Specifying and generating behavioral service adapters based on transformation rules

Christian Gierds[1], Arjan J. Mooij[2], and Karsten Wolf[3]

[1]Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin, Germany*.
gierds@informatik.hu-berlin.de
[2]Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands**.
A.J.Mooij@tue.nl
[3]Universität Rostock, Institut für Informatik, 18051 Rostock, Germany*.
karsten.wolf@informatik.uni-rostock.de

Abstract. Behavioral adapters are a way to establish proper interaction between services that have been developed independently. We present a novel approach for specifying such adapters, based on domain-specific transformation rules that reflect the elementary operations that adapters can perform. We show how complex adapters that adhere to these rules can be generated using existing controller generation algorithms. We discuss some example applications, including real-world business processes.

1 Introduction

Service-oriented computing [1] advocates the idea of creating complex business activities by composing less-complex business activities, called *services*. From the beginning it has been clear that this process of service composition requires a mechanism for bridging various incompatibilities (format, semantics, behavior, non-functional) between the services to be composed. Such mechanisms are called *mediators* or *adapters*; we shall use the latter term.

Behavioral adaptation aims to adjust the communication between services such that certain behavioral properties (like deadlock-freedom, or weak termination) hold in the composed system. Formally, for any behavioral property X, an X-adapter for services P ("provider") and R ("requester") is a service A ("adapter") such that $P \oplus A \oplus R$ has property X, where $\oplus$ denotes a composition operator.

Without further restrictions, however, this setting is impractical. In many cases, it would allow A to consist of any two unconnected components A_P and A_R such that both $P \oplus A_P$ and $A_R \oplus R$ have property X. Such an adapter works,

* Supported by German Research Foundation (DFG) under grants RE 834/18-1 and WO 1466/11-1

** This work has been carried out as a part of the Poseidon project under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

however, only under the assumption that it can arbitrarily create messages, including passwords or messages with semantically relevant contents. Like in all existing approaches, unintended behavior must be excluded by restricting the capabilities of the adapter to those that make sense for the particular message types. We propose a specification of elementary activities (SEA in the sequel) that consists of transformation rules on message types.

Thus the adapter specification consists of the given services and the SEA. In this article we study the use of an SEA; obtaining an SEA is outside the scope of this article. Although the given services will be used for generating the adapter, they will not be used by the final adapter. The final adapter only acts on the streams of communicated messages.

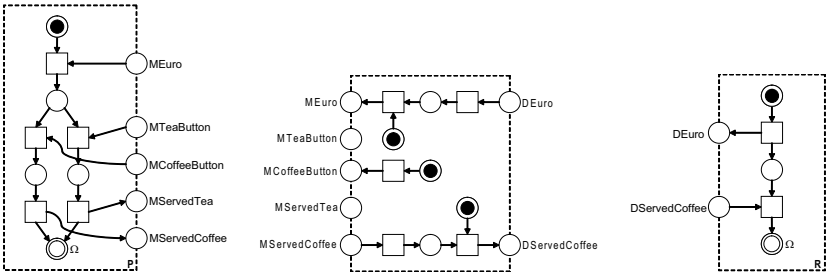
Our adapter synthesis approach constructs an X-adapter A that consists of two sub-units, i.e., $A = E \oplus C$, where service E models the elementary activities in the SEA, and service C is a controller that triggers the activities in E according to the actual communications between P and R. In this article, we show how E can be generated from the given SEA. To synthesize C, we show how to use existing technology for controller synthesis. Given services P, R, and E, controller C is an X-strategy of service $P \oplus E \oplus R$, i.e., a service C such that $P \oplus E \oplus R \oplus C$ has property X. Notice that our general approach is independent from the property X, so as soon as the controllability problem for a property is solved, the adapter synthesis problem can be reduced to it.

The computation of X-strategies has been studied in [2] and is supported by publicly available tools like FIONA [3]. For some properties X (most prominently, deadlock freedom), FIONA is even capable of computing an operational characterization of *all* (possibly infinitely many) controllers C for a given service. This feature can be used in future work for selecting a cost-effective adapter (according to various criteria) from the set of *all* possible adapters.

Thus, given the services and the SEA, our approach is fully automated. To validate our approach, we have implemented the required tool chain. Even on real-world business processes, it turns out to perform reasonable. As related work on automated adapter generation reports only little about experimental results, we do not compare our results.

