Composing Web-service-like abstract state machines (ASM)

Andreas Friesen and Jens Lemcke

SAP AG, SAP Research at CEC Karlsruhe, Germany
{andreas.friesen, jens.lemcke}@sap.com

Abstract. We ease the design of collaborative business processes respecting desired business goals by the composition algorithm presented in this paper. The composition of multiple parties' business processes is always done with a specific objective in mind. Not only in the positive case, but also if the objective of a business process can not be fulfilled, all participating business processes need to be in some expected recovery state.
We propose a composition algorithm solving the task of designing a collaborative business process while respecting a set of primary and recovery goals. In our model, each business process is described as a finite state machine. The multiplication of all business processes in one single model of possible executions would lead to an explosion of the number of states. Therefore, our composition algorithm directly interprets the multiple finite state machine (FSM) representations and creates a collaborative business process without integrating all FSMs into one single FSM upfront.
Our composition algorithm returns an orchestration of the given business processes only in the case that it can be assured that each execution only leads to an expected primary or recovery goal. In order to prove our concepts, we first mathematically define the execution of business processes and orchestrations by providing abstract state machine (ASM) representations for them. Second, we execute the ASMs in the execution engine CoreASM which shows that the generated orchestration steers the execution of the business processes as intended.

1 Introduction

A typical service-oriented environment consists of a set of Web services distributed in some kind of electronic network. Web services can be understood as the abstraction of a company's IT system interface. Web service technology, namely WSDL,[1] provides a standardized way to represent acceptable operations for electronic communication with a business partner's IT systems.

In a realistic setting, the IT system interface of a company consists of a set of Web service operations. For simplicity, we assume that these operations are grouped into one single Web service definition. The Web service operations have to be communicated with in a specific way according to the internal business process of the company. We shortly refer to its public part as a company's *business process*. The business process can be understood as a behavioral description of the Web service. It can be formulated

[1] http://www.w3.org/TR/wsdl

Dagstuhl Seminar Proceedings 07061
Autonomous and Adaptive Web Services
http://drops.dagstuhl.de/opus/volltexte/2007/1034

in a standard way by using a workflow language, such as WSBPEL,[2] UML activity diagrams [1], or an extension of SAWSDL.[3]

The target of this work is to automatically generate an execution plan for the collaborative business process of multiple business partners based on their individual business processes. We refer to the execution plan as an orchestration of the individual Web services. The engine generating the orchestration is called composer. We define the composition algorithm it performs and the environment it needs to operate, called composition system, in this paper. We describe the architecture of the composition system in the following section. The structure of the Sects 3 to 7 will be based on the architecture. Section 8 concludes.

2 Architecture

In this section, we describe the implementation of our composition engine. For both the Web-service-related specifications and the explanation of the composition algorithm, we use the abstract state machines (ASM) theory [2] throughout the following sections. For the understanding of the activities carried out by the different modules of the composition system, we go through the interrelation of the touched information artifacts in Sect. 2.1, before we define the steps of its execution in Sect. 2.2.

2.1 Correlation of the information artifacts

The correlations of the involved information artifacts is depicted in Fig. 1 on the facing page. In the first place, the Web services and their behavior specifications are given in an appropriate language. The target is to compute an orchestration which directly interacts with the Web services. Thereby, the orchestration has to fulfill the behavioral specifications of each business partner.

The Web service interface is abstracted by the ASMs SEND and RECEIVE. The behavioral specification becomes transformed to an EXECUTE ASM. The composer computes new ASM rules orchestrating the EXECUTE ASM. The orchestrating rules and the ASM themselves communicate through a set of shared variables. As an outlook, all ASMs on the lower part of Fig. 1 on the next page are needed for the transformation back to the executable Web service orchestration on the upper part of the figure.

Throughout the paper, we instantiate our concepts by using SAWSDL and WSBPEL for the representation of Web services and orchestration. Our approach is however general and not limited to these specific languages. Other appropriate representations can be used for the real-world artifacts. If one wants to do this, one has to adapt only the transformations addressed in the following section.

