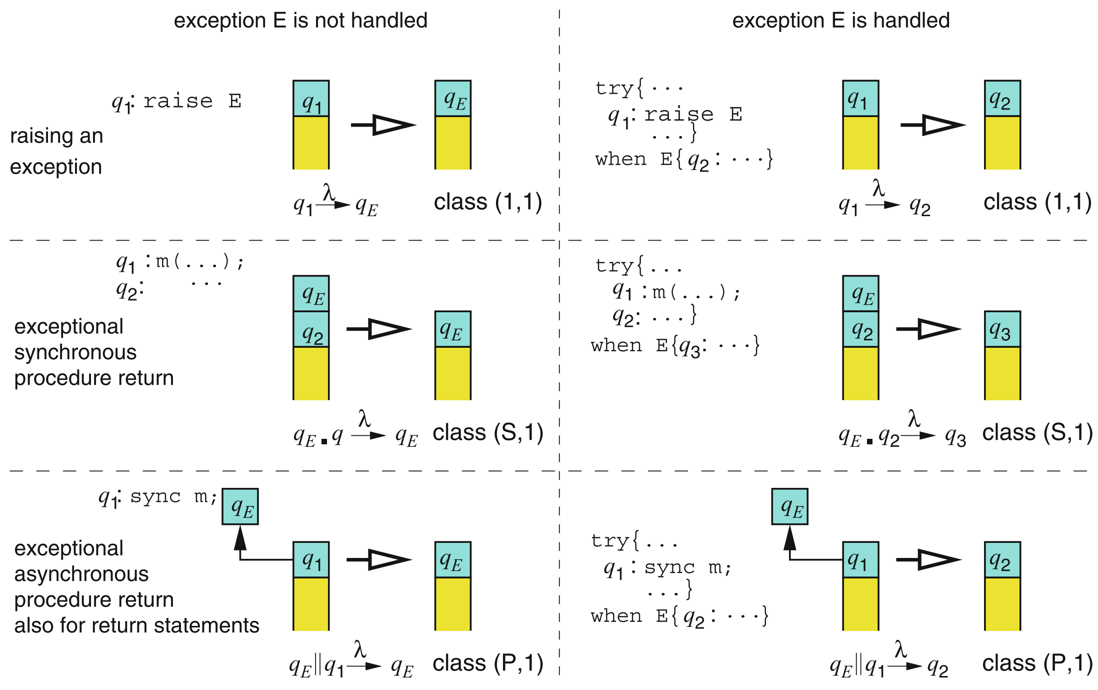
Figure 6 summarizes the approach for protocol conformance checking of a service-oriented system, cf. [4]. Each service is equipped with a protocol A_i. First, an abstract behaviour Π_i is determined and published for each service. Second, these abstract behaviours are glued to an abstract semantics Π_S that is defined as above. Third, the use U_i of each service S_i is determined as described above, and finally, for each service S_i, it is checked whether $L(U_i) \subseteq L(A_i)$.

3 Abstraction of exception handling

The aim of this section is to extend the abstraction to exception handling. We first discuss the abstractions due to exception handling without a finally-block and then discuss the abstraction of exception blocks with finally-blocks.

3.1 Exception handling without finally-Blocks

Figure 7 shows the PRS-rules specifying the abstract semantics for the raise statement and for exceptional procedure returns for synchronous and asynchronous procedures, i.e., the execution of the procedure stops with an unhandled exception. The left column shows the abstract semantics if the exception is not handled, the right column shows the abstract semantics if the exception is handled by a uniquely defined exception handler.



exception E is not handled
exception E is handled
q1: raise E
raising an exception
q1 →λ qE class (1,1)
try{ ··· q1: raise E ···} when E{q2: ···}
q1 →λ q2 class (1,1)
q1 : m(...); q2: ···
exceptional synchronous procedure return
qE.q →λ qE class (S,1)
try{ ... q1 : m(...); q2: ···} when E{q3: ···}
qE.q2 →λ q3 class (S,1)
q1: sync m;
exceptional asynchronous procedure return also for return statements
qE‖q1 →λ qE class (P,1)
try{ ... q1 : sync m; ...} when E{q2: ···}
qE‖q1 →λ q2 class (P,1)

Fig. 7 Exception handling without finally-blocks

Remark 7 For each statement (raise, procedure call, synchronization) that may raise an exception, it can be statically determined whether a given exception E is handled or unhandled. If exception E is handled, then the corresponding exception handler is statically defined. Without loss of generality, we assume that the set of unhandled exceptions of a procedure is contained in its signature (such a set can be determined statically). Thus, the execution of a procedure p may end with an exceptional state q_E. Thus, the PRS-rules in Fig. 7 can be computing from service implementations by using classical compiler technology.

The rules for the raise statement are straightforward, because they model the execution semantics exactly: If the exception is unhandled, the current execution stops with the exceptional state. Otherwise, the execution continues with the corresponding exception handler.

Consider now the case that a synchronous procedure m ends with an exceptional state q_E. Then, the caller of m raises exception E. There are two cases: exception E is unhandled (left column, Fig. 7) or handled by a corresponding exception handler (right column). For both cases, q_E is popped and handling exceptions is analogous to the raise statement, i.e., the top state is replaced by q_E if E is unhandled and replaced by the program point of the corresponding exception handler, if E is handled. The rules in the left row of Fig. 7 must be included into the abstract semantics for *each* procedure call of m.

Remark 8 For modelling exception handling, it is necessary to use PRS-rules with the sequential composition operator on its left-hand side because it must be modelled that an unhandled exception q_E within a procedure m leads to a re-raising exception at the caller. Therefore, instead of continue with the program point q' after the call, the program point must be replaced by the exception state. In contrast to the normal procedure return (which can be modelled by $q \to \varepsilon$), this replacement is only possible by the rule $q_E.q' \to q_E$.

The abstract semantics of exceptional returns from asynchronous procedures is analogous to exceptional returns from synchronous procedures. The main difference is that the state is a cactus stack that forks to a state q_E stemming from a call of an asynchronous procedure m. Since this is not a regular return from m, either m must be synchronized explicitly or implicitly by a returning from the callee.

3.2 Exception handling with finally-Blocks

The abstract semantics for finally-blocks must ensure that the finally block is being executed, even if the corresponding try-block (or a corresponding exception handler) executes a return statement or raise an exception not handled within a corresponding exception handler. This also applies for nested try-blocks. Consider for example the following situation:

try { ···
  try { ··· q : **raise** E' ··· }
  when E { ··· } //$\mathsf{E} \neq \mathsf{E}'$
  finally { ··· }
when E' { ··· }
finally { ··· }

The raised exception is only handled in an outer try-block. Before executing the exception handler for E′, all finally-blocks that correspond to inner finally-blocks containing the statement raising E must be executed. Thus, the abstract semantics of the rules in Fig. 7 can only be used if none of the inner try-blocks contains a finally-block.

The main idea to ensure the execution of a finally-block is to push the first program point of the finally-block to the stack when entering a try-block, cf. Rule (1) of Fig. 8. For this purpose, program points are assigned to each try-block with a corresponding finally-block, to the end of each try-block, to the end of exception handling block, and to the end of the finally-block, cf. Fig. 8. Furthermore, if an inner try-block raises an exception E that is not handled by its exception handlers, the try-block ends with state q_E. Figure 8 shows the PRS-rules for the abstract semantics.

If the execution reaches the end of the try-block or the end of one of its exception handlers, then the abstract associated program point can be obtained by the second element of the stack, cf. Rules (2) and (3). Together with Rule (4), these rules ensure that the finally-block is being executed and after its execution, the statement s after the finally-block is being executed. This is the reason for pushing the program point of s onto the stack in Rules (2) and (3). If the try-block executes a return statement, then this execution must be postponed until the try-block has been completed and its program point must be pushed onto the stack. For this, the two elements on the top of the stack must be exchanged, cf. Rule (5). Together with Rule (6) for regularly leaving the finally-block, these rules ensure that the return statement is being executed after finishing the finally-block.

The situation is more complicated if the finally-block executes a return statement because in this case, the return statement of the try-block is not being executed. Consider for example Fig. 8: If q_7: **return** is executed, then q_2: **return** is not executed although q_2 is the second top element on the stack (by Rule (5)). Therefore, the second element on the stack must be removed, cf. Rule (7). Rule (7) is also able to deal with nested try-blocks. Consider for example a synchronous procedure p where the body contains the following nested try-block:

try { $\cdots$
  try { $\cdots$ q_1 : **return** $\cdots$ }
  when E { $\cdots$ } //E $\neq$ E′
  finally { $\cdots$ q_2 : **return** $\cdots$ }
when E′ { $\cdots$ }
finally { $\cdots$ q_3 : **return** $\cdots$ }

When executing q_3 : **return**, the four program points on the top of the stack are $q_3.q_2.q_1.q'$ where q' is a program point after a procedure call of p. According to Rule (7),

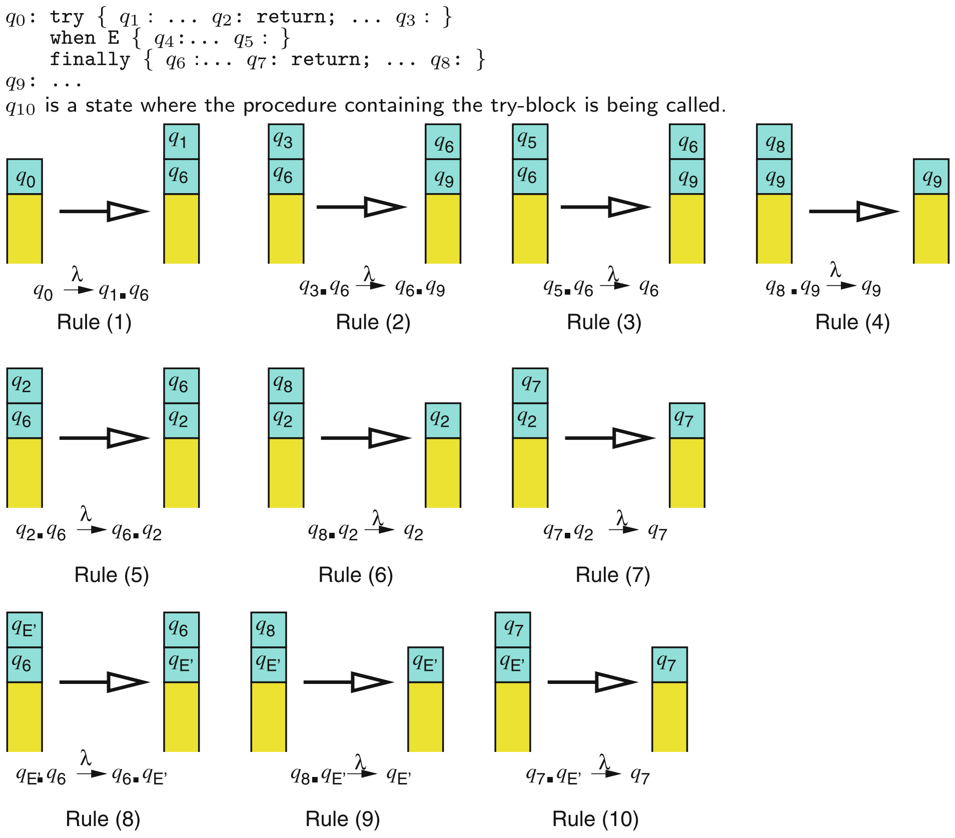


Fig. 8 Abstract semantics for exception handling with finally-blocks

q_0: try { q_1 : ... q_2: return; ... q_3 : }
    when E { q_4:... q_5 : }
    finally { q_6 :... q_7: return; ... q_8: }
q_9: ...
q_{10} is a state where the procedure containing the try-block is being called.

$q_0 \xrightarrow{\lambda} q_1.q_6$
Rule (1)
$q_3.q_6 \xrightarrow{\lambda} q_6.q_9$
Rule (2)
$q_5.q_6 \xrightarrow{\lambda} q_6$
Rule (3)
$q_8.q_9 \xrightarrow{\lambda} q_9$
Rule (4)
$q_2.q_6 \xrightarrow{\lambda} q_6.q_2$
Rule (5)
$q_8.q_2 \xrightarrow{\lambda} q_2$
Rule (6)
$q_7.q_2 \xrightarrow{\lambda} q_7$
Rule (7)
$q_{E'}.q_6 \xrightarrow{\lambda} q_6.q_{E'}$
Rule (8)
$q_8.q_{E'} \xrightarrow{\lambda} q_{E'}$
Rule (9)
$q_7.q_{E'} \xrightarrow{\lambda} q_7$
Rule (10)