Running Example As an illustration we use a beverage vending machine, which can easily be discussed in detail. The beverage vending machine P, modeled in Fig. 1(a) as a Petri net (see Sect. 2), expects the input of a Euro (MEuro), the choice for coffee or tea (MCoffeeButton or MTeaButton, respectively) and finally serves coffee or tea (MServedCoffee or MServedTea) accordingly.

On the other side, the coffee drinker R, modeled in Fig. 1(c), just provides a Euro (DEuro) and awaits his coffee (DServedCoffee). We generally assume that the interfaces of the given services are disjoint, which explains the pairs of messages like MEuro and DEuro. Figure. 1(b) contains the kind of adapter we are looking for. Although the adapter is very well capable of pressing the tea button, the adapter generator has intentionally determined not to do so.



(a) Vending machine (b) Adapter $E \oplus C$ (c) Coffee drinker

Fig. 1. Example: beverage vending machine

Related Work Our approach differs in various ways from existing work on behavioral adaptation. Related work like [4,5] is based on traces, which traditionally makes it hard to reason about deadlocks and moments of choice. From concurrency theory, it is known that these are of utmost importance.

Many authors, like [6,7,8], develop new dedicated algorithms for adapter generation. In contrast, we can immediately use existing algorithms for controller generation, and thus we also obtain a nicer conceptual structure. Moreover, in [7] the interaction patterns (i.e., abstract models of the given services) may not contain recursion, while we support arbitrary services, including those that contain loops.

The fully-automated approach of [4] relies on a data structure called *modified reachability tree* [9]. However, in [10] it is reported that some of the claimed properties are incorrect, including a fundamental theorem about deadlocks. Furthermore, their used interface matchings are very restricted.

In [5] adapters are studied to ensure that a provided service behaves as a required service. We expect that similar techniques should be applicable as in our setting. Nevertheless, the kind of adapters that we are looking for differs from those studied by [5]. Their adapters modify the given services, and in that sense, their adapters transform the (occurrences of) actions instead of the transmitted messages. In this way they can address different kinds of mismatches (e.g., some instances of collapse), but at the expense that the adapters need to modify the given services including their control flow. Obviously, such an approach is not valid if the given services are not running under our command.

Furthermore our generated adapters do not need to be verified as in [11], since the synthesis of the adapter's sub-unit C guarantees the adapters correctness by construction.

Overview In Sect. 2, we introduce Petri nets as the underlying formalism of our approach. In Sect. 3, we present our language for specifying elementary adapter activities. In Sect. 4, we describe the synthesis of the adapter, and in particular, the partial adapter E that is responsible for executing the specified elementary

activities. This approach is used for the examples described in Sect. 5. Finally, in Sect. 6, we draw some conclusions and discuss future work.

2 Modeling Services

We formulate our approach using Petri net models, which is appropriate given the strong links between Petri nets and real-world service description languages. There exist feature-complete translations from languages like WS-BPEL to Petri nets [12,13,14,3] as well as translations from Petri nets to WS-BPEL [15]. Another reason for this choice are existing tools like Fiona , which we shall use.

We use an extension of Petri nets that is called open nets [16,17]. To model asynchronous communication with an environment, open nets have an interface that consists of input places and output places. To model successful completion of a service instance, open nets have a set of final markings.

Definition 1 (Open net). *An* open net N *consists of*

- *two finite and disjoint sets P (of places) and T (of transitions);*
- *two disjoint subsets P_i (of input places) and P_o (of output places) of P;*
- *a flow relation F such that $F \subseteq ((P \setminus P_o) \times T) \cup (T \times (P \setminus P_i))$;*
- *an initial marking m_0 such that $m_0(p) = 0$ for all $p : p \in P_i \cup P_o$;*
- *a set Ω of final markings such that no final marking enables any transition in T, and such that $m(p) = 0$ for all $m : m \in \Omega$ and for all $p : p \in P_i \cup P_o$.*

Given an open net N, a marking is a mapping from P to the naturals; for any marking m, it is said that there are $m(p)$ tokens in each place p. A transition t is enabled in marking m iff $m(p) > 0$ for all places $p : (p, t) \in F$. A marking m' is reachable from marking m iff there exists a sequence of firings from m to m'. Firing an enabled transition t in marking m leads to marking a m' such that $m'(p) = m(p) - W(p, t) + W(t, p)$ for all places p, where $W(x, y) = 1$ if $(x, y) \in F$ and $W(x, y) = 0$ otherwise.