2.2 Process

Figure 2 on the facing page shows the architecture of our composition approach. In the following, we walk through the single parts of the picture. Where applicable, we link to the appropriate section for a deeper explanation of the individual components.

[2] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[3] http://www.w3.org/2002/ws/sawsdl/spec/

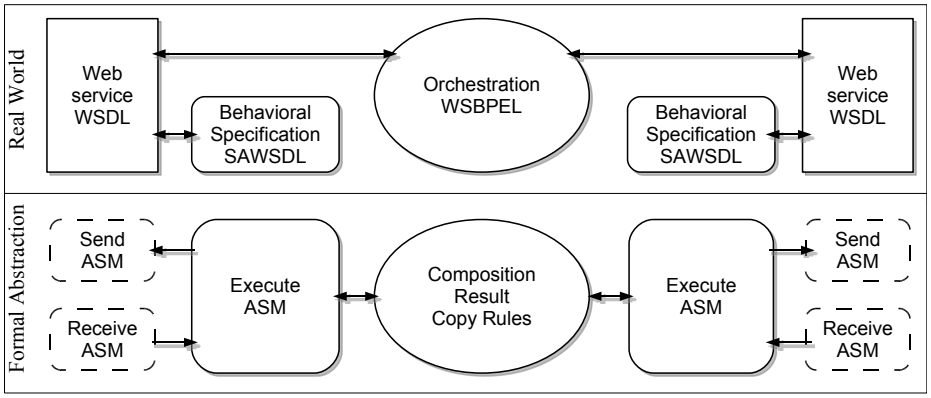

Fig. 1. Artifacts overview.

Repository. The central element of our algorithm is a repository to store the set of Web services to be composed. The repository assigns a unique ID to each Web service contained. A repository should only include a single initiator. An initiator is a Web service whose first action is sending an output message.

Definition 1 (Repository).

$$\text{Repository} := 2^{\text{ID}\times\text{WebService}}$$

$$\text{ID} \ \ldots \text{ set of unique identifiers for Web services in the repository}$$

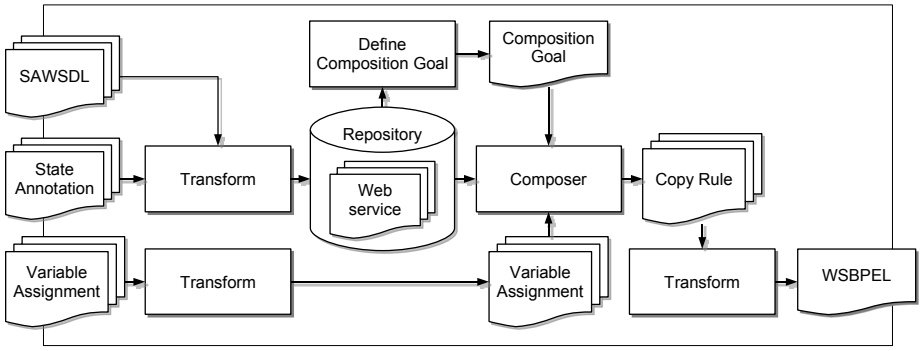

Fig. 2. Process overview.

The mathematical notion of Web services will be detailed in Sect. 3. Figure 3 on the next page shows a set of Web services that make up our example repository. We use the Web services presented in the picture to illustrate the technical descriptions of the rest of the paper.