Fig. 9 Abstract semantics of the services in Fig. 1

procedure $main$:
(1) $q_0 \xrightarrow{\lambda} q_1$ (2) $q_1 \xrightarrow{\lambda} q_2$ (3) $q_1 \xrightarrow{\lambda} q_5$ (4) $q_2 \xrightarrow{f} q_{34}.q_3$ (5) $q_3 \xrightarrow{c} q_{24}.q_4$ (6) $q_4 \xrightarrow{\lambda} q_5$
procedure a:
(7) $q_6 \xrightarrow{\lambda} q_7.q_{16}$ (14) $q_{\mathsf{Exc}_2} \| q_{\mathsf{Exc}_1} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (20) $q_{\mathsf{Exc}_1}.q_{16} \xrightarrow{\lambda} q_{12}.q_{16}$ (26) $q_{16} \| q_{23} \xrightarrow{\lambda} q_{17}$
(8) $q_7 \xrightarrow{b} q_8 \| q_{20}$ (15) $q_{\mathsf{Exc}_1} \| q_{\mathsf{Exc}_2} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (21) $q_{\mathsf{Exc}_2}.q_{16} \xrightarrow{\lambda} q_{14}.q_{16}$ (27) $q_{16} \| q_{\mathsf{Exc}_1} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$
(9) $q_8 \xrightarrow{c} q_{24}.q_9$ (16) $q_{\mathsf{Exc}_1} \| q_{\mathsf{Exc}_1} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (22) $q_{12} \xrightarrow{b} q_{13} \| q_{20}$ (28) $q_{17} \xrightarrow{d} q_{36}.q_{18}$
(10) $q_9 \| q_{23} \xrightarrow{\lambda} q_{10}$ (17) $q_{10}.q_{35} \xrightarrow{\lambda} q_{35}$ (23) $q_{13}.q_{16} \xrightarrow{\lambda} q_{16}$ (29) $q_{18}.q_{10} \xrightarrow{\lambda} q_{10}$
(11) $q_{23} \| q_9 \xrightarrow{\lambda} q_{10}$ (18) $q_{10}.q_{16} \xrightarrow{\lambda} q_{16}.q_{10}$ (24) $q_{14} \xrightarrow{c} q_{24}.q_{15}$ (30) $q_{18}.q_{19} \xrightarrow{\lambda} q_{19}$
(12) $q_{\mathsf{Exc}_1} \| q_9 \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (19) $q_{11}.q_{16} \xrightarrow{\lambda} q_{16}.q_{19}$ (25) $q_{15}.q_{16} \xrightarrow{\lambda} q_{16}$ (31) $q_{19}.q_{35} \xrightarrow{\lambda} q_{35}$
(13) $q_9 \| q_{\mathsf{Exc}_1} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$
procedure b:
(32) $q_{20} \xrightarrow{\lambda} q_{21}$ (33) $q_{20} \xrightarrow{\lambda} q_{22}$ (34) $q_{21} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (35) $q_{22} \xrightarrow{\lambda} q_{23}$ (36) $q_8 \| q_{23} \xrightarrow{\lambda} q_8$
procedure c:
(37) $q_{24} \xrightarrow{\lambda} q_{25}$ (42) $q_{28} \| q_{23} \xrightarrow{\lambda} q_{29}$ (47) $q_{\mathsf{Exc}_1}.q_{32} \xrightarrow{\lambda} q_{30}.q_{32}$ (52) $q_{32} \xrightarrow{\lambda} q_{33}$
(38) $q_{24} \xrightarrow{\lambda} q_{26}$ (43) $q_{23} \| q_{28} \xrightarrow{\lambda} q_{29}$ (48) $q_{30} \xrightarrow{f} q_{34}.q_{31}$ (53) $q_{33}.q_4 \xrightarrow{\lambda} q_4$
(39) $q_{25} \xrightarrow{\lambda} q_{\mathsf{Exc}_2}$ (44) $q_{\mathsf{Exc}_1} \| q_{28} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (49) $q_{31}.q_{32} \xrightarrow{\lambda} q_{32}$ (54) $q_{33}.q_9 \xrightarrow{\lambda} q_9$
(40) $q_{26} \xrightarrow{\lambda} q_{27}.q_{32}$ (45) $q_{28} \| q_{\mathsf{Exc}_1} \xrightarrow{\lambda} q_{\mathsf{Exc}_1}$ (50) $q_{\mathsf{Exc}_2}.q_{32} \xrightarrow{\lambda} q_{32}.q_{\mathsf{Exc}_2}$ (55) $q_{33}.q_{39} \xrightarrow{\lambda} q_{39}$
(41) $q_{27} \xrightarrow{b} q_{28} \| q_{20}$ (46) $q_{29}.q_{32} \xrightarrow{\lambda} q_{32}$ (51) $q_{32}.q_{\mathsf{Exc}_2} \xrightarrow{\lambda} q_{\mathsf{Exc}_2}$
procedure f
(56) $q_{34} \xrightarrow{a} q_6.q_{35}$ (57) $q_{35}.q_3 \xrightarrow{\lambda} q_3$ (58) $q_{35}.q_{31} \xrightarrow{\lambda} q_{31}$ (59) $q_{35}.q_{43} \xrightarrow{\lambda} q_{43}$
procedure d
(60) $q_{36} \xrightarrow{\lambda} q_{37}$ (64) $q_{39} \xrightarrow{c} q_{24}.q_{40}$ (68) $q_{42} \xrightarrow{f} q_{34}.q_{43}$ (72) $q_{44} \xrightarrow{\lambda} q_{45}$
(61) $q_{37} \xrightarrow{\lambda} q_{38}$ (65) $q_{40} \xrightarrow{\lambda} q_{41}$ (69) $q_{43}.q_{44} \xrightarrow{\lambda} q_{44}$ (73) $q_{45}.q_{18} \xrightarrow{\lambda} q_{18}$
(62) $q_{37} \xrightarrow{\lambda} q_{45}$ (66) $q_{41}.q_{44} \xrightarrow{\lambda} q_{44}$ (70) $q_{\mathsf{Exc}_2}.q_{44} \xrightarrow{\lambda} q_{44}.q_{\mathsf{Exc}_2}$
(63) $q_{38} \xrightarrow{\lambda} q_{39}.q_{44}$ (67) $q_{\mathsf{Exc}_1}.q_{44} \xrightarrow{\lambda} q_{42}.q_{44}$ (71) $q_{44}.q_{\mathsf{Exc}_2} \xrightarrow{\lambda} q_{\mathsf{Exc}_2}$

the abstract semantics contains the rules[2] $q_3.q_2 \xrightarrow{\lambda} q_3$ and $q_3.q_1 \xrightarrow{\lambda} q_3$. Together with the PRS-rule $q_3.q' \xrightarrow{\lambda} q'$, it can be shown that $q_3.q_2.q_1.q' \overset{\lambda}{\Rightarrow} q'$, i.e., the procedure returns by the return statement of the outer finally block. A simple inductive argument shows that such a derivation can be constructed for arbitrarily nested try-blocks with corresponding finally-blocks that contain return statements. Rules (8), (9), and (10) of Fig. 8 correspond to Rules (5), (6), and (7) when the try-block raises an unhandled exception E' or an exception handler raises exception E'.

Remark 9 Similar arguments as above also apply to asynchronous procedures. Here, two stacks of a cactus stack have to be considered. The situation on the top of the two stacks is $(q_3.q_2.q_1) \| q'$ where q' is a synchronization statement or any program point between the asynchronous procedure call before the corresponding synchronization statement. Then, it holds $(q_3.q_2.q_1) \| q' \overset{\lambda}{\Rightarrow} q_3 \| q'$. Now, the process rewrite rules of the abstract semantics for synchronization or asynchronous procedure return can be applied.

Example 3 Figure 9 shows the process rewrite rules of the abstract semantics for the service-oriented system in Fig. 1. The initial state is q_0, the set of final states is $F = \{q_9, q_{\mathsf{Exc}_1}, q_{\mathsf{Exc}_2}\}$. The following derivation corresponds to the execution in Fig. 3 (the rule is as a lower index of the arrow):

$$q_0 \overset{\lambda}{\Rightarrow}_{(2)} q_2 \overset{f}{\Rightarrow}_{(4)} q_{34}.q_3 \overset{a}{\Rightarrow}_{(56)} q_6.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(7)} q_7.q_{16}.q_{35}.q_3 \overset{b}{\Rightarrow}_{(8)} (q_8 \| q_{20}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(32)} (q_8 \| q_{21}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(34)} (q_8 \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{c}{\Rightarrow}_{(9)} ((q_{24}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(38)} ((q_{26}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(40)} ((q_{27}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{b}{\Rightarrow}_{(41)} (((q_{28} \| q_{20}).q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(32)} (((q_{28} \| q_{21}).q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(34)} (((q_{28} \| q_{Exc_1}).q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(45)} ((q_{Exc_1}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(47)} ((q_{30}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{f}{\Rightarrow}_{(48)} ((q_{34}.q_{31}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{a}{\Rightarrow}_{(56)} ((q_6.q_{35}.q_{31}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(7)} ((q_7.q_{16}.q_{35}.q_{31}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{b}{\Rightarrow}_{(8)} (((q_8 \| q_{20}).q_{16}.q_{35}.q_{31}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

$$\overset{\lambda}{\Rightarrow}_{(33)} (((q_8 \| q_{22}).q_{16}.q_{35}.q_{31}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$$

At the end of the second line, rules (9) and rules (32) are applicable. It is worth to compare the last expression of the seventh line with the cactus stack in Fig. 3 after step (3). It has basically the same shape except that program points q_{32} and q_{16} are not present in the cactus stack. These program points are the first program points of the finally-blocks and are pushed onto the stack when the corresponding try statement is executed.

[2] Formally, it also contains the rule $q_2.q_1 \xrightarrow{\lambda} q_2$, but this rule plays no role in the discussion.

4 Protocol conformance checking

For checking the conformance of a protocol of a service s in a service-oriented system S, it must be checked whether $L(U_s) \subseteq L(A_s)$ where the use of s is defined by the PRS $U_s \triangleq (\Sigma_s, Q, \rightarrow, q_0, F)$ and $A_s \triangleq (\Sigma_s, R_s, \rightarrow_s, r_0^s, F_s)$ is the finite state machine defining the protocol of s. Both and Zimmermann [5] proves that this problem is undecidable for the classes of (x, G) process rewrite systems, i.e., if sequential composition and parallel composition occurs in a PRS, the protocol conformance checking becomes undecidable.

The goal is to construct—similar to [5]—the *Combined Abstraction*. The Combined Abstraction is a PRS K with the following properties:

(i) K belongs to the same class of PRSs as U_s,
(ii) $L(K) \supseteq L(U_s) \cap (\Sigma_s^* \backslash L(A_s))$, and
(iii) if U_s belongs to one of the classes of (x, y)-PRSs, $y \in \{1, S\}$) then $L(K) = L(U_s) \cap (\Sigma_s^* \backslash L(A_s))$.

Note that $\Sigma_s^* \backslash A_s$ is the language accepted by the finite state machine $\bar{A}_s \triangleq (\Sigma_s, R_s, \rightarrow_s, r_0^s, \bar{F})$ where $\bar{F}_s \triangleq R_s \setminus F_s$ is the set of all non-final states of A_s. Thus, $L(K) \neq \emptyset$ implies protocol conformance.

Both and Zimmermann [5] defines the Combined Abstraction for the class of $(1, G)$-PRS and [4] extends it to the class of (P, G)-PRS. This article extends it further to the class of (G, G)-PRS. Here, the construction of the Combined Abstraction is based on a *normalized process rewrite system* that consists only of rules of the forms $q \xrightarrow{\alpha} q'$, $q.q' \xrightarrow{\alpha} q''$, $q \xrightarrow{\alpha} q'.q''$, $q \parallel q' \xrightarrow{\alpha} q''$, and $q \xrightarrow{\alpha} q' \parallel q''$. Mayr [26] shows that for any PRS Π there exists a normalized PRS Π' with $L(\Pi) = L(\Pi')$. However, Π' may have more atomic processes.

In contrast to [5], the construction of the Combined Abstraction for (G, G)-PRS is based on the construction of a pushdown system in [22] that accepts the intersection of a context-free language and a regular language. The *Combined Abstraction* of $U_s = (\Sigma_s, Q, \rightarrow, q_0, F)$ and $\bar{A}_s = (\Sigma_s, R_s, \rightarrow_s, r_0^s, \bar{F}_s)$ is a process rewrite system $K \triangleq (\Sigma_s, Q_K, \rightarrow_K, q_0^K, F_K)$ where $Q_K \triangleq R_s \times Q$, $q_0^k \triangleq (r_0, q_0)$, $\rightarrow_K \triangleq T_{11} \cup T_{1S} \cup T_{S1} \cup T_{1P} \cup T_{P1} \cup T_0$ as defined in Fig. 10, and $F_K \triangleq F_s \times F$. The proof of its correctness is based on the following

Theorem 1 *Let $U_s \triangleq (\Sigma_s, Q, \rightarrow, q_0, F)$ be a PRS, $\bar{A}_s \triangleq (\Sigma_s, R_s, \rightarrow_s, r_0^s, \bar{F}_s)$ be a finite state machine, and $K \triangleq (\Sigma_s, Q_K, \rightarrow_K, q_0^K, F_K)$ be the Combined Abstraction of U_s and $\bar{A}_s$. Furthermore, let be $e \in PEX(Q)$ be a process-algebraic expression over Q and $r \in R_s$ a protocol state, and $x \in \Sigma_s^*$ such that there is a $f \in F$ and $\bar{r} \in \bar{F}_s$ with $e \overset{x}{\Rightarrow} f$ and $r \overset{x}{\Rightarrow}_s \bar{r}$. Then, there is a $e' \in PEX(Q_K)$ and a $f_k \in F_K$ such that the following properties are satisfied:*

(i) e and e' have the same shape, i.e., e is obtained from e' by removing the first component contained in each atomic process of e'.
(ii) All states on the top of the stacks in the cactus stack corresponding to e' have the protocol state r.
(iii) $e' \overset{x}{\Rightarrow}_K f_k$

Proof See Appendix 1 □

Corollary 1 *Let $U_s \triangleq (\Sigma_s, Q, \rightarrow, q_0, F)$ be a PRS, $\bar{A}_s \triangleq (\Sigma_s, R_s, \rightarrow_s, r_0^s, \bar{F}_s)$ be a finite state machine, and $K \triangleq (\Sigma_s, Q_K, \rightarrow_K, q_0^K, F_K)$ be the Combined Abstraction of U_s and $\bar{A}_s$. Then, $L(U_s) \cap L(A_s) \subseteq L(K)$.*

Proof Let be $x \in L(A_s) \cap L(U_s)$. Then, $q_0 \overset{x}{\Rightarrow} f$ for a $f \in F$ and $r_0 \overset{x}{\Rightarrow} \bar{r}$ for a $\bar{r} \in \bar{F}_s$. Hence, by Theorem 1(iii) there is a $f_k \in F_K$ such that $(r_0, q_0) \overset{x}{\Rightarrow}_K (a\bar{r}, f)$. Thus, $x \in L(K)$ since $q_0^K = (r_0, q_0)$. □

Remark 10 If the Combined Abstraction only contains rules from $T_{11} \cup T_{1S} \cup T_{S1}$, then equivalence holds. In particular, there is no need to apply Lemma 2. The combined abstraction specializes to the construction as well the proof in Appendix 1 to the intersection of pushdown machines and finite state machines as described in [22]. In this case, equivalence holds [22].

If condition (ii) is violated, then the cactus stack would have two top-of-stack elements with different protocol states. This makes no sense because there is only one protocol of service s, and therefore, the protocol cannot be at the same time in two different states. We call cactus stack violating condition (ii) as *inconsistent*.