Their graphical representation uses circles for places, squares for transitions, and arrows for arcs (i.e., elements from the flow relation). The interface places are depicted on the border, and the places with a token in the initial marking are marked by a black dot. The final markings are typically omitted, or indicated suggestively, e.g., using a label Ω (for an example see Fig. 1(a)).

Open nets can be composed by fusing the interface places that are an input place of one net and an output place of the other. Two open nets are composable if they only share such places. Without loss of generality, we assume that all constituents except the interfaces of the involved nets are disjoint, e.g., by renaming the internal (non-interface) places and transitions. Regarding our running example, all pairs of nets in Fig. 1 are composable.

Definition 2 (Composition). *Let N_1 and N_2 be open nets. N_1 and N_2 are* composable *iff $(P_1 \cup T_1) \cap (P_2 \cup T_2) \subseteq (P_{i1} \cap P_{o2}) \cup (P_{o1} \cap P_{i2})$. If N_1 and N_2 are composable, their* composition $N = N_1 \oplus N_2$ *is defined by*

$P = P_1 \cup P_2; \quad T = T_1 \cup T_2; \quad F = F_1 \cup F_2;$
$P_i = (P_{i1} \cup P_{i2}) \setminus (P_{o1} \cup P_{o2}); \quad P_o = (P_{o1} \cup P_{o2}) \setminus (P_{i1} \cup P_{i2});$
$m_0 = m_{01} \oplus m_{02}; \quad \Omega = \{m_1 \oplus m_2 \mid m_1, m_2 : \ m_1 \in \Omega_1 \ \wedge \ m_2 \in \Omega_2\}.$

For markings m_1 of N_1 and m_2 of N_2, the marking $m_1 \oplus m_2$ is defined as $(m_1 \oplus m_2)(p) = m_1(p)$ for $p : \ p \in P_1$ and $(m_1 \oplus m_2)(p) = m_2(p)$ for $p : \ p \in P_2$. Places $p : \ p \in P_1 \cap P_2$ give no conflict in the definition of composition as these are interface places, and hence $m_1(p) = m_2(p)$ in the initial and final markings.

Services are made to be composed with other services. In turn, complete service choreographies correspond to services with an empty interface. As usual, we define some behavioral properties for nets with an empty interface.

Definition 3 (Deadlock freedom, weak termination). *Let N be an open net with an empty interface. Marking m is a deadlock if $m \notin \Omega$ and no transition is enabled in m. N is* deadlock-free *iff no deadlock markings are reachable from m_0. N is* weakly terminating *iff for every marking m that is reachable from m_0, a marking in Ω is reachable from m.*

To define similar properties for arbitrary open nets, we define the notion of controllability. This means that the net can be composed with another net such that the result has an empty interface, and has the behavioral property. As the composition of the three nets in Fig. 1 is deadlock-free, the composition of each subset of these services is deadlock-free controllable.

Definition 4 (Controllability). *Let N be an open net and X a behavioral property. N is* X-controllable *iff there exists an open net N' such that $N \oplus N'$ has an empty interface and has property X. Such an N' is an X-strategy of N.*

There exist algorithms for deciding controllability with respect to deadlock freedom or weak termination [18,2,19,20]. These algorithms rely on the actual construction of a witness net N'; tools like Fiona [3] or Wombat4WS [21] provide efficient implementations. More precisely, the algorithms construct transition systems (automata) that represent the behavior of N'. This is, however, not a significant difference as there is a mature theory for the transformation of transition systems to Petri nets [22] which is supported by existing tools like Petrify [23].

3 Specification of the Elementary Ectivities

The synthesis of an adapter has to start with a specification. Our specifications consist of the services to be adapted, and the capabilities of the adapter. These capabilities always include receiving and sending messages, and hence delaying the delivery of messages. In addition, they include ways in which messages can be transformed; thus reflecting semantic dependencies between messages.

Most approaches [6,7,8,5,24,4] agree that the capabilities of a behavioral adapter should include the following activities on messages:

Creation "Can an adapter arbitrarily generate a message?" This is possible for basic control messages, such as simple acknowledgments, and for messages with a clear default value. However, it is typically impossible for messages containing important data such as passwords, personal data of a user, etc.

Copy "Can an adapter deliver a message multiple times?" This is possible for most electronic messages, although it could be inappropriate for single-use data such as transaction numbers or single-access passwords. It is also inappropriate for messages that represent real goods, e.g., a book, or a seat.

Delete "Can an adapter delete a message?" This is possible for most electronic data (except legally important messages like invoices, cancelation notices etc.), while it is inappropriate for real goods.