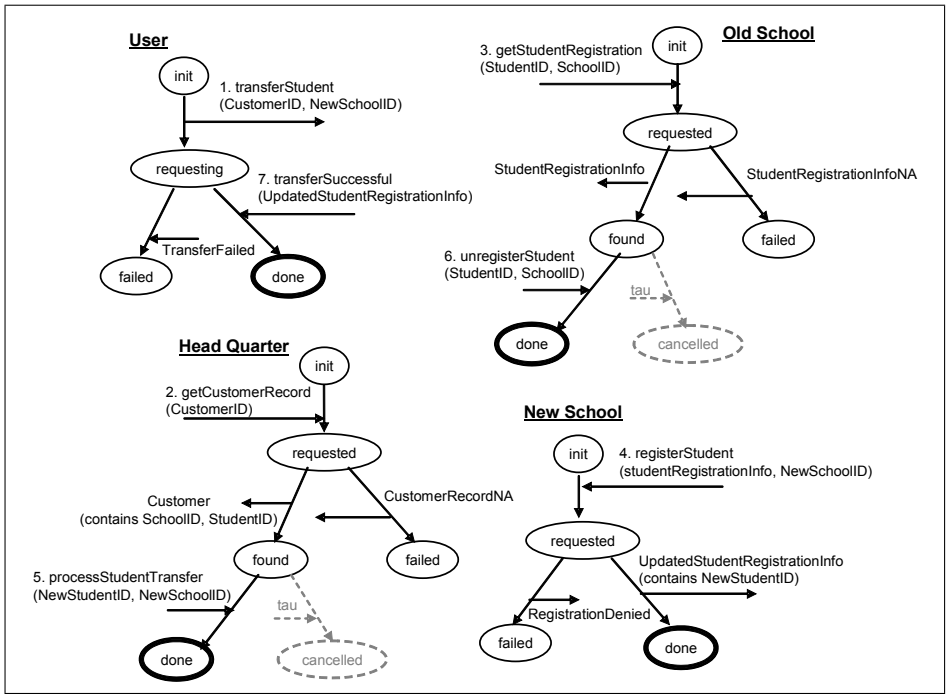

Fig. 3. Exemplary repository. The whole picture represents one Web service repository. Underlined text denotes the ID of a Web service. Each graph describes a Web service's behavior as a finite state machine. Thereby, ellipses denote states, arrows between states denote state transitions, and arrows leaving or arriving at state transitions denote output or input messages. Text at messages starting with a capital letter denotes a variable name. Successful, final states are drawn in bold lines. The "tau"-transitions are virtual transitions. They translate as follows. The pre-state is a final, unsuccessful state. The transition and its posterior state do not exist.

Define composition goal. The Web services in the repository are the bases to define a composition goal. A composition goal states the desired and necessary properties of every possible execution of a composition. A composition is correct if it fulfills the composition goal. We mathematically define the notion of a composition goal and a correct composition in Sect. 4. The composition goal is one of the inputs for our main composition algorithm. The other input is introduced in the following section.

Variable assignments. Web services communicate via a set of variables. In our model, variables are local to Web services. Thus, they can be globally, uniquely referred to by stating the tuple of Web service identifier and local variable name.

Definition 2 (Variable).

$$\mathrm{IN}^{wsId}, \mathrm{OUT}^{wsId} \quad \ldots \quad \text{sets of input and output variables of Web service } \textit{wsId}$$
$$\mathrm{Variable} := \{\, (\, \textit{wsId}, v\,) : \ \textit{wsId} \in \mathrm{ID},\, v \in (\mathrm{IN}^{wsId} \cup \mathrm{OUT}^{wsId}) \,\}$$

In order to be flexible with respect to data and protocol mediation, we explicitly model the possible associations between the variables of different Web services as variable assignments (VarAss). We use the tuple consisting of a Web service ID and the respective variable for this definition.

Definition 3 (Variable assignment).

$$\begin{aligned}\mathrm{VarAss} := \{ ((\mathit{wsId}_1, o), (\mathit{wsId}_2, i)) : \; & \mathit{wsId}_1, \mathit{wsId}_2 \in \mathrm{ID}, \\ & \mathit{wsId}_1 \neq \mathit{wsId}_2,\ o \in \mathrm{OUT}^{\mathit{wsId}_1},\ i \in \mathrm{IN}^{\mathit{wsId}_2} \}\end{aligned}$$

The following restrictions on repositories and variable assignments complement the above definitions.

- Each repository must contain exactly one initiator Web service.
- Variable assignments may never assign an output variable of a service to an input variable of the same service.

Composer. With the Web services available in the repository, a composition goal and a set of allowed variable assignments, all inputs for the main composition component are defined. The outcome of the composer is a set of copy rules. A copy rule states what data are exchanged by which Web service at which point in their execution. These rules are called orchestration of the given Web services. Our notion of orchestration is detailed in Sect. 5. The functioning of the composer is detailed in Sect. 6 including an exemplary run of the composer. The exemplary run of the composer results in a set of copy rules representing an orchestration. An exemplary execution of this orchestration is described in Sect. 7.