The transition rules T_{11}, T_{1S}, and T_{S1} are a slight generalization of those in [22]. Note that $r \overset{\lambda}{\Rightarrow}_s r$, and for $a \in \Sigma_s$

Fig. 10 Transition rules for the combined abstraction

$$\begin{aligned}
T_{11} &\triangleq \{(r,q) \xrightarrow{\alpha}_K (r',q') : \alpha \in \Sigma_s \cup \{\lambda\} \wedge r, r' \in R_s \wedge q, q' \in Q \wedge (r \xrightarrow{\alpha}_s r') \wedge (q \xrightarrow{\alpha} q')\} \\
T_{1S} &\triangleq (r,q) \xrightarrow{\alpha}_K (r',q').(r'',q'') : \alpha \in \Sigma_s \cup \{\lambda\} \wedge r, r', r'' \in R_s \wedge \\
&\qquad q, q', q'' \in Q \wedge (r \overset{\alpha}{\Rightarrow}_s r') \wedge (q \xrightarrow{\alpha} q'.q'')\} \\
T_{S1} &\triangleq \{(r,q).(r'',q') \xrightarrow{\alpha}_K (r',q'') : \alpha \in \Sigma_s \cup \{\lambda\} \wedge r, r', r'' \in R_s \wedge \\
&\qquad q, q', q'' \in Q \wedge (r \overset{\alpha}{\Rightarrow}_s r') \wedge (q.q' \xrightarrow{\alpha} q'')\} \\
T_{1P} &\triangleq (r,q) \xrightarrow{\alpha}_K (r',q') \parallel (r',q'') : \alpha \in \Sigma_s \cup \{\lambda\} \wedge r, r' \in R_s \wedge \\
&\qquad q, q', q'' \in Q \wedge (r \overset{\alpha}{\Rightarrow}_s r') \wedge (q \xrightarrow{\alpha} q' \parallel q'')\} \\
T_{P1} &\triangleq \{(r,q) \parallel (r,q') \xrightarrow{\alpha}_K (r',q'') : \alpha \in \Sigma_s \cup \{\lambda\} \wedge r, r' \in R_s \wedge \\
&\qquad q, q', q'' \in Q \wedge (r \overset{\alpha}{\Rightarrow}_s r') \wedge (q \parallel q' \xrightarrow{\alpha} q'')\} \\
T_0 &\triangleq \{(r,q) \xrightarrow{\lambda}_K (r',q) : r, r' \in R_s \wedge q \in Q \wedge \exists \alpha \in \Sigma \cup \{\lambda\} \bullet r \xrightarrow{\alpha}_s r'\}
\end{aligned}$$

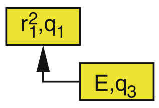


Fig. 11 Inconsistent cactus stack

it is $r \overset{a}{\Rightarrow}_s r'$ iff $r \overset{a}{\rightarrow}_s r'$. The transition rules T_{1P} and T_{P1} are straightforward. The ideas stem from [4,5]. The transition rules T_0 are required for maintaining a consistent state of the protocol as demonstrated by Example 4:

Example 4 Consider the finite state machine A_2 in Fig. 2 and suppose that U_s contains the following transition rules $q_0 \overset{b}{\rightarrow} q_1 \parallel q_2$, $q_1 \overset{c}{\rightarrow} q_3$, $q_2 \overset{b}{\rightarrow} q_3$, $q_3 \parallel q_3 \overset{c}{\rightarrow} q_4$ where q_4 is the final state. Then, $bbcc \in L(U_s) \setminus L(A_2)$, i.e., there is a protocol violation. Without the rules in T_0, the following derivations are possible:

$$
\begin{array}{lll}
q_K \overset{\lambda}{\Rightarrow} (r_0^2, q_0) & \text{by } T_S \\
\overset{b}{\Rightarrow} (r_1^2, q_1) \parallel (r_1^2, q_2) & \text{by } T_{1P} \\
\overset{b}{\Rightarrow} (r_1^2, q_1) \parallel (E, q_3) & \text{by } T_{11} \\
\overset{c}{\Rightarrow} (r_1^2, q_1) \parallel (E, q_4) & \text{by } T_{11} \\
\overset{c}{\Rightarrow} (r_2^2, q_3) \parallel (E, q_4) & \text{by } T_{11}
\end{array}
\qquad
\begin{array}{lll}
q_K \overset{\lambda}{\Rightarrow} (r_0^2, q_0) & \text{by } T_S \\
\overset{b}{\Rightarrow} (r_1^2, q_1) \parallel (r_1^2, q_2) & \text{by } T_{1P} \\
\overset{b}{\Rightarrow} (r_1^2, q_1) \parallel (E, q_3) & \text{by } T_{11} \\
\overset{c}{\Rightarrow} (r_2^2, q_3) \parallel (E, q_3) & \text{by } T_{11} \\
\overset{c}{\Rightarrow} (r_2^2, q_4) \parallel (E, q_3) & \text{by } T_{11}
\end{array}
$$

For none of these two process-algebraic expressions, there are λ-transitions that lead to a final state. Thus, the protocol violation is not detected. The reason is that for both derivations, a change of the protocol state was not taken into account. For both derivations, after the second step, the left operand of $\parallel$ still indicates that A_2 is in state r_1^2 although by the transitions the (final) protocol state E is reached for the right operand. Figure 11 shows the corresponding cactus stack. Each top element of a cactus stack should have the same protocol state because there is only one protocol, and therefore, the protocol state must be unique. With the rules of T_0, it is possible to change the protocol state of the left operand to E before applying another transition rule. Thus, $(r_1^2, q_1) \parallel (E, q_3) \overset{\lambda}{\Rightarrow} (E, q_1) \parallel (E, q_3) \overset{c}{\Rightarrow} (E, q_3) \parallel (E, q_3) \overset{c}{\Rightarrow} (E, q_4) \in F_K$ for both cases. Hence, the protocol violation is detected.

Example 5 (Combined Abstraction) The Combined Abstraction of the PRS in Fig. 9 and the protocol automaton A_2 in Fig. 2 has 220 atomic processes and 398 transition rules. For reasons of space we only give a derivation demonstrating the protocol violation discussed in Example 2 (the class of the applied transition rule according to Fig. 10 is indicated below the derivation step, T_{SS} is a combination of T_{S1} and T_{1S}), cf. Fig. 12. This derivation should be compared with the derivation in Example 3. Note that all operations different from b and c are replaced by λ since the other operations are not contained in the interface of service s_2.

$$
\begin{array}{l}
\langle r_0^2, q_0\rangle \underset{T_{11}}{\overset{\lambda}{\Rightarrow}} \langle r_0^2, q_1\rangle \underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} \langle r_0^2, q_{34}\rangle.\langle r_0^2, q_3\rangle \underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} \langle r_0^2, q_6\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} \langle r_0^2, q_7\rangle.\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \underset{T_{1P}}{\overset{b}{\Rightarrow}} (\langle r_1^2, q_8\rangle \| \langle r_1^2, q_{20}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{11}}{\overset{\lambda}{\Rightarrow}} (\langle r_1^2, q_8\rangle \| \langle r_1^2, q_{21}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{11}}{\overset{\lambda}{\Rightarrow}} (\langle r_1^2, q_8\rangle \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{c}{\Rightarrow}} ((\langle r_2^2, q_{24}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_0}{\overset{\lambda}{\Rightarrow}} ((\langle r_2^2, q_{24}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_2^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{11}}{\overset{\lambda}{\Rightarrow}} ((\langle r_2^2, q_{26}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_2^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} ((\langle r_2^2, q_{27}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_2^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1P}}{\overset{b}{\Rightarrow}} (((\langle r_1^2, q_{28}\rangle \| \langle r_1^2, q_{20}\rangle).\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_2^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_0}{\overset{\lambda}{\Rightarrow}} (((\langle r_1^2, q_{28}\rangle \| \langle r_1^2, q_{20}\rangle).\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{11}}{\overset{\lambda}{\Rightarrow}} (((\langle r_1^2, q_{28}\rangle \| \langle r_1^2, q_{21}\rangle).\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{11}}{\overset{\lambda}{\Rightarrow}} (((\langle r_1^2, q_{28}\rangle \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{P1}}{\overset{\lambda}{\Rightarrow}} ((\langle r_1^2, q_{\mathsf{Exc}_1}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{SS}}{\overset{\lambda}{\Rightarrow}} ((\langle r_1^2, q_{30}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} ((\langle r_1^2, q_{34}\rangle.\langle r_1^2, q_{31}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} ((\langle r_1^2, q_6\rangle.\langle r_1^2, q_{35}\rangle.\langle r_1^2, q_{31}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1S}}{\overset{\lambda}{\Rightarrow}} ((\langle r_1^2, q_7\rangle.\langle r_1^2, q_{16}\rangle.\langle r_1^2, q_{35}\rangle.\langle r_1^2, q_{31}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle).\langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\underset{T_{1P}}{\overset{b}{\Rightarrow}} (((\langle E, q_8\rangle \| \langle E, q_{20}\rangle).\langle r_1^2, q_{16}\rangle.\langle r_1^2, q_{35}\rangle.\langle r_1^2, q_{31}\rangle.\langle r_2^2, q_{32}\rangle.\langle r_2^2, q_9\rangle) \| \langle r_1^2, q_{\mathsf{Exc}_1}\rangle). \\
\hfill \langle r_0^2, q_{16}\rangle.\langle r_0^2, q_{35}\rangle.\langle r_0^2, q_3\rangle \\
\overset{cbcb}{\Longrightarrow} \langle E, q_9\rangle
\end{array}
$$

Fig. 12 A derivation using the rules of a combined abstraction (Example 5)

5 Related work

Many works on static protocol checking of components consider local protocol checking on FSMs. The same approach can also be applied to check protocols of objects in object-oriented systems. The idea of static type checking by using FSMs goes back to Nierstrasz [28]. His approach uses regular languages to model the dynamic behaviour of objects, which is less powerful than context-free grammars (CFG). Therefore, the approach cannot handle recursive callbacks. In [25], object-life cycles for the dynamic exchange of implementations of classes and methods using a combination of the bridge/strategy pattern are considered. The approach comprises dynamic as well as static conformance checking. However, it is also based on FSMs, which are in general still used widely in similar approaches (e.g., [15,27,35]). Tenzer and Stevens [38] investigate approaches for checking object-life cycles. They assume that object-life cycles of UML-classes are described using UML state-charts and that for each method of a client, there is a FSM that describes the calling sequence from that method. In order to deal with recursion, Tenzer and Stevens add a rather complicated recursion mechanism to FSMs. It is not clear whether this recursion mechanism is as powerful as pushdown automata and therefore could accept general context-free languages. Pradel et al. [32] discuss protocol conformance checking of APIs analogous to ours. It is designed for object-oriented programs and focuses on containers and iterators. Their approach learns protocols and statically checks them. The static protocol conformance checker may report false positives but no false negatives. It is based on an intra-procedural branch-sensitive program analysis. Hence, their approach is able to take into account data flow but has a rather imprecise abstraction of recursion, since the latter would require a context-sensitive interprocedural program analysis. Furthermore, [32] does not discuss exception handling and concurrency.

Zimmermann and Schaarschmidt [40] show that if the behaviour is a context-free language due to recursive callbacks, finite state approaches may lead to false positives. Furthermore, they introduced an approach for protocol conformance checking based on context-free systems. Exception handling would require pushdown systems. Lin et al. [24] use pushdown systems and discuss an approach of adaptation to protocols based on pushdown systems. All these works are for sequential systems.

Schmidt et al. [37] propose an approach for protocol checking of concurrent component-based systems. Their approach is also FSM-based. Thus, it is also unable to deal with recursive callbacks. Both and Zimmermann [4,5] use the restricted class of (P, G)-PRS. Thus, it allows the adequate modelling of unbound recursion, unbound concurrency, and explicit synchronizations. However, exceptions are not considered in these works.

An alternative approach for an investigation of protocol conformance is the use of process-algebras such as CSP (e.g., [2]). These approaches are more powerful than FSMs and context-free grammars. However, mechanized checking requires some restrictions on the specification language. For example, [2] uses a subset of CSP that allows only the specification of finite processes. At the end the conformance checking is reduced to checking FSMs similar to [37]. In [30], behavioural protocol conformance is used to describe a problem similar to ours. In contrast to our approach the developer has to define not only the allowed receivable calls but also the calls of the component. This approach cannot handle recursive callbacks, since the verification is reduced to finite state model. Many works use process-algebras as abstractions for the formal (behavioural) analysis of e.g. BPEL applications.

[16] uses CSP, while [31,36] use CCS-Process-algebras are similar to (P, G)-PRSs. These two works do not verify the behaviour in our sense. To the best of our knowledge, we are not aware of works in protocol conformance checking taking into account unbound recursion, unbound concurrency, and exception handling.

Other works such as [8] use another notion of behavioural conformance as this article. Their notion of conformance basically implies absence of deadlocks and livelocks, i.e., they want to reach a desired state. In contrast, protocols in this article specifies sequences of operation calls that must be satisfied, i.e., it is more a safety condition rather than a liveness condition. Furthermore, [8] does not abstract the service behaviour from an implementation. The latter is done by [33] who abstracts the service implementation to a ZING model. They check also a kind of absence of deadlocks as [8] using a simulation relation.

Bouajjani and Emmi [7] discuss the analysis of recursive parallel programs. They restrict themselves to finite data types and explore the decidability of problems such as e.g. reachability. It seems that there model is slightly more general as there are situations where the reachability problem becomes undecidable. Their approach doesn't consider exception handling.

de Caso et al. [12] abstract contracts to protocols (modelled as finite state machines) in the sense of this article and validates them. This is done for both, the client and the server and a simulation, bi-simulation or protocol conformance can be checked automatically. Abstractions from implementations are not considered. Dumez et al. [14] derive a composition specification from interaction protocols and derives an implementation of composed services satisfying the specification. Ghezzi et al. [17] discuss a dynamic approach for protocol conformance checking.