Transform "Can an adapter transform the content of some messages into the content of some other messages?" They can do so if the underlying transformation routine is provided, e.g., calculating a metric measure from an imperial one, or deriving a city name from a zip code.

Based on these example activities, it becomes clear that the applicability of an activity to a particular message type strongly depends on semantic considerations. Hence, the capabilities of the adapter must be specified per message type.

3.1 Transformation Rules

We specify the capabilities of an adapter using a *specification of elementary activities* (SEA). Given a set of message types M, an SEA is a set of transformation rules on these message types. Each rule has the shape $X \overset{Z}{\mapsto} Y$, where X and Y are bags (multi sets) over the set M, and where Z is the actual transformation (e.g. in terms of XSL). Such a rule denotes that, using Z, messages of the types X are consumed, and messages of the types Y are produced.

The set M may contain auxiliary message types, which do not correspond to an interface. Moreover, given some messages that were received from any number of services, the adapter can apply several rules before sending any messages to the interface. Synthesizing an adapter then boils down to applying these rules in the right order, and sending messages to the interface at the right moments. So, rule applications are the elementary building blocks of the adapter.

For the synthesis of adapters, we can largely abstract from the actual data transformations Z. Therefore we often omit Z in the transformation rules, but we will discuss how Z can be integrated in the generated adapters.

Table 1 shows some examples of how the activities discussed before can be expressed in terms of transformation rules. For some rules, we give two versions: one for real items and one for electronic items. Of course, similar rules can be built for mixtures of electronic and real items. Some more-complicated patterns would require multiple rules. For example, a typical collapse pattern, where an arbitrary series of a messages has to be merged into one message b, could be modeled using the two rules $\mapsto\ b$ and $a, b\ \mapsto\ b$.

Table 1. Examples of elementary activities in terms of transformation rules

Elementary activity	Transformation rule
Create a	$\mapsto a$
Copy a	$a \mapsto a, a$
Delete a	$a \mapsto$
Transform a into b	$a \mapsto b$ or $a \mapsto a, b$
Split a into b, c, d	$a \mapsto b, c, d$ or $a \mapsto a, b, c, d$
Merge a, b, c into d	$a, b, c \mapsto d$ or $a, b, c \mapsto a, b, c, d$
Recombine a, b, c to d, e, f	$a, b, c \mapsto d, e, f$ or $a, b, c \mapsto a, b, c, d, e, f$

Running Example In our running example, the messages corresponding to euro and coffee can simply be transformed. However, the creation or deletion of euro, coffee, or tea should not be permitted, as these messages represent real goods. Pressing a button can be done by the adapter, and hence the corresponding messages can be created. A possible SEA is given in the following table:

DEuro	$\mapsto$ MEuro
	$\mapsto$ MCoffeeButton
	$\mapsto$ MTeaButton
MServedCoffee	$\mapsto$ DServedCoffee

3.2 Related Techniques

In this article we focus on using a given SEA, although there may be several ways to get one. First, if a particular set of services is composed, the rules may be simple enough to be specified manually. Second, if the interfaces of the services are specified using technologies of semantic web or ontologies, this information could be analyzed to create an SEA automatically. Third, in an intra-organizational environment, there could be strict rules for message names and types from which an SEA can be deduced. These and related techniques like semantic mapping [25] are research areas on their own, and [24] presents an interactive approach to find and refine SEA-like specifications for adapters using mismatch trees. Further exploring these techniques is beyond the scope of this article, but there are enough opportunities to justify the use of an SEA.

A similar kind of rules is proposed in [6,7,8], but there are essential semantic differences. Our transformation rules are asymmetric, while their rules have no direction [26]. There are many situations where asymmetry is important, e.g., when the actual transformation involves public key encryption like "text $\mapsto$ encrypted_text", which is irreversible in an asymmetric cryptographic system. Second, their rules relate messages of one service to messages of the other service, i.e., they do not seem to support transformations where a message is generated from messages of both services. For example, suppose one service executes the sequence send(initiate), send(order), send(money), receive(good); while the

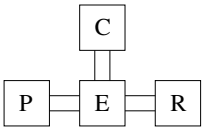


Fig. 2. General structure of our adaptation approach

other service executes the sequence receive(initiate), send(case-id), receive(order-and-case-id), receive(money-and-case-id), send(good). It would be reasonable to specify that order-and-case-id can be generated from order and case-id although these messages arrived from different services.

Other specification languages that have similarities with our approach are presented in [5,24]. However, they typically do not support multiple alternative rules, like $A \mapsto B$ and $A \mapsto C$, which in our case means that at any time the adapter can choose which rule to apply. Moreover, the interface transformations from [5] are local, although, e.g., gather looks similar to our merge pattern. However, in their case the merged actions have to be subsequent actions, while our approach also handles non-local instances.

4 Adapter Generation

To construct an adapter, we first generate a service E that transforms messages according to the given SEA. To control the order in which the transformation rules are applied, and in which the generated messages are released, service E has an additional interface for a controller. So, E is a partial adapter, and to obtain a full adapter a controller C needs to be generated.

At the conceptual level, the following steps need to be performed given two services P and R and an SEA W:

1. generate a partial adapter E from the SEA W;
2. generate a controller C for $P \oplus E \oplus R$;
3. compose E and C to obtain the final adapter $E \oplus C$.

Figure 2 shows a schematic representation of this construction. In the remainder of this section we describe the generation of service E as an open net, which also provides a formal semantics to the SEA format. Afterwards we briefly discuss the synthesis of service C.

4.1 Generating a Partial Adapter for the SEA

Consider some given services for which an adapter must be generated on the basis of a set of rules W. We assume that the interfaces of these services are disjoint, which can be achieved by renaming. For simplicity of presentation, we assume that for each rule in W, the bags before and after the $\mapsto$ are sets; the general case follows analogously using Petri nets with arc multiplicities.

Let I and O denote the union of the input and output places, respectively, of the given services. Let M be a set of message types, containing (the types of) the sets of places I and O, and the set of message types used in W. Thus we allow W to contain auxiliary message types (ones that do not occur in the given services), and hence we have $I \cup O \subseteq M$. For defining the service E, we use names from the space $(M \cup W) \times \{e, n, c, r, s\}$, where e, n, c, r, s denote characters instead of variables; hence we assume that these names do not occur in the given services, and that the sets M and W are disjoint.

The interface of E consists of output places I, input places O (i.e., the interfaces of P and R in opposite orientation), and some input and output places for interaction with the controller. For each message type $m : m \in M$, we introduce in service E an internal place (m, c) (c for "copy"). In the initial and final markings, the internal places are empty, although this can easily be generalized in future work.

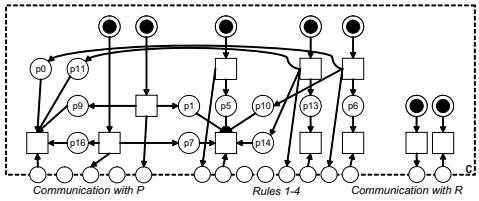
Service E has three kinds of transitions. For every input place $o : o \in O$, there is a transition (o, r) (r for "receive") to move arriving messages from interface place o to their internal place (o, c). For every transformation rule $w : w \in W$, there is a transition (w, c) to perform the actual transformation in terms of the internal places. Finally, for every output place $i : i \in I$, there is a transition (i, s) (s for "send") to move messages from their internal place (i, c) to interface place i.

Finally, we discuss the additional interface places for the controller. For every input place $o : o \in O$, output place (o, n) (n for "notify") notifies an arrived message o. For every transformation rule $w : w \in W$, input place (w, e) (e for enable) enables transformation rule w, and output place (w, n) notifies an execution of w. Finally, for every output place $i : i \in I$, input place (i, e) enables the delivery of a message i (once available).

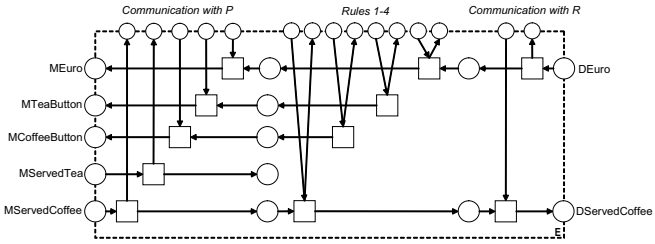
Definition 5 (Service E). *Let I, O, M, W be as introduced before. The corresponding service E is defined as an open net with the following constituents:*