6 Conclusions

This article extends our previous work of protocol conformance checking towards exception handling. The approach is capable to represent exception handling even via service interactions. The abstractions are computed using an automatic translation. A more rugged composition of SOAs is now possible. In contrast to our previous work, the most general class of process rewrite systems is needed for modelling exception handling, unbound recursion, unbound concurrency, and explicit synchronization. Table 1 shows an interesting correspondence between the rule classifications according to Mayr and the adequate modelling of programming language concepts. In particular, it shows what is required for modelling the language concepts if one abstracts completely from data. Thus, we have the correspondence between Mayr's hierarchy of process rewrite systems and programming language concepts shown in Fig. 13. In our previous work, we had a correspondence to the class (P, G)-PRS (Process Algebra Nets). With exception handling, we have a correspondence to the (G, G)-PRS, the general class of process rewrite systems.

The reachability problem is decidable for each class of PRS while the inclusion problem to regular languages becomes undecidable in any class containing a G, i.e., that includes parallel as well as sequential composition. In a similar way as [5], we have defined a Combined Abstraction that approximates the inclusion problem by a reachability problem such that the approximation is exact iff the process rewrite system belongs to a decidable class. The reachability problem can be solved by the algorithm in [26].

However, this algorithm requires exponential space (and therefore at least exponential time) in the worst case since the reachability problem for process rewrite systems is EXPSPACE-hard [26]. It is subject to future work to check whether the worst-case behaviour practically occurs and to apply some heuristics to get it more efficient if necessary. For the latter, the same ideas as in [3,4,6] may apply. In particular, [6] has shown that protocol conformance checking of complex systems is possible in acceptable time.

Taking into account data is a challenge: the data types of variables must be abstracted to finite domains. However, this leads to a severe state explosion problem as in classical model checking. Thus, in order to consider data in protocol conformance checking, a more goal-oriented abstraction is required.

Acknowledgments We thank the anonymous referees for their remarks and suggestions. They were helpful to considerably improve the article.

Table 1 Rule classes and programming language concepts

Rule	Language concept	Rule class
$q \xrightarrow{\alpha} q'$	Internal state transition	(1, 1)
$q \xrightarrow{\alpha} q'.q''$	Synchronous procedure call	$(1, S)$
$q \xrightarrow{\alpha} \varepsilon$	Regular procedure return	(1, 1)
$q \xrightarrow{\alpha} q' \parallel q''$	Asynchronous procedure call	$(1, P)$
$q \parallel q' \xrightarrow{\alpha} q''$	Synchronization	(P, P)
$q.q' \xrightarrow{\alpha} q''.\bar{q}$	Exception handling	(S, S)
$q.q' \xrightarrow{\alpha} q''$	Exceptional procedure return	(S, S)

α is a function symbol or empty

Appendix A: Proof of Theorem 1

This appendix contains the complete proof of Theorem 1 and the formalization of the notion of top-of-stack elements. Before Theorem 1 is being proven, we have to introduce some notions and properties of these notions to be used in the proof. The first subsection discusses properties of process-algebraic expressions in general and shows formally how process rewrite rules are applied. The second subsection discusses properties of the Combined Abstraction K of PRS U_s and a protocol A_s. In particular, process-algebraic expressions of K are related to process-algebraic expressions of U_s

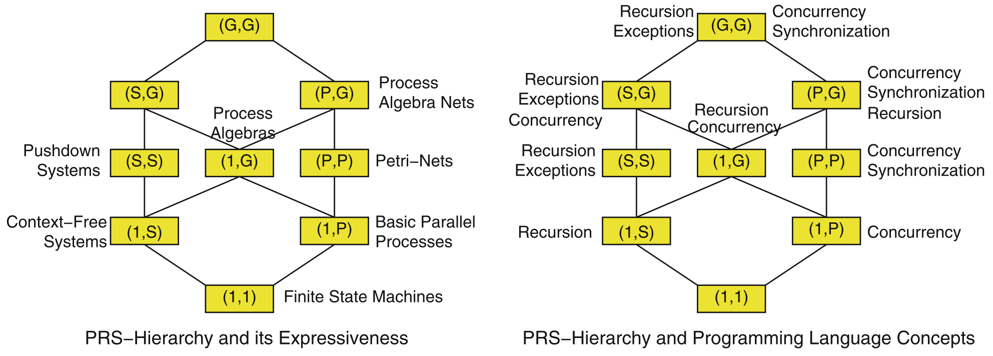


Fig. 13 PRS-Hierarchy and expressiveness w.r.t. programming language concepts **a** PRS-Hierarchy and its expressiveness **b** PRS-Hierarchy and programming language concepts

$[]$	empty list
$[c_1, \ldots, c_n]$	list with elements $c_1, \ldots, c_n$
$o +\!\!+ o'$	concatenation of o and o'
$c : o$	$\triangleq [c] +\!\!+ o$
$o \sqsubseteq o'$	prefix relation on lists (i.e. there is a $\bar{o}$ such taht $o +\!\!+ \bar{o} = o'$.
$o \sqsubset o'$	proper prefix relation (excludes $o = o'$)
$PREF(o)$	$\triangleq \{o' \in \{0,1\}^* : o \sqsubseteq o'\}$ is the set of all sequences with prefix o
$o : O$	$\triangleq \{(o : o') : o' \in O\}$ for $O \subseteq \{0,1\}^*$
$o +\!\!+ O$	$\triangleq \{o +\!\!+ o' : o' \in O\}$ for $O \subseteq \{0,1\}^*$

Fig. 14 Some notions on sequences

and protocol states of A_s. The last subsection contains the proof of Theorem 1.

A.1 Properties of process-algebraic expressions

Let Q be a set of atomic processes and $e \in PEX(Q)$ be a process-algebraic expression over Q. Although the operators . and $\|$ are associative, we assume for the purpose of this appendix a left associative bracketing. Furthermore, we assume that expressions containing the empty process are being simplified.

An *occurrence* is a finite sequence $o \in \{0, 1\}^*$. Figure 14 shows some notations on occurrences. $+\!\!+$ is an associative operator with identity $[]$.

Occurrences are used to navigate in process-algebraic expressions, to denote specific sub-expressions, and to replace specific sub-expressions by other sub-expressions. The following definitions are the usual definitions used for general terms over a given signature and are specialized to process-algebraic expressions.

Definition 1 (*Navigation by Occurrences*) Let Q be a set of atomic expressions and $\perp \notin Q$ (it represents the symbol for *undefined*). The *subexpression* of $e \in PEX(Q)$ at occurrence $o \in \{0, 1\}^*$ is a process-algebraic expression $e[o] \in PEX(Q)$ inductively defined as follows:

(i) $\varepsilon[o] \triangleq \perp$ and $q[o] \triangleq \perp$ for $q \in Q$, $o \neq []$
(ii) $e[[]] \triangleq e$ for $e \neq \varepsilon$
(iii) $(e_1 \circ e_2)[0 : o] \triangleq e_1[o]$ for $e_1, e_2 \neq \varepsilon$, $\circ \in \{., \|\}$
(iv) $(e_1 \circ e_2)[1 : o] \triangleq e_2[o]$ for $e_1, e_2 \neq \varepsilon$, $\circ \in \{., \|\}$

Definition 2 (*Replacement*) Let Q be a set of atomic expressions, $e \in PEX(Q)$ and $o \in \{0, 1\}^*$ be such that $e[o] \neq \perp$. The *replacement of e at o by $e' \in PEX(Q)$* is a process-algebraic expression $e[e'/o] \in PEX(Q)$ inductively defined by:

(i) $e[e'/[]] \triangleq e'$
(ii) $(e_1 \circ e_2)[e'/0 : o] \triangleq e_1[e'/o] \circ e_2$ for $e_1, e_2 \neq \varepsilon$, $\circ \in \{., \|\}$
(iii) $(e_1 \circ e_2)[1 : o] \triangleq e_1 \circ e_2[e'/o]$ for $e_1, e_2 \neq \varepsilon$, $\circ \in \{., \|\}$

Example 6 (Occurrences) Let $e \triangleq (((q_{28} \| q_{23}).q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$. Then, $e[[0, 0, 0, 0, 0, 0, 0]] = q_{28}$, $e[[0, 0, 0, 0, 0, 0, 1]] = q_{21}$, $e[[0, 0, 1]] = q_{\mathsf{Exc}_1}$, $e[[1, 1]] = \perp$, $e[[0, 0, 0]] = (q_{28} \| q_{21}).q_{32}.q_9)$, and $e[[0, 0, 0, 0, 0, 0]] = q_{28} \| q_{23}$. Furthermore $e[q_{29}/[0, 0, 0, 0, 0, 0]) = ((q_{29}.q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$.

Figure 15 shows the expression tree of the expression in Example 6 and the result of the replacement. In general, an occurrence o defines a path in $e \in PEX(Q)$ from the root to a sub-tree, $e[o]$ is the expression corresponding to this sub-tree and $e[e'/o]$ replaces this subtree by the tree corresponding to expression e'.

Informally, Proposition 1(i) states that if e is replaced by e' at o then each occurrence $\bar{o}$ with prefix o refers to a subexpression of e', in particular to those where o' is the suffix obtained by removing $\bar{o}$ from o. Note that o refers to the root of e'. Proposition 1(ii) states that replacing twice an expression by a sub-expression, the last replacement overrides the first replacement. Proposition 1(iii) states that if an expression is replaced at occurrence o the sub-expression of e at o, then e remains unchanged. Proposition 1(iv) states that a sub-expression at an occurrence o' is independent of a replacement if o' does not refer to a sub-tree of o. Note that o' refers to a sub-tree of $e[o]$ iff $o \sqsubseteq o'$. Proposition 1(v) states for this case, the replacements can be exchanged.

Proposition 1 (Properties of Occurrences and Replacements) *Let Q be a set of atomic processes, $e, e', e'' \in PEX(Q)$ be process-algebraic expressions over Q, and $o, o' \in \{0, 1\}^*$ such that $e[o] \neq \perp$ and $e[o'] \neq \perp$. Then, the following properties hold:*

(i) $e[e'/o][o +\!\!+ o'] = e'[o']$. *In particular* $e[e'/o][o] = e'$.
(ii) $e[e'/o][e''/o] = e[e''/o]$
(iii) *If* $e[o] = e'$ *then* $e[e'/o] = e$
(iv) *If* $o \not\sqsubseteq o'$ *and* $o' \not\sqsubseteq o$ *then* $e[e'/o][o'] = e[o']$
(v) *If* $o \not\sqsubseteq o'$ *and* $o' \not\sqsubseteq o$ *then* $e[e'/o][/e''/o'] = e[e''/o'][e'/o]$

Proof (i) The proof is by induction on o

Case 1: $o = []$. Then,

$$\begin{aligned} e[e'/[]][[] +\!\!+ o'] &= e[e'/[]][o'] && [] \text{ is identitiy of } +\!\!+ \\ &= e'[o] && \text{by Definition 2(i)} \end{aligned}$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. Then, $e = e_1 \circ e_2$ for a $\circ \in \{., \|\}$. Hence,

$$\begin{aligned} &(e_1 \circ e_2)[e'/0 : \bar{o}][(0 : \bar{o}) +\!\!+ o'] \\ &= (e_1 \circ e_2)[e'/0 : \bar{o}][0 : (\bar{o} +\!\!+ o')] && \text{associativity of } +\!\!+ \\ &= (e_1[e'/\bar{o}] \circ e_2)[0 : (\bar{o} +\!\!+ o')] && \text{Definition 2(ii)} \\ &= e_1[e'/\bar{o}][\bar{o} +\!\!+ o'] && \text{by Definition 1(iii)} \\ &= e'[o'] && \text{by induction hypothesis} \end{aligned}$$

Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. The proof is analogous to Case 2.

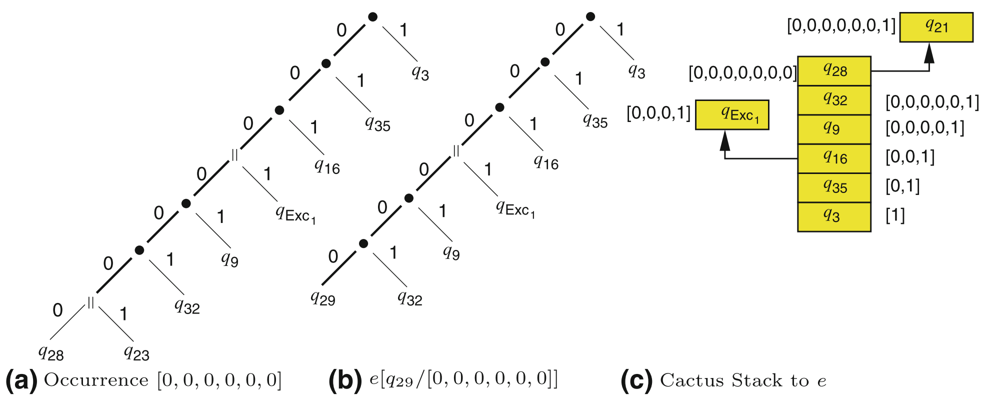


Fig. 15 Expression tree and cactus stack to $(((q_{28}\|q_{21}).q_{32}.q_9)\|q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$, an occurrence, and its replacement

(ii) The claim is proved by induction on o.

Case 1: $o = []$. Then,

$$\begin{aligned} e[e'/[]][e''/[]] &= e'' && \text{by Definition 2(i)} \\ &= e[e''/[]] && \text{by Definition 2(i)} \end{aligned}$$

Case 2: $o = 0 : \bar{o}$. Then, $e = e_1 \circ e_2$ for a $\circ \in \{., \|\}$. Hence,

$$\begin{aligned} &(e_1 \circ e_2)[e'/0 : \bar{o}][e''/0 : \bar{o}] \\ &= (e_1[e'/\bar{o}] \circ e_2)[e''/0 : \bar{o}] && \text{by Definition 2(ii)} \\ &= (e_1[e'/\bar{o}][e''/\bar{o}] \circ e_2) && \text{by Definition 2(ii)} \\ &= (e_1[e''/\bar{o}] \circ e_2) && \text{by induction hypothesis} \\ &= (e_1 \circ e_2)[e''/0 : \bar{o}] && \text{by Definition 2(ii)} \end{aligned}$$

Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. The proof is analogous to Case 2.

(iii) Let be $e[o] = e'$. Then, it must be shown that $e[e'/o] = e$. The proof is also an induction on o.

Case 1: $o = []$. Then, it holds

$$\begin{aligned} e[e'/[]] &= e' && \text{by Definition 2(i)} \\ &= e[[]] && \text{since } e[o] = e' \\ &= e && \text{by Definition 1(ii)} \end{aligned}$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. Then, $e = e_1 \circ e_2$ for a $\circ \in \{., \|\}$. Hence,

$$\begin{aligned} e_1[\bar{o}] &= (e_1.e_2)[0 : \bar{o}] && \text{by Definition 1(iii)} \\ &= e' && \text{since } e[o] = e'. \end{aligned}$$

Thus, by induction hypothesis it is $e_1[e'/\bar{o}] = e_1$. Then, it holds:

$$\begin{aligned} (e_1 \circ e_2)[e'/0 : \bar{o}] &= (e_1[e'/\bar{o}] \circ e_2) && \text{by Definition 2(ii)} \\ &= e_1 \circ e_2 && \text{by induction hypothesis} \end{aligned}$$

Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. The proof is analogous to Case 2.

(iv) The proof is by induction on the longest common prefix of o and o'.

Case 1: $[]$ is the longest common prefix of o and o'. Then, it is $o = [c_1 : \bar{o}]$ and $o' = [c_2 : \bar{\hat{o}}]$ for $c_1, c_2 \in \{0, 1\}$, $c_1 \neq c_2$, $\bar{o}, \hat{o} \in \{0, 1\}^*$. Since $e[o] \neq \bot$, it is $e = e_1 \circ e_2$ for a $\circ \in \{., \|\}$. Let be $c_1 = 0$ and $c_2 = 1$ (the case $c_1 = 1$ and $c_2 = 0$ is proven analogously). Then,

$$\begin{aligned} &(e_1 \circ e_2)[e'/0 : \bar{o}][1 : \hat{o}] \\ &= (e_1[e'/\bar{o}] \circ e_2)[1 : \hat{o}] && \text{by Definition 2(ii)} \\ &= e_2[\hat{o}] && \text{by Definition 1(iv)} \\ &= (e_1 \circ e_2)[1 : \hat{o}] && \text{by Definition 1(iv)} \end{aligned}$$

Case 2: The longest common prefix of o is $c : o''$ for a $o'' \in \{0, 1\}^*$. Then, it is $o = [c : \bar{o}]$ and $o' = [c : \bar{\hat{o}}]$, $\bar{o}, \hat{o} \in \{0, 1\}^*$ and o'' is the longest common prefix of $\bar{o}$ and $\hat{o}$. We prove the case $c = 0$ (the case $c = 1$ is proven analogously):

$$\begin{aligned} &(e_1 \circ e_2)[e'/0 : \bar{o}][0 : \hat{o}] \\ &= (e_1[e'/\bar{o}] \circ e_2)[0 : \hat{o}] && \text{by Definition 2(ii)} \\ &= e_1[e'/\bar{o}][\hat{o}] && \text{by Definition 1(iii)} \\ &= e_1[\hat{o}] && \text{by induction hypothesis} \\ &= (e_1 \circ e_2)[0 : \hat{o}] && \text{by Definition 1(iii)} \end{aligned}$$

(v) The proof is also by induction on the longest common prefix of o and o'.

Case 1: $[]$ is the longest common prefix of o and o'. Then, it is $o = [c_1 : \bar{o}]$ and $o' = [c_2 : \bar{\hat{o}}]$ for $c_1, c_2 \in \{0, 1\}$, $c_1 \neq c_2$, $\bar{o}, \hat{o} \in \{0, 1\}^*$. Since $e[o] \neq \bot$, it is $e = e_1 \circ e_2$ for a $\circ \in \{., \|\}$. Let be $c_1 = 0$ and $c_2 = 1$ (the case $c_1 = 1$ and $c_2 = 0$ is proven analogously). Then,

$$
\begin{aligned}
&(e_1 \circ e_2)[e'/0 : \bar{o}][e''/1 : \hat{o}] \\
&\quad = (e_1[e'/\bar{o}] \circ e_2)[e''/1 : \hat{o}] && \text{by Definition 2(ii)} \\
&\quad = (e_1[e'/\bar{o}] \circ e_2[e''/\hat{o}]) && \text{by Definition 2(iii)} \\
&\quad = (e_1 \circ e_2[e''/\hat{o}])[e'/0 : \bar{o}] && \text{by Definition 2(ii)} \\
&\quad = (e_1 \circ e_2)[e''/1 : \hat{o}][e'/0 : \bar{o}] && \text{by Definition 2(iii)}
\end{aligned}
$$

Case 2: The longest common prefix of o is $c : o''$ for a $o'' \in \{0, 1\}^*$. Then, $o = [c : \bar{o}]$ and $o' = [c : \bar{\hat{o}}]\,\bar{o}$, $\hat{o} \in \{0, 1\}^*$ and o'' is the longest common prefix of $\bar{o}$ and $\hat{o}$. We prove the case $c = 0$ (the case $c = 1$ is proven analogously):

$$
\begin{aligned}
&(e_1 \circ e_2)[e'/0 : \bar{o}][e''/0 : \hat{o}] \\
&\quad = (e_1[e'/\bar{o}] \circ e_2)[e''/0 : \hat{o}] && \text{by Definition 2(ii)} \\
&\quad = e_1[e'/\bar{o}][e''/\hat{o}] \circ e_2 && \text{by Definition 2(ii)} \\
&\quad = e_1[e''/\hat{o}][e'/\bar{o}] \circ e_2 && \text{by induction hypothesis} \\
&\quad = (e_1[e''/\hat{o}] \circ e_2)[e'/0 : \bar{o}] && \text{by Definition 2(ii)} \\
&\quad = (e_1 \circ e_2)[e''/0 : \hat{o}][e'/0 : \bar{o}] && \text{by Definition 2(ii)}
\end{aligned}
$$

□

The following definition defines for process-algebraic expressions the occurrences for the top-of-stack elements of the corresponding cactus stacks, respectively.

Definition 3 (*Top-of-Stack Occurrences*) Let Q be a set of atomic processes and $e \in PEX(Q)$. The set $TOP(e)$ of *top-of-stack occurrences of* e is inductively defined by:

(i) $TOP(\varepsilon) \triangleq \emptyset$
(ii) $TOP(q) \triangleq \{[]\}$ for $q \in Q$
(iii) $TOP(e_1.e_2) \triangleq 0 : TOP(e_1)$
(iv) $TOP(e_1 \| e_2) \triangleq (0 : TOP(e_1)) \cup (1 : TOP(e_2))$

Example 7 (Top-of-Stack Elements) For the process-algebraic expression $e \triangleq (((q_{28} \| q_{23}).q_{32}.q_9) \| q_{\mathsf{Exc}_1}).q_{16}.q_{35}.q_3$, it is

$$
\begin{aligned}
&TOP(e) = \\
&\quad \{[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 1]\}.
\end{aligned}
$$

This expression corresponds to the cactus stack in Fig. 15c.

Proposition 2 defines some properties on top-of-stack elements. The first property states that if an expression e is replaced at a occurrence o by a sub-expression e' and $e[o]$ contains top-of-stack elements, a top-of-stack element of $e[e'/o]$ is a top-of-stack element of e outside of $e[o]$ or a top-of-stack element of e' (adjusted for $e[/e'/o]$). The second property states that if $e[o]$ doesn't contain top-of-stack elements then the top-of-stack elements of e are not affected by replacements at o. Furthermore, each top-of-stack elements refers to an atomic expression and two top-of-stack elements cannot be proper sub-expressions of each other.

Proposition 2 (Properties of Top-Of-Stack Elements) *Let Q be a set of atomic processes, $e, e' \in PEX(Q)$ be process-algebraic expressions over Q, and $o, o' \in \{0, 1\}^*$ be occurrences. Then, the following properties hold:*

(i) If $e[o] \neq \bot$ and $TOP(e) \cap PREF(o) \neq \emptyset$ then $TOP(e[e'/o]) = TOP(e) \setminus PREF(o) \cup (o +\!\!+ TOP(e'))$.
(ii) If $e[o] \neq \bot$ and $TOP(e) \cap PREF(o) = \emptyset$ then $TOP(e[e'/o]) = TOP(e)$.
(iii) For each $o \in TOP(e)$ it is $e[o] \in Q$
(iv) If $o \sqsubseteq o'$ for a $o' \in TOP(e)$ then $e[o] \neq \bot$.
(v) If $o, o' \in TOP(e)$ and $o \neq o'$, then $o \not\sqsubseteq o'$ and $o' \not\sqsubseteq e$.

Proof (i) The proof is by induction on o.

Case 1: $o = []$. Then, $e \neq \varepsilon$ since otherwise $e[o] = \bot$. Furthermore, it holds $PREF([]) = \{0, 1\}^*$. Then,

$$
\begin{aligned}
&TOP(e) \setminus PREF([]) \cup ([] +\!\!+ TOP(e')) \\
&\quad = TOP(e) \setminus \{0, 1\}^* \cup TOP(e') && \text{by definition of } +\!\!+ \text{ for sets} \\
&\quad = \emptyset \cup TOP(e') \\
&\quad = TOP(e') = TOP(e[e'/[]]) && \text{by Definition 2(i)}
\end{aligned}
$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. Then, $e = e_1.e_2$ or $e = e_1 \| e_2$. Otherwise it would be $e[o] = \bot$. By induction hypothesis it holds

$$TOP(e_1[e'/\bar{o}]) = TOP(e_1) \setminus PREF(\bar{o}) \cup (\bar{o} +\!\!+ TOP(e')) \tag{1}$$

and

$$PREF(0 : \bar{o}) = 0 : PREF(\bar{o}) \tag{2}$$

Case 2.1: $e = e_1.e_2$. Then,

$$
\begin{aligned}
&TOP((e_1.e_2)[e'/0 : \bar{o}]) \\
&\quad = TOP(e_1[e'\bar{o}].e_2) && \text{by Definition 2(ii)} \\
&\quad = 0 : TOP(e_1[e'/\bar{o}]) && \text{by Definition 3(iii)} \\
&\quad = 0 : (TOP(e_1) \setminus PREF(\bar{o}) \cup (\bar{o} +\!\!+ TOP(e')) && \text{by (1)} \\
&\quad = (0 : TOP(e_1)) \setminus (0 : PREF(\bar{o})) \cup (0 : (\bar{o} +\!\!+ TOP(e')) \\
&\quad = (0 : TOP(e_1)) \setminus (PREF(0 : \bar{o})) \cup (0 : (\bar{o} +\!\!+ TOP(e')) && \text{by (2)} \\
&\quad = (0 : TOP(e_1)) \setminus (PREF(0 : \bar{o})) \cup ((0 : \bar{o}) +\!\!+ TOP(e')) && \text{by associativity of } +\!\!+ \\
&\quad = TOP(e_1.e_2) \setminus (PREF(0 : \bar{o})) \cup ((0 : \bar{o}) +\!\!+ TOP(e')) && \text{by Definition 3(iii)}
\end{aligned}
$$

Case 2.2: $e = e_1 \| e_2$. Then,

$$\begin{aligned}
&TOP((e_1\|e_2)[e'/0:\bar{o}]) \\
&\quad = TOP(e_1[e'\|o].e_2) && \text{by Definition 2(ii)} \\
&\quad = 0 : TOP(e_1[e'/\bar{o}]) \cup (1 : TOP(e_2)) && \text{by Definition 3(iii)} \\
&\quad = (0 : TOP(e_1)) \setminus (PREF(0:\bar{o})) \cup ((0:\bar{o}) +\!\!+ TOP(e')) \cup (1 : TOP(e_2)) && \text{analogous to Case 2.1} \\
&\quad = ((0 : TOP(e_1)) \cup (1 : TOP(e_2)) \setminus (PREF(0:\bar{o})) \cup ((0:\bar{o}) +\!\!+ TOP(e')) \\
&\quad = TOP(e_1\|e_2) \setminus (PREF(0:\bar{o})) \cup ((0:\bar{o}) +\!\!+ TOP(e')) && \text{by Definition 3(iii)}
\end{aligned}$$

Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$. Then, $e = e_1 \| e_2$. Otherwise it would be $e[o] = \bot$ or $TOP(e) \cap PREF(1:\bar{o}) = \emptyset$. The case is proven analogously to Case 2.2

(ii) For the case $o = []$, it always holds $TOP(e) \cap PREF([]) = TOP(e) \neq \emptyset$. Similarly, if $o = 0 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$ and for $e = e_1 \circ e_2$, $\circ \in \{.\ \|\}$ it holds $TOP(e) \supseteq 0 : TOP(e_1) \neq \emptyset$. If $e = e_1\|e_2$, it holds $TOP(e) \supseteq 1 : TOP(e_2) \neq \emptyset$. Hence, it remains $e = e_1.e_2$ and $o = 1 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$. Then, it holds

$$\begin{aligned}
TOP((e_1.e_2)[e'/1:\bar{o}]) &= TOP(e_1.e_2[\bar{o}]) && \text{by Definition 2(iii)} \\
&= 0 : TOP(e_1) && \text{by Definition 3(iii)} \\
&= TOP(e_1.e_2) && \text{by Definition 3(iii)}
\end{aligned}$$

(iii) Suppose $e[o] \notin Q$ for a $o \in TOP(e)$. Then, $e = e_1.e_2$ or $e = e_1\|e_2$. We prove by induction on o that $o \notin TOP(e)$.