$$
\begin{array}{lcl}
P & = & (M \times \{c\}) \ \cup \ P_i \ \cup \ P_o \\
P_i & = & O \ \cup \ (W \times \{e\}) \ \cup \ (I \times \{e\}) \\
P_o & = & I \ \cup \ (W \times \{n\}) \ \cup \ (O \times \{n\}) \\
T & = & (O \times \{r\}) \ \cup \ (W \times \{c\}) \ \cup \ (I \times \{s\}) \\
\\
F & = & F_r \cup F_c \cup F_s \\
F_r & = & \bigcup_{o \in O} \{ \ [o, (o, r)], \ [(o, r), (o, n)], \ [(o, r), (o, c)] \ \} \\
F_c & = & \bigcup_{w = X \mapsto Y \in W} (\ \{[(m, c), (w, c)] \mid m : \ m \in X\} \cup \\
 & & \quad \{[(w, e), (w, c)], \ [(w, c), (w, n)]\} \ \cup \ \{[(w, c), (m, c)] \mid m : \ m \in Y\} \ ) \\
F_s & = & \bigcup_{i \in I} \{ \ [(i, c), (i, s)], \ [(i, e), (i, s)], \ [(i, s), i] \ \} \\
\\
m_0 & = & \underline{0} \\
\Omega & = & \{\underline{0}\}
\end{array}
$$

We use $\underline{0}$ to denote the marking that is zero in every place. Figure 3(b) shows the service E for the SEA of our running example. By construction, all outputs



(a) Controller C



(b) Partial adapter E

Fig. 3. Services synthesized for the running example

to the given services have been obtained from the inputs of these services using the transformation rules only. The actual scheduling of rule applications and message deliveries is left to a controller using the remaining interface.

4.2 Synthesizing a controller for the partial adapter

The purpose of the controller is to schedule the application of transformation rules and the delivery of messages. To obtain a sufficient record of the interior of service E, the controller is informed about the receipt of incoming messages and completed transformation rules.

For the synthesis of the controller, we rely on existing technology that is based on the concept of controllability. In our setting, we would construct C as a witness for X-controllability of the open net $P \oplus E \oplus R$. The synthesis approach is available for all properties X for which X-controllability has been solved, including deadlock freedom and weak termination. In this way, we achieve—without further efforts—that $P \oplus E \oplus R \oplus C$ has property X. In other words, $E \oplus C$ is an adapter that enforces behavior X on the composition of P and R. Moreover, $E \oplus C$ only uses message transformations specified in the given SEA.

Although controllers are not unique, every controller yields a valid adapter. To maintain most communication capabilities, we are particularly interested in *most-permissive* controllers, which feature *all* behaviors that enforce the desired property. Although it can be quite big, we will see that it is still feasible to compute it. Figure 3(a) shows a representation of the most-permissive controller for our running example. Such controllers contain a lot of non-determinism (includ-

ing concurrency), which can be resolved on the basis of some cost considerations. We leave the careful investigation of these opportunities as future work.

4.3 Refining the adapter

In each resulting adapter $E \oplus C$, there is still a transition (w, c) for each transformation rule $w = X \overset{Z}{\mapsto} Y$. If Z is of the right shape, the transition (w, c) can easily be refined into an implementation of the transformation rule, e.g., in terms of Colored Petri net. For example, a transformation rule that converts distances in meters to distances in feet, can be implemented as multiplication by 3.2808.

Moreover, with the results of [15], the adapter could be transformed into an abstract WS-BPEL process, containing an opaque activity for each transformation rule $w = X \overset{Z}{\mapsto} Y$. If Z is an XSL transformation, we can inject Z into the opaque activity, and thus obtain an executable adapter in a practically relevant language.

5 Examples

In this section, we want to validate our approach using a few examples. For all examples, we use deadlock freedom as the behavioral correctness property to be established. The tool and the examples discussed can be downloaded from the site: `http://www.service-technology.org/icsoc2008/` .

5.1 Tool Support

For the fully-automatic generation of adapters, we extended the tool Fiona such that it can create the partial adapter E from the given SEA, and then uses its controller synthesis algorithm to create the controller C for $P \oplus E \oplus R$. With the means provided by Fiona , we have implemented this functionality with only small efforts. The used chain of procedures consists of the following steps:

1. Create a partial adapter E from the given SEA;
2. Compose P, E, and R into a single open net $P \oplus E \oplus R$;
3. (Use Petri-net reduction to decrease the complexity of the next steps);
4. Use controller synthesis to produce the controller C as a transition system;
5. Use Petrify to transform this transition system into an open net;
6. Compose C and E into a complete adapter;
7. (Use Petri-net reduction to obtain a smaller adapter).