Case 1: $o = []$: Then, $e[o] = e$.
Case 1.1: $e = e_1.e_2$. Then, $TOP(e) = 0 : TOP(e_1) \not\ni []$.
Case 1.2: $e = e_1\|e_2$. Then, $TOP(e) = (0 : TOP(e_1)) \cup (1 : TOP(e_2) \not\ni []$.
Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$.
Case 2.1: $e = e_1.e_2$. Then, $e[o] = e_1[\bar{o}]$. By induction hypothesis $\bar{o} \notin TOP(e_1)$. Hence, $0 : \bar{o} \notin 0 : TOP(e_1) = TOP(e_1.e_2)$
Case 2.2: $e = e_1 \| e_2$. By induction hypothesis $\bar{o} \notin TOP(e_1)$. Hence, it is $0 : \bar{o} \notin (0 : TOP(e_1)) \cup (1 : TOP(e_2)) = TOP(e_1.e_2)$
Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$. Hence, $1 : \bar{o} \notin 0 : TOP(e_1) = TOP(e_1.e_2)$
Case 3.1: $e = e_1.e_2$. Hence, $1 : \bar{o} \notin 0 : TOP(e_1) = TOP(e_1.e_2)$
Case 3.2: $e = e_1\|e_2$. The proof is analogous to the proof of Case 2.2.

(iv) By (iii), $e[o] \neq \bot$ for a $o \in TOP(e)$. Hence, by Definition 1(i), $e[o'] \neq \bot$ for each *prefix* $o' \sqsubseteq o$.
(v) Let $o \sqsubset o'$ for a $o' \in TOP(e)$. By (iii): $e[o'] \in Q$. Hence, by Definition 1(ii) $e[o] \notin Q$ is a composed expression. □

The following Lemma states that the application of a process rewrite rule $l \rightarrow r$ to a process-algebraic expression replaces l by r at an occurrence o where $e[o]$ contains TOP-of-stack elements.

Lemma 1 (Application of Process Rewrite Rules) *Let $\Pi \triangleq (\Sigma, Q, \rightarrow, q_0, F)$ be a process rewrite system, $e \in PEX(Q)$ be a process-algebraic expression, $l \overset{a}{\rightarrow} r$ be a process rewrite rule, and $o \in \{0,1\}^*$ be an occurrence with $TOP(e) \cap PREF(o) \neq \emptyset$ and $e[o] = l$. Then, it holds $e \overset{a}{\Rightarrow} e[r/o]$.*

Proof The proof is an induction on o.

Case 1: $o = []$. Then,

$$\begin{aligned}
e[[]] &= l && \text{by assumption} \\
&\overset{a}{\Rightarrow} r && \text{by inference rule (R)} \\
&= e[r/[]] && \text{by Definition 2(i)}
\end{aligned}$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$. Then, $e = e_1 \circ e_2$ for a $\circ \in \{.,\|\}$. Then, it holds

$$\begin{aligned}
e_1[\bar{o}] &= e[0:\bar{o}] && \text{by Definition 1(iii)} \\
&= l && \text{by assumption}
\end{aligned}$$

Hence, by induction hypothesis, it is

$$e_1 \overset{a}{\Rightarrow} e_1[r/\bar{o}] \tag{3}$$

Case 2.1: $e = e_1.e_2$. Then,

$$\begin{aligned}
e_1.e_2 &\overset{a}{\Rightarrow} e_1[r/\bar{o}].e_2 && \text{by (3) and inference rule (S)} \\
&= (e_1.e_2)[r/0:o] && \text{by Definition 2(ii)}
\end{aligned}$$

Case 2.2: $e = e_1\|e_2$. Then,

$$\begin{aligned}
e_1\|e_2 &\overset{a}{\Rightarrow} e_1[r/\bar{o}]\|e_2 && \text{by (3) and inference rule (P1)} \\
&= (e_1\|e_2)[r/0:o] && \text{by Definition 2(ii)}
\end{aligned}$$

Case 3: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0,1\}^*$. Then, $e = e_1\|e_2$ since $TOP(e) \cap PREF(o) = \emptyset$ for $e = e_1.e_2$. The proof for this case is analogous to Case 2.2. □

Appendix 1.2 Properties of the combined abstraction

Throughout this subsection, let $U_s \triangleq (\Sigma_s, Q, \rightarrow, q_0, F)$ be a process rewrite system specifying an abstraction for the use of service s, $R_s \triangleq (\Sigma_s, R_s, \rightarrow_s, q_o^s, \bar{F}_s)$ be a finite state machine without λ-transitions specifying an (inverted) protocol for s, and $K \triangleq (\Sigma_s, Q_K, \rightarrow_K, q_0^K, F_K)$ be the Combined Abstraction of U_s and R_s as defined in Sect. 4.

Definition 4 (*Stripping Protocol States and Protocol States on Top of Stacks*) Let $e \in PEX(Q_K)$ a process-algebraic expression over the atomic processes of the Combined Abstraction. The *stripping of protocol state from e* is a process-algebraic expression $\pi(e) \in PEX(Q)$ over the atomic processes of U_s, inductively defined by:

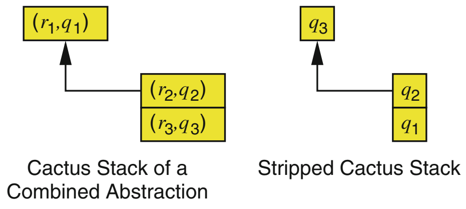
(i) $\pi(\varepsilon) \triangleq \varepsilon$ and $\pi((r, q)) \triangleq q$
(ii) $\pi(e_1 \circ e_2) \triangleq \pi(e_1) \circ \pi(e_2)$ for $\circ \in \{., \|\}$

The set $TOS(e) \triangleq \{r$: there is a$o \in TOP(e)$ and $q \in Q$ such that $e[o] = (r, q)\}$ is the set of top-of-stack protocol states of $e \in PEX(Q_K)$. The process-algebraic expression e is *inconsistent* with r iff $TOP(e) \neq \{r\}$.

The cactus stack corresponding to $\pi(e)$ has the same shape as the cactus stack corresponding e, and if a stack element of e contains (r, q) then the stack element of $\pi(e)$ contains q. The top-of-stack protocol states is the set of protocol states on the top-of-stack elements of the cactus stack corresponding to e.

Example 8 Let $e \triangleq ((r_1, q_1)\|(r_2, q_2)).(r_3.q_3)$. Then,

$$\pi(e) = (q_1\|q_2).q_3$$
$$TOS(e) = \{r_1, r_2\}$$



The following proposition states some properties on π. The first one states that stripping and navigation by occurrences can be interchanged. The second states that stripping a replacement can be interchanged by stripping the expression and the replacing expression. The third property states that the set of top-of-stack occurrences are not affected by stripping. The fourth proposition and fifth property are technical ones. They relate replacements on a stripped expression to stripping a replacement. The fifth property is a generalization of the fourth as it considers the states at the top-of-stack occurrences.

Proposition 3 (Stripping Protocol States) *Let $e', e'', e''' \in PEX(Q_K)$ process-algebraic expressions over the atomic processes of the Combined Abstraction of U_s and R_s, $e, e_1, e_2 \in PEX(Q)$ be process-algebraic expressions over the atomic states of U_s, and $o \in \{0, 1\}^*$ be an occurrence such that $e'[o] \neq \bot$ (and $e[o] \neq \bot$). Then, it holds:*

(i) $\pi(e'[o]) = \pi(e)[o]$
(ii) $\pi(e'[e''/o]) = \pi(e')[\pi(e'')/o]$
(iii) $TOP(\pi(e')) = TOP(e')$
(iv) If $e[o] = e_1$, $\pi(e'') = e[e_2/o]$, $\pi(e''') = e_1$, and $e' = e''[e'''/o]$, then $\pi(e') = e$
(v) Let $o_1, \ldots, o_k \in TOP(e)$, $o \notin \{o_1, \ldots, o_k\}$ and $q_i \triangleq e[o_i]$, $i = 1, \ldots, k$. Then, it holds for any protocol states $r_1, \ldots, r_k \in R_s$: If $e[o] = e_1$, $\pi(e'') = e[e_2/o]$, $\pi(e''') = e_1$, and $e' = e''[(r_1, q_1)/o_1] \cdots [(r_k, q_k)/o_k][e'''/o]$, then $\pi(e') = e$

Proof (i) The proof is an induction on o.

Case 1: $o = []$. Then, by Definition 1(ii), it holds

$$\pi(e'[[]]) = \pi(e') = \pi(e')[[]].$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. Then, $e' = \bar{e}_1 \circ \bar{e}_2$ for a $\circ \in \{., \|\}$. Hence,

$$\begin{aligned} &\pi((\bar{e}_1 \circ \bar{e}_2)[0 : \bar{o}]) \\ &= \pi(\bar{e}_1[\bar{o}]) && \text{by Definition 1(iii)} \\ &= \pi(\bar{e}_1)[\bar{o}] && \text{by induction hypothesis} \\ &= (\pi(\bar{e}_1) \circ \pi(\bar{e}_2))[0 : \bar{o}] && \text{by Definition 1(iii)} \\ &= \pi(\bar{e}_1 \circ \bar{e}_2)[0 : \bar{o}] && \text{by Definition 4(ii)} \end{aligned}$$

Case 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. The proof is analogous to Case 2.

(ii) The proof is an induction on o.

Case 1: $o = []$. Then, by Definition 2(i), it holds

$$\pi(e'[e''/[]]) = \pi(e'') = \pi(e')[\pi(e'')/[]].$$

Case 2: $o = 0 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. Then, $e' = \bar{e}_1 \circ \bar{e}_2$ for a $\circ \in \{., \|\}$. Hence,

$$\begin{aligned} &\pi((\bar{e}_1 \circ \bar{e}_2)[e''/0 : \bar{o}]) \\ &= (\pi(\bar{e}_1) \circ \pi(\bar{e}_2))[e''/0 : \bar{o}] && \text{by Definition 4(ii)} \\ &= \pi(\bar{e}_1)[e''/\bar{o}] \circ \pi(\bar{e}_2) && \text{by Definition 2(ii)} \\ &= \pi(\bar{e}_1)[\pi(e'')/\bar{o}] \circ \pi(\bar{e}_2) && \text{by induction hypothesis} \\ &= (\pi(\bar{e}_1) \circ \pi(\bar{e}_2))[\pi(e'')/0 : \bar{o}] && \text{by Definition 2(ii)} \\ &= \pi(\bar{e}_1 \circ \bar{e}_2)[\pi(e'')/0 : \bar{o}] && \text{by Definition 4(ii)} \end{aligned}$$

CASE 3: $o = 1 : \bar{o}$ for a $\bar{o} \in \{0, 1\}^*$. The proof is analogous to Case 2.

(iii) The proof is by induction on e'.

CASE 1: $e' = \varepsilon$ or $e' = (r, q)$. Then, (iii) follows directly from Definition 4(i) and Definition 3(i), (ii).
CASE 2: $e' = \bar{e}_1.\bar{e}_2$. Then,

$$\begin{aligned}
&TOP(\pi(\bar{e}_1.\bar{e}_2)) \\
&\quad = TOP(\pi(\bar{e}_1).\pi(\bar{e}_2)) && \text{by Definition 4(ii)} \\
&\quad = TOP(\pi(\bar{e}_1)) && \text{by Definition 3(iii)} \\
&\quad = TOP(e_1) && \text{by induction hypothesis} \\
&\quad = TOP(e_1.e_2) && \text{by Definition 3(iii)}
\end{aligned}$$

CASE 3: $e' = \bar{e}_1 \| \bar{e}_2$. Then,

$$\begin{aligned}
&TOP(\pi(\bar{e}_1 \| \bar{e}_2)) \\
&\quad = TOP(\pi(\bar{e}_1) \| \pi(\bar{e}_2)) && \text{by Definition 4(ii)} \\
&\quad = TOP(\pi(\bar{e}_1)) \cup TOP(\pi(\bar{e}_2)) && \text{by Definition 3(iv)} \\
&\quad = TOP(e_1) \cup TOP(e_2) && \text{by induction hypothesis} \\
&\quad = TOP(e_1 \| e_2) && \text{by Definition 3(iv)}
\end{aligned}$$

(v)

$$\begin{aligned}
\pi(e') &= \pi(e''[e'''/o]) && \text{since } e' = e''[e'''/o] \\
&= \pi(e'')[\pi(e''')/o] && \text{by (i)} \\
&= e[e_2/o][e_1/o] && \text{since } \pi(e'') = e[e_2/o] \text{ and } \pi(e''') = e_1 \\
&= e[e_1/o] && \text{by Proposition 1(ii)} \\
&= e && \text{by Proposition 1(iii) and } e[o] = e_1
\end{aligned}$$

(iv)

$$\begin{aligned}
&\pi(e') \\
&\quad = \pi(e''[(r_1, q_1)/o_1] \cdots [(r_k, q_k)/o_k][e'''/o]) && \text{by assumption on } e' \\
&\quad = \pi(e'')[q_1/o_1] \cdots [q_k/o_k][\pi(e''')/o] && \text{by (i)} \\
&\quad = e[e_2/o][q_1/o_1] \cdots [q_k/o_k][e_1/o] && \text{since } \pi(e'') = e[e_2/o] \text{ and } \pi(e''') = e_1 \\
&\quad = e[q_1/o_1] \cdots [q_k/o_k][e_2/o][e_1/o] && \text{by Proposition 1(v)} \\
&\quad = e[e_2/o][e_1/o] && \text{by Proposition 1(iii) and } e[o_i] = q_i \\
&\quad = e[e_1/o] && \text{by Proposition 1(ii)} \\
&\quad = e && \text{by Proposition 1(iii) and } e[o] = e_1
\end{aligned}$$

□

The following properties for the top-of-stack protocol states are direct consequence of Proposition 2. The first property states that an atomic process of the Combined Abstraction can be deduced from the top-of-stack elements and the stripped process-algebraic expression. The second states that a protocol state is a top-of-stack protocol state in $e[e'/o] \in PEX(Q_K)$ if it is a top-of-stack protocol state outside of o or it is a top-of-stack protocol state of e'. The third and fourth property are special cases of the second.