The mentioned Petri-net reduction applies net transformations that preserve the essential properties, in particular the interface behavior of the service. Due to the final (but optional) application of Petri-net reduction, the resulting adapter is usually not recognizable anymore as the composition of E and C, but it is much more compact.

In some cases, the conceptual service E as defined in Sect. 4 contains more interface places than necessary for the controller. For example, using the final

markings of E, we do not need a triggering input nor a notifying output for a transformation rule like $X \mapsto Y$, provided that X is non-empty, X contains no output interfaces and there are no other transformation rules with elements from X at the left-hand side. So, such a rule can be executed as soon as the required messages of types X are present. Thus the interface between E and C becomes smaller, and this improves the performance of the controller synthesis.

Even without such optimizations, the generated service E has to deviate slightly from the one proposed in Sect. 4. The activities for message creation may lead to unbounded places in E and thus to an unbounded state space. As the controllability of unbounded open nets [27] is undecidable, we need to introduce an artificial but arbitrary bound k to the internal places of E. We do so by adding, for each internal place, a complementary place with initially k tokens.

Similarly, we have to ensure that the adapter A respects a message bound m on the interface with $P \oplus R$. To this end, we have slightly modified FIONA such that the markings where such a place contains more than m tokens are immediately classified as uncontrollable. The same restriction could be imposed by inserting a bound of $m+1$ to all these places, and inserting a transition that fires as soon as $m+1$ token are available and that subsequently makes the service uncontrollable. The latter approach does not require a modification of FIONA , but exhibits worse performance.

The stated modifications to the service E are no principal restriction, since every finite state adapter for services P and R can be generated if k and m are chosen sufficiently large.

5.2 Running Example (continued)

For our running example, the tools compute the adapter in Fig. 4 based on a most-permissive controller. This adapter turns out to be equivalent to the one in Fig. 1(b), but this one is the exact result of composing E and C and applying Petri-net reduction. The initial tokens are related to the fact that most transitions only need to fire once, but it is clear that Petri-net reduction does not give the smallest adapter. Although the adapter has similarities with the SEA, notice in particular that the controller generator decided not to use the rule for MTeaButton.

To obtain an adapter with a less-trivial control flow, we extend this example with a repetition. After serving coffee or tea, the machine returns to its initial marking, which is then a final marking as well. The customer can order and receive coffee arbitrarily often, until he non-deterministically chooses to terminate. As the SEA represents domain-specific knowledge, it does not need to be modified. In this case the adapter (see the website mentioned before) can no longer treat the euro's and buttons independently.

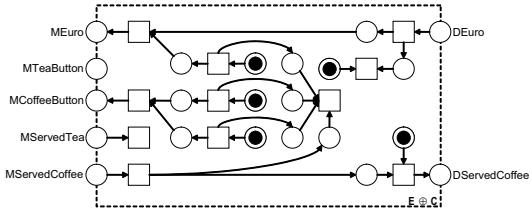


Fig. 4. Generated adapter

Table 2. Comparison of the examples' key data

	SEA	$P \oplus E \oplus R$	most permissive		arbitrary	
	rules	size (p/t)	size (p/t)	time	size (p/t)	time
Beverage	4	25/11	22/11	1 s	19/ 9	0 s
Beverage (Repeated)	4	24/12	34/28	2 s	22/12	0 s
Airline ticket counter	6	34/15	45/22	2 s	35/18	0 s
Car rental service	7	42/20	56/28	60 s	61/26	0 s

5.3 Consultancy Examples

We also applied our approach to two examples that we got as challenges from a consulting company. One models an airline ticket counter, the other one a car rental service. They have been passed to us as WS-BPEL processes, and we used the tool BPEL2oWFN [3] to automatically obtain Petri-net models. The resulting models, the used SEA, and the resulting adapters as well as some explanations can be found on the website mentioned before.

The airline example is interesting because it closes a control cycle: as long as no route is found, the adapter needs to ask for a new route (on another date), until it can deliver a bookable route to the customer. The car rental example is interesting because its services contain a large degree of concurrency.

5.4 Experimental Results

Table 2 contains for every example the number of rules, the number of places (p) and transitions (t) of $P \oplus E \oplus R$, the size of the adapter, and finally the computation time (of Fiona and Petrify together). Apart from adapters based on the most-permissive controller, we also show the results for an arbitrary controller. We have used a computer with 6 GB RAM and four Intel Xeon 3.06 GHz CPUs, although each experiment was running on a single CPU.