Proposition 4 (Top-Of-Stack Protocol States) *Let $e, e' \in PEX(Q_K)$ be process-algebraic expressions over the atomic processes of the Combined Abstraction, $o \in \{0, 1\}^*$ be an occurrence with $e[o] \neq \bot$, and $o_1, \ldots, o_k \in \{0, 1\}^*$ be occurrence such that $TOP(e) \setminus PREF(e) = \{o_1, \ldots, o_k\}$. Then,*

(i) If $TOS(e) = r$, $o \in TOP(e)$ then $e[o] = (r, q)$ where $q = \pi(e)[o] \in Q$.
(ii) Let be $(r_i, q_i) \triangleq e[o_i]$, $i = 1, \ldots, k$. Then, $TOS(e[e'/o]) = \{r_1, \ldots, r_k\} \cup TOS(e')$.
(iii) If there is a $r \in R_s$ and $q_1, \ldots, q_k \in Q$ such that $e[o_i] = (r, q_i)$, $i = 1, \ldots, k$ and $TOS(e') = \{r\}$ then $TOS(e[e'/o]) = \{r\}$.
(iv) If $TOS(e') = \{r\}$ then $TOS(e[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][e'/o]) = \{r\}$ for all $q_1, \ldots, q_k \in Q$.

Proof (i) follows directly from Definitions 3 and 4. Proposition 2(i) and Definition 4 directly imply (ii). (iii) and (iv) are special cases of (ii) (just choosing $r_i = r$ for $i = 1, \ldots, k$ □

Appendix 1.3 Proof of the main theorem

For this subsection, let $U_s \triangleq (\Sigma_s, Q, \rightarrow, q_0, F)$ be a process rewrite system specifying an abstraction for the use of service s, $R_s \triangleq (\Sigma_s, R_s, \rightarrow_s, q_o^s, \bar{F}_s)$ be a finite state machine without λ-transitions specifying an (inverted) protocol for s, and $K \triangleq (\Sigma_s, Q_K, \rightarrow_K, q_0^K, F_K)$ be the Combined Abstraction of U_s and R_s as defined in Sect. 4.

The application of process rewrite rule in $T_{11} \cup T_{1S} \cup T_{S1} \cup T_{1P} \cup T_{P1}$ to a consistent $e \in PEX(Q_K)$ may change the protocol state on a top-of-stack element from a protocol state r to r'. Thus, if $e \overset{a}{\Rightarrow}_K e'$ then $TOS(e') = \{r, r'\}$ is possible, i.e., e' is inconsistent. With following lemma it is possible to rewrite e' into e'' with the same shape and $TOP(e'') = \{r'\}$.

Lemma 2 (Construction of Consistent Process-Algebraic Expressions) *Let be $e \in PEX(Q_K)$, $o \in TOP(e)$, $e[o] = (r, q)$. If $r \overset{a}{\rightarrow}_s r'$ then $e \overset{\lambda}{\Rightarrow}_K e[(r', q)/o]$.*

Proof Since $r \overset{a}{\rightarrow}_s r'$, it is $(r, q) \overset{\lambda}{\rightarrow}_K (r, q') \in T_0$. Thus, Lemma 1 implies $e \overset{\lambda}{\Rightarrow}_K e[(r', q)/o]$. □

The following Theorem is the same as Theorem 1 where (i) and (ii) are rephrased using Definition 4

Theorem 1 *Let be $e \in PEX(Q_K)$ be a process-algebraic expression over Q and $r \in R_s$ a protocol state, and $x \in \Sigma_s^*$ such that there is a $f \in F$ and $\bar{r} \in \bar{F}_s$ with $e \overset{x}{\Rightarrow} f$ and $r \overset{x}{\Rightarrow}_s \bar{r}$. Then, there is a $e' \in PEX(Q_K)$ and a $f_k \in F_K$ such that the following properties are satisfied:*

(i) $\pi(e') = e$
(ii) $TOP(e') = \{r\}$
(iii) $e' \overset{x}{\Rightarrow}_K f_k$

Proof The proof is by induction on the number of applications of process rewrite rule $\rightarrow$ of PRS U_s in $e \overset{x}{\Rightarrow} f$.
BASE CASE:: No PRS-rule is being applied. Then, $e = f$ and $r = \bar{r}$ because R_s doesn't contain rules $r \overset{\lambda}{\rightarrow}_s r'$.Thus

$r \triangleq \bar{r} \overset{\lambda}{\Rightarrow}_s \bar{r}$. By definition of the combined abstraction, it holds $f_k \triangleq (\bar{r}, f) \in F$. Define $e' \triangleq f_k$. Then, $\pi(e') = f (= e)$ by Definition 4(i), $TOS(e') = \{\bar{r}\} (= \{r\})$, and $e' = f_k \overset{\lambda}{\Rightarrow}_K f_k$, i.e., it holds (i), (ii), and (iii) if no PRS-rule has been applied.

Inductive Case: Let $lhs \overset{a}{\rightarrow} rhs$ be the first PRS-rule being applied in $e \overset{x}{\Rightarrow} f$. Let $o \in \{0, 1\}^*$ be the occurrence where it is applied. Hence, by Lemma 1 it holds:

$$e[o] = lhs \tag{4}$$

$$e \overset{a}{\Rightarrow} e[rhs/o] \overset{y}{\Rightarrow} f \tag{5}$$

$$r \overset{a}{\Rightarrow}_s r' \overset{y}{\Rightarrow} \bar{r} \tag{6}$$

$$r' \triangleq \begin{cases} r & \text{if } a = \lambda \\ \hat{r} & \text{if } r \overset{a}{\rightarrow}_s \hat{r} \end{cases} \tag{7}$$

$$x = ay \tag{8}$$

By induction hypothesis, there is a $e'' \in PEX(Q_K)$ such that

$$\pi(e'') = e[rhs/o] \tag{9}$$

$$e'' \overset{y}{\Rightarrow}_K f_k \text{ for a } f_k \in F_K \tag{10}$$

$$TOP(e'') = \begin{cases} \{r\} & \text{if } a = \lambda \\ \{\hat{r}\} & \text{if } a \in \Sigma_s \end{cases} \tag{11}$$

Case 1: $q \overset{\lambda}{\rightarrow} q'$ has been applied, i.e. $lhs = q$, $rhs = q'$, and $a = \lambda$. Define

$$e' \triangleq e''[(r, q)/o] \tag{12}$$

Then, $\pi(e') = e$ by Proposition 3(iv) and $TOS(e) = \{r\}$ by Proposition 4(iii) and (11). It holds:

$$\begin{aligned} \pi(e''[o]) &= \pi(e'')[o] && \text{by Proposition 3(i)} \\ &= e[q'/o][o] && \text{by (9)} \\ &= q && \text{by Proposition 1(i)} \end{aligned}$$

Together with (11), Proposition 4(i) implies

$$e''[o] = (r, q') \tag{13}$$

By definition of the combined abstraction, it is

$$(r, q) \overset{\lambda}{\rightarrow}_K (r, q') \in T_{11} \tag{14}$$

Hence,

$$\begin{aligned} e' &\overset{\lambda}{\Rightarrow}_K e'[(r, q')/o] && \text{by Lemma 1 and (14)} \\ &= e''[(r, q)/o][(r, q')/o] && \text{by (12)} \\ &= e''[(r, q')/o] && \text{by Proposition 1(ii)} \\ &= e'' && \text{by Proposition 1(iii) and (13)} \\ &\overset{x}{\Rightarrow}_K f_k && \text{by (10), } a = \lambda, \text{ and (8)} \end{aligned}$$

Thus (i), (ii), and (iii) hold for Case 1

Case 2: $q \overset{a}{\rightarrow} q'$, $a \in \Sigma_s$, has been applied, i.e. $lhs = q$ and $rhs = q'$. Let $o_1, \ldots, o_k$ be the top-of-stack occurrences in e outside of o, i.e.

$$TOP(e) \setminus PREF(o) = \{o_1, \ldots, o_k\} \tag{15}$$

$$q_i = e[o_i],\ i = 1, \ldots, k, \text{ and} \tag{16}$$

$$e' \triangleq e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(r, q)/o] \tag{17}$$

Then, $\pi(e') = e$ by Proposition 3(v) and $TOS(e) = \{r\}$ by Proposition 4(iv), (11), and (16). Furthermore, by definition of the combined abstraction it is

$$(r, q) \overset{a}{\rightarrow}_K (\hat{r}, q') \in T_{11} \tag{18}$$

By Proposition 1(i), it is $e[q'/o][o] = q'$. Together with (11) and Proposition 4(i), this implies

$$e''[o] = (\hat{r}, q') \tag{19}$$

Similarly, (11), (16), and Proposition 4(i) imply

$$e''[o_i] = (\hat{r}, q_i), \tag{20}$$

$i = 1, \ldots, k$. Hence,

$$\begin{aligned} e' &\overset{a}{\Rightarrow}_K e'[(\hat{r}, q')/o] && \text{by Lemma 1 and (14)} \\ &= e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(r, q)/o][(\hat{r}, q')/o] && \text{by (17)} \\ &= e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(\hat{r}, q')/o] && \text{by Proposition 1(ii)} \\ &\overset{\lambda}{\Rightarrow}_K e''[(\hat{r}, q_1)/o_1] \cdots [(\hat{r}, q_k)/o_k][(\hat{r}, q')/o] && \text{by Lemma 2} \\ &= e'' && \text{by Proposition 1(iii), (19), and (20)} \\ &\overset{y}{\Rightarrow}_K f_k && \text{by (10)} \end{aligned}$$

Hence, (i), (ii), and (iii) hold also for Case 2.

Case 3: $q \overset{\lambda}{\rightarrow} q'.q''$ has been applied, i.e. $lhs = q$, $rhs = q'.q''$, and $a = \lambda$. By (9) and Proposition 3(i) there is $r'' \in R_s$ such that

$$e''[o] = (r, q'), (r''.q'') \tag{21}$$

Define e' as in Case 1, i.e. by (12). Then, $\pi(e') = e$ by Proposition 3(iv) and $TOS(e) = \{r\}$ by Proposition 4(iii) and (11). By definition of the combined abstraction, it is

$$(r, q) \overset{\lambda}{\rightarrow}_K (r, q').(r'', q'') \in T_{1S} \tag{22}$$

Hence,

$$\begin{aligned} e' &\overset{\lambda}{\Rightarrow}_K e'[(r, q').(r'', q'')/o] && \text{by Lemma 1 and (22)} \\ &= e''[(r, q)/o][(r, q').(r'', q'')/o] && \text{by (12)} \\ &= e''[(r, q').(r'', q'')/o] && \text{by Proposition 1(ii)} \\ &= e'' && \text{by Proposition 1(iii), (8)} \\ &\overset{x}{\Rightarrow}_K f_k && \text{by (10), } a = \lambda, \text{ and (8)} \end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 3.

Case 4: $q \overset{a}{\rightarrow} q'.q''$, $a \in \Sigma_s$, has been applied, i.e. $lhs = q$ and $rhs = q'.q''$. Let $o_1, \ldots, o_k$ be the top-of-stack occurrences in e outside of o, i.e. (15) is satisfied, $q_1, \ldots, q_k$ be defined by (16) (cf. Case 2), and

$$e' \triangleq e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(r, q)/o] \tag{23}$$

Then, $\pi(e') = e$ by Proposition 3(v) and $TOS(e) = \{r\}$ by Proposition 4(iv), (11), and (16). Furthermore, by definition of the combined abstraction it is

$$(r, q) \overset{a}{\rightarrow}_K (\hat{r}, q').(r'', q'') \in T_{1S} \tag{24}$$

The same arguments as in Case 3 show that (21) is satisfied. The same arguments as in Case 2 show that (20) holds. Hence,

$$\begin{aligned} e' &\overset{a}{\Rightarrow}_K e'[(\hat{r}, q').(r'', q'')/o] && \text{by Lemma 1 and (24)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(r, q)/o][(\hat{r}, q').(r'', q'')/o] && \text{by (23)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(\hat{r}, q')(r'', q'')/o] && \text{by Proposition 1(ii)}\\ &\overset{\lambda}{\Rightarrow}_K e''[(\hat{r}, q_1)/o_1]\cdots[(\hat{r}, q_k)/o_k][(\hat{r}, q').(r'', q'')/o] && \text{by Lemma 2}\\ &= e'' && \text{by Proposition 1(iii), (20), and (21)}\\ &\overset{y}{\Rightarrow}_K f_k && \text{by (10)} \end{aligned}$$

Hence, (i), (ii), and (iii) hold also for Case 4. CASE 5: $q.q'\xrightarrow{\lambda}q''$ has been applied, i.e. $lhs = q.q'$, $rhs = q''$, and $a = \lambda$. Then, analogous to Case 1, it holds

$$e''[o] = (r, q'') \tag{25}$$

Define for any $r'' \in R_s$:

$$e' \triangleq e''[(r, q).(r'', q')/o] \tag{26}$$

Then, $\pi(e') = e$ by Proposition 3(iv) and $TOS(e) = \{r\}$ by Proposition 4(iii) and (11). By definition of the combined abstraction, it is

$$(r, q)(r'', q')\xrightarrow{\lambda}_K(r, q'') \in T_{S1} \tag{27}$$

Then,

$$\begin{aligned} e' &\overset{\lambda}{\Rightarrow}_K e'[(r, q'')/o] && \text{by Lemma 1 and (27)}\\ &= e''[(r, q).(r'', q')/o][(r, q'')/o] && \text{by (26)}\\ &= e''[(r, q'')/o] && \text{by Proposition 1(ii)}\\ &= e'' && \text{by Proposition 1(iii) and (25)}\\ &\overset{x}{\Rightarrow}_K f_k && \text{by (10), } a = \lambda, \text{ and (8)} \end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 5.
CASE 6: $q.q'\xrightarrow{a}q''$, $a \in \Sigma_s$, has been applied, i.e. $lhs = q.q'$ and $rhs = q''$. Let $o_1, \ldots, o_k$ be the top-of-stack occurrences in e outside of o, i.e. (15) is satisfied, $q_1, \ldots, q_k$ be defined by (16) (cf. Case 2), and for any $r'' \in R_s$ be

$$e' \triangleq e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(r, q).(r'', q'))/o] \tag{28}$$