The time and memory needed for adapter generation depend on the number of events that need to be evaluated during controller synthesis. In turn, this is influenced by the number of interface places, the number of rules in the SEA, and the level of concurrency in the given services (see the car rental service).

Let us compare the results for most-permissive and arbitrary controllers. The synthesis of a most-permissive controller requires the computation of a larger transition system as intermediate step, which explains the larger computation time. The size of this intermediate result is mostly invisible in the generated Petri-net adapters, since PETRIFY transforms arbitrarily-interleaved transitions into concurrently-enabled Petri-net transitions.

The small run-times for arbitrary controllers suggest that our approach could in fact be feasible in some on-line scenario, where services are composed interactively, e.g., by dragging and dropping icons. Some of the run-times for most-permissive controllers seem to be closer to an off-line scenario, where a layout of a service composition is carefully designed, and then wired by calculating the appropriate adapters. We shall explore the available options in future research.

6 Conclusions and Future Work

We have proposed a new way to generate behavioral adapters. Its first ingredient is a powerful specification of elementary activities (SEA). Its second ingredient is the use of a partial adapter. Its third ingredient is a reduction to controller synthesis, which enables the re-use of existing tools for generating such complex adapters. The approach is systematic and can be extended in various directions.

We described our general approach in terms of Petri nets, for which there exist tools that link it to industrially-relevant languages like WS-BPEL. Using a few examples, we have shown that an implementation of our approach yields acceptable run-times, with opportunities for further improvements.

In future work, we intend to look for optimal adapters, according to some definitions of optimality. To this end, we want to compare different controllers based on finite representations of the complete set of finite-state controllers.

References

1. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (July 2007)
2. Schmidt, K.: Controllability of Open Workflow Nets. In: Enterprise Modelling and Information Systems Architectures. Volume P-75 of LNI. (2005) 236–249
3. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data Knowl. Eng. **64**(1) (2008) 38–54
4. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Proc. ICSOC. Volume 4294 of LNCS. (2006) 27–39
5. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 65–80
6. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R., Toumani, F.: Developing Adapters for Web Services Integration. In: Proc. CAiSE. Volume 3520 of LNCS. (2005) 415–429

7. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74**(1) (2005) 45–54
8. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing Web Service Choreographies. Electr. Notes Theor. Comput. Sci. **105** (2004) 73–94
9. Wang, F., Gao, Y., Zhou, M.: A modified reachability tree approach to analysis of unbounded Petri nets. IEEE Trans Syst Man Cybern B Cybern. **34**(1) (2004) 303–308
10. Yu, R., Wu, W., Hadjicostis, C.: Comments on "A modified reachability tree approach to analysis of unbounded Petri nets". IEEE Trans Syst Man Cybern B Cybern. **36**(5) (2006) 1210
11. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. **19**(2) (1997) 292–333
12. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2-3) (2007) 162–198
13. Moser, S., Martens, A., Häbich, M., Müller, J.: A Hybrid Approach for Generating Compatible WS-BPEL Partner Processes. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 458–464
14. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Proc. WS-FM. Volume 4937 of LNCS. (2007) 77–91
15. Lohmann, N., Kleine, J.: Fully-automatic Translation of Open Workflow Net Models into Human-readable Abstract BPEL Processes. In: Proc. Modellierung. Volume P-127 of Lecture Notes in Informatics (LNI). (March 2008) 57–72
16. Kindler, E.: A compositional partial order semantics for Petri net components. In: Proc. ICATPN. Volume 1248 of LNCS. (1997) 235–252
17. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
18. Martens, A.: Analyzing Web Service based Business Processes. In: Proc. FASE'05. Volume 3442 of Lecture Notes in Computer Science. (April 2005)
19. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Proc. ICATPN. Volume 4546 of LNCS. (2007) 321–341
20. Massuthe, P., Wolf, K.: An Algorithm for Matching Non-deterministic Services with Operating Guidelines. IJBPIM **2**(2) (2007) 81–90
21. Martens, A., Moser, S.: Diagnosing SCA components using wombat. In: Proc. BPM. Volume 4102 of LNCS. (2006) 378–388
22. Badouel, E., Darondeau, P.: Theory of Regions. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS., Springer-Verlag (1996) 529–586
23. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. IEEE transactions on computers **47** (1998) 859–882
24. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. WWW. (2007) 993–1002
25. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal **10**(4) (2001) 334–350
26. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. Science of Computer Programming **61** (2006) 136–151
27. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Informatik-bericht, Universität Rostock (2008) submitted to a journal.