Then, $\pi(e') = e$ by Proposition 3(v) and $TOS(e) = \{r\}$ by

Proposition 4(iv), (11), and (16). Furthermore, by definition of the combined abstraction it is

$$(r, q).(r'', q')\xrightarrow{a}_K(\hat{r}, q'') \in T_{S1} \tag{29}$$

The same arguments as in Case 5 show that (25) is satisfied. The same arguments as in Case 2 and Case 4 show that (20) holds. Hence,

$$\begin{aligned} e' &\overset{a}{\Rightarrow}_K e'[(\hat{r}, q'')/o] && \text{by Lemma 1 and (29)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(r, q).(r'', q')/o][(\hat{r}, q'')/o] && \text{by (28)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(\hat{r}, q'')/o] && \text{by Proposition 1(ii)}\\ &\overset{\lambda}{\Rightarrow}_K e''[(\hat{r}, q_1)/o_1]\cdots[(\hat{r}, q_k)/o_k][(\hat{r}, q'')/o] && \text{by Lemma 2}\\ &= e'' && \text{by Proposition 1(iii), (20), and (21)}\\ &\overset{y}{\Rightarrow}_K f_k && \text{by (10)} \end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 6.
CASE 7: $q\xrightarrow{\lambda}q'. \parallel q''$ has been applied, i.e. $lhs = q$, $rhs = q' \parallel q''$, and $a = \lambda$. By (9) and Proposition 3(i) it holds

$$e''[o] = (r, q')\|(r, q'') \tag{30}$$

Define e' as in Case 1 and Case 3, i.e. by (12). Then, $\pi(e') = e$ by Proposition 3(iv) and $TOS(e) = \{r\}$ by Proposition 4(iii) and (11). By definition of the combined abstraction, it is

$$(r, q)\xrightarrow{\lambda}_K(r, q')\|(r, q'') \in T_{1P} \tag{31}$$

Hence,

$$\begin{aligned} e' &\overset{\lambda}{\Rightarrow}_K e'[(r, q')\|(r'', q'')/o] && \text{by Lemma 1 and (31)}\\ &= e''[(r, q)/o][(r, q')\|(r, q'')/o] && \text{by (12)}\\ &= e''[(r, q')\|(r, q'')/o] && \text{by Proposition 1(ii)}\\ &= e'' && \text{by Proposition 1(iii) and (21)}\\ &\overset{x}{\Rightarrow}_K f_k && \text{by (10), } a = \lambda, \text{ and (8)} \end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 7.
CASE 8: $q\xrightarrow{a}q'\|q''$, $a \in \Sigma_s$, has been applied, i.e. $lhs = q$ and $rhs = q'\|q''$. Let $o_1, \ldots, o_k$ be the top-of-stack occurrences in e outside of o, i.e. (15) is satisfied, $q_1, \ldots, q_k$ be defined by (16) (cf. Case 2), and e' be defined by (17) (as in Case 2 and 4). Then, $\pi(e') = e$ by Proposition 3(v) and $TOS(e) = \{r\}$ by Proposition 4(iv), (11), and (16). Furthermore, by definition of the combined abstraction it is

$$(r, q)\xrightarrow{a}_K(\hat{r}, q')\|(\hat{r}, q'') \in T_{1P} \tag{32}$$

The same arguments as in Case 3 show that (21) is satisfied. The same arguments as in Case 2 show that (20) holds.Hence,

$$\begin{aligned} e' &\overset{a}{\Rightarrow}_K e'[(\hat{r}, q')\|(\hat{r}, q'')/o] && \text{by Lemma 1 and (32)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(r, q)/o][(\hat{r}, q')\|(\hat{r}, q'')/o] && \text{by (17)}\\ &= e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(\hat{r}, q')(\hat{r}, q'')/o] && \text{by Proposition 1(ii)}\\ &\overset{\lambda}{\Rightarrow}_K e''[(\hat{r}, q_1)/o_1]\cdots[(\hat{r}, q_k)/o_k][(\hat{r}, q')\|(\hat{r}, q'')/o] && \text{by Lemma 2}\\ &= e'' && \text{by Proposition 1(iii), (20), and (21)}\\ &\overset{y}{\Rightarrow}_K f_k && \text{by (10)} \end{aligned}$$

Hence, (i), (ii), and (iii) hold also for Case 8.
CASE 9: $q \parallel q'\xrightarrow{\lambda}q''$ has been applied, i.e. $lhs = q \parallel q'$, $rhs = q''$, and $a = \lambda$. Then, it holds (25) analogous to Case 5. Define

$$e' \triangleq e''[(r, q)\|(r, q')/o] \tag{33}$$

Then, $\pi(e') = e$ by Proposition 3(iv) and $TOS(e) = \{r\}$ by Proposition 4(iii) and (11). By definition of the combined abstraction, it is

$$(r, q)\|(r, q')\xrightarrow{\lambda}_K(r, q'') \in T_{P1} \tag{34}$$

Then,

$$\begin{aligned} e' &\overset{\lambda}{\Rightarrow}_K e'[(r, q'')/o] && \text{by Lemma 1 and (34)}\\ &= e''[(r, q)\|(r'', q')/o][(r, q'')/o] && \text{by (33)}\\ &= e''[(r, q'')/o] && \text{by Proposition 1(ii)}\\ &= e'' && \text{by Proposition 1(iii) and (25)}\\ &\overset{x}{\Rightarrow}_K f_k && \text{by (10), } a = \lambda, \text{ and (8)} \end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 9.
CASE 10: $q \parallel q'\xrightarrow{a}q''$, $a \in \Sigma_s$, has been applied, i.e. $lhs = q\|q'$ and $rhs = q''$. Let $o_1, \ldots, o_k$ be the top-of-stack occurrences in e outside of o, i.e. (15) is satisfied, $q_1, \ldots, q_k$ be defined by (16) (cf. Case 2), and e' be defined by

$$e' \triangleq e''[(r, q_1)/o_1]\cdots[(r, q_k)/o_k][(r, q)\|(r, q' = /o] \tag{35}$$

Then, $\pi(e') = e$ by Proposition 3(v) and $TOS(e) = \{r\}$ by Proposition 4(iv), (11), and (16). Furthermore, by definition of the combined abstraction it is

$$(r, q)\|(r'', q') \xrightarrow{a}_K (\hat{r}, q'') \in T_{P1} \tag{36}$$

The same arguments as in Case 5, 6, and 9 show that (25) is satisfied. The same arguments as in Case 2, 4, 6, and 8 show that (20) holds. Hence,

$$\begin{aligned}
e' &\overset{a}{\Rightarrow}_K e'[(\hat{r}, q'')/o] && \text{by Lemma 1 and (36)} \\
&= e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(r, q)\|(r, q')/o][(\hat{r}, q'')/o] && \text{by (25)} \\
&= e''[(r, q_1)/o_1] \cdots [(r, q_k)/o_k][(\hat{r}, q'')/o] && \text{by Proposition 1(ii)} \\
&\overset{\lambda}{\Rightarrow}_K e''[(\hat{r}, q_1)/o_1] \cdots [(\hat{r}, q_k)/o_k][(\hat{r}, q'')/o] && \text{by Lemma 2} \\
&= e'' && \text{by Proposition 1(iii), (20), and (21)} \\
&\overset{y}{\Rightarrow}_K f_k && \text{by (10)}
\end{aligned}$$

Thus, (i), (ii), (iii) is also satisfied for Case 10.

With Case 1–10 all possibilities are considered, and for each case (i), (ii), (iii) are satisfied. This completes the proof □

References

1. Afrati FN, Borkar V, Carey M, Polyzotis N, Ullman JD (2011) Map-reduce extensions and recursive queries. In: Proceedings of the 14th international conference on extending database technology. ACM, pp 1–8
2. Allen R, Garlan D (1997) A formal basis for architectural connection. ACM Trans Softw Eng Methodol (TOSEM) 6(3): 213–249
3. Both A, Zimmermann W, Franke R (2010) Model checking of component protocol conformance—optimizations by reducing false negatives. Electron. Notes Theor. Comput. Sci. 263:67–94
4. Both A, Zimmermann W (2008) Automatic protocol conformance checking of recursive and parallel BPEL systems. In: IEEE sixth European conference on web services (ECOWS '08), pp 81–91. IEEE Computer Society
5. Both A, Zimmermann W (2008) Automatic protocol conformance checking of recursive and parallel component-based systems. In: Component-based software engineering, 11th international symposium (CBSE 2008), pp 163–179
6. Both A, Zimmermann W (2009) A step towards a more practical protocol conformance checking algorithm. In: 35th Euromicro conference on software engineering and advanced applications (SEAA 2009), pp 458–465. IEEE Computer Society
7. Bouajjani A, Emmi M (2012) Analysis of recursively parallel programs. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages POPL'12, pp 203–214
8. Bravetti M, Zavattaro G (2009) Contract-based discovery and composition of web services. Formal Methods Web Serv, pp 261–295
9. Bures T, Hnetynka P, Plasil F (2006) Sofa 2.0: Balancing advanced features in a hierarchical component model. In: SERA '06: Proceedings of the fourth international conference on software engineering research, management and applications, pp 40–48. IEEE Computer Society
10. Burkart O, Steffen B (1992) Model checking for context-free processes. In: CONCUR'92: Proceedings of the 3rd international conference on concurrency theory, lecture notes in computer science, vol. 630. Springer, pp 123–137
11. Burkart O, Steffen B (1994) Pushdown processes: parallel composition and model checking. In: CONCUR'94: Proceedings of the 5th international conference on concurrency theory, lecture notes in computer science, vol. 836. Springer, pp 98–113
12. de Caso G, Braberman V, Garbervetsky D, Uchitel S (2012) Automated abstractions for contract validation. IEEE Trans Softw Eng 38(1):141–162
13. Dahl OJ, Nygaard K (1966) Simula: an algol-based simulation language. Commun ACM 9:671–678
14. Dumez C, Bakhouya M, Gaber J, Wack M, Lorenz P (2013) Model-driven approach supporting formal verification for web service composition protocols. J Netw Comput Appl 36(4):1102–1115
15. Durán F, Ouederni M, Salaün G (2012) A generic framework for n-protocol compatibility checking, Sci. Comput. Program. vol. 77, pp 870–886. Elsevier North-Holland, Inc.
16. Foster H, Uchitel S, Magee J, Kramer J (2003) Model-based verification of web service compositions. In: ASE, pp 152–163
17. Ghezzi C, Mocci A, Sangiorgio M (2012) Runtime monitoring of functional component changes with behavior models. In: Models in software engineering. Springer, pp 152–166
18. Gorbenko A, Romanovsky A, Kharchenko V, Mikhaylichenko A (2008) Experimenting with exception propagation mechanisms in service-oriented architecture. In: Proceedings of the 4th international workshop on exception handling, WEH '08, pp 1–7. ACM
19. Gosling J, Joy B, Steele G (2012) The java language specification. Addison-Wesley, Reading
20. Haddad S, Poitrenaud D (2007) Recursive petri nets. Acta Informatica 44(7–8):463–508
21. Hauck EA, Dent BA (1968) Burroughs' b6500/b7500 stack mechanism. In: AFIPS '68 (Spring): proceedings of the April 30-May 2, 1968, spring joint computer conference. ACM, pp 245–251
22. Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation, 2nd edn. Addison-Wesley, Reading
23. Jannach D, Gut A (2011) Exception handling in web service processes. The evolution of conceptual modeling, pp 225–253
24. Lin HH, Aoki T, Katayama T (2010) Non-regular adaptation of services using model checking. In: 13th IEEE international symposium on object/component/service-oriented real-time distributed computing (ISORC). IEEE, pp 170–174
25. Löwe W, Neumann R, Trapp M, Zimmermann W (1999) Robust dynamic exchange of implementation aspects. In: TOOLS 29—technology of object-oriented languages and systems. IEEE, pp 351–360
26. Mayr R (2000) Process rewrite systems. Inf Comput 156(1–2):264–286
27. Müller M, Balz M, Goedicke M (2011) Enriching java enterprise interfaces with formal sequential contracts. In: Proceedings of the third workshop on behavioural modelling (BM-FA '11). ACM, pp 5–11
28. Nierstrasz O (1995) Regular types for active objects. In: Nierstrasz O, Tsichritzis D (eds) Object-oriented software composition. Prentice-Hall, Englewood Cliffs, pp 99–121
29. Papazoglou MP, Traverso P, Dustdar S, Leymann F (2007) Service-oriented computing: state of the art and research challenges. Comput Innov Technol Comput Prof 40(11):38–45
30. Parizek P, Plasil F (2008) Modeling of component environment in presence of callbacks and autonomous activities. In: TOOLS (46), pp 2–21
31. Plasil F, Visnovsky S (2002) Behavior protocols for software components. In: IEEE Trans Softw Eng (28), pp 1056–1076
32. Pradel M, Jaspan C, Aldrich J, Gross TR (2012) Statically checking API protocol conformance with mined multi-object specifications. In: Proceedings of the 2012 international conference on software engineering (ICSE 2012), pp 925–935. IEEE

33. Rajamani S, Rehof J (2006) Models for contract conformance. Leverag Appl Formal Methods pp 181–196
34. Reisig W (2005) Modeling- and analysis techniques for web services and business processes. In: Steffen M, Zavattaro G (eds) Formal methods for open object-based distributed systems: 7th IFIP WG 6.1 international conference, FMOODS 2005, Athens, Greece, June 15–17, 2005. Proceedings, LNCS, vol. 3535, pp 243–258. Springer
35. Ren H, Liu J (2012) Service substitutability analysis based on behavior automata. Innov Syst Softw Eng 8(4):301–308
36. Salaün G, Bordeaux L, Schaerf M (2004) Describing and reasoning on web services using process algebra. In: International conference on web services, p 43. IEEE Computer Society
37. Schmidt HW, Krämer BJ, Poernemo I, Reussner R (2002) Predictable component architectures using dependent finite state machines. In: Proceedings of the NATO workshop radical innovations of software and systems engineering in the future, lecture notes in computer science, vol. 2941. Springer, pp 310–324
38. Tenzer J, Stevens P (2003) Modelling recursive calls with uml state diagrams. In: 6th international conference on fundamental approaches to software engineering (FASE'03), lecture notes in computer science, vol. 2621. Springer, pp 135–149
39. van der Aalst WMP (1997) Verification of workflow nets. LNCS 1248:407–426
40. Zimmermann W, Schaarschmidt M (2006) Automatic checking of component protocols in component-based systems. In: Löwe W, Südholt M (eds) Software composition, LNCS, vol. 4089. Springer, pp 1–17