Internal model and representation languages. So far, we have talked about our internal mathematical Web service model. The linking to real-world Web service modeling languages is depicted by the transformation blocks in Fig. 2 on page 3. Their explanation is distributed over the appropriate sections. The linking of our Web service model to SAWSDL is explained in Sect. 3. We do not give a specific real-world representation for variable assignments, because there is no standard language for this. Linking the orchestration to the real-world is done in Sect. 5.

3 Web service model

In this section, we focus on the abstraction we use to formally model the behavior of Web services. Figure 4 on the next page details the part of our ASM specification regarding the Web service abstraction. In the following sections, we go through its single parts. In Sect. 3.1, we first give an ASM abstraction of operation calls. The operation calls are denoted by the arrows leading from the SEND and to the RECEIVE blocks in the figure. Second, we define a mathematical model representing a Web service including its behavior in Sect. 3.2. Finally, we give the transformation from the mathematical model to executable ASM specifications in Sect. 3.3. This step covers the SEND, RECEIVE and EXECUTE blocks of the figure.

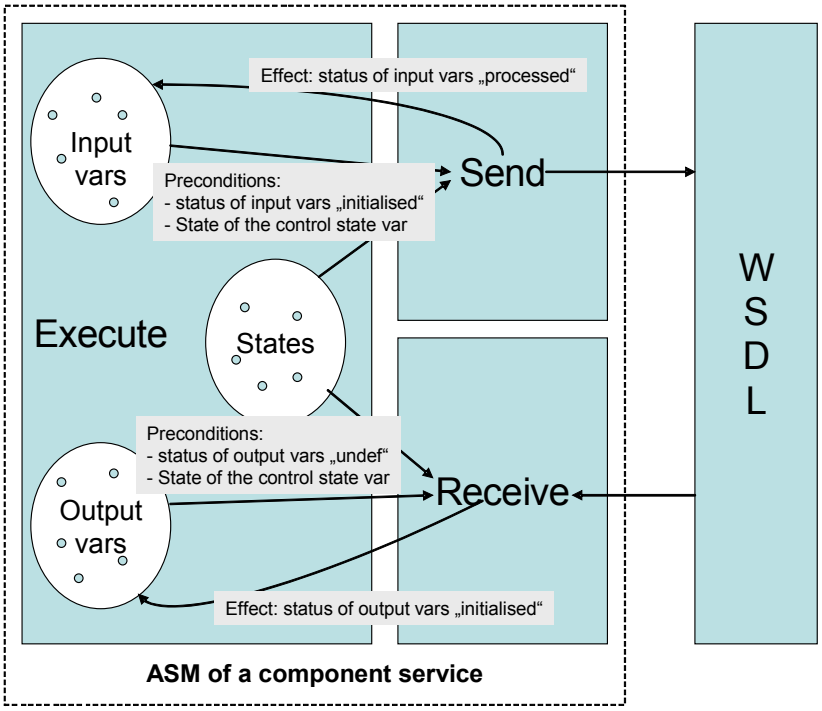

Fig. 4. Web service abstraction. This figure details the parts of Fig. 1 on page 3 that are concerned with abstracting a Web service.

3.1 Operations

Abstract model. In our abstract view, Web services consist of a set of operations. Each operation is either an input or an output operation. In addition, each operation communicates a defined set of variables. We represent this view via the ASMs INVOKEWEBSERVICE and RECEIVEDFROMWEBSERVICE. INVOKEWEBSERVICE calls the input operation communicating the input variables I of Web service *wsId*. RECEIVEDFROMWEBSERVICE returns true if the Web service *wsId* has responded with the output operation communicating the output variables O.

INVOKEWEBSERVICE($wsId \in \mathrm{ID}$, $I \subseteq \mathrm{IN}^{wsId}$) ... native implementation

RECEIVEDFROMWEBSERVICE($wsId \in \mathrm{ID}$, $O \subseteq \mathrm{OUT}^{wsId}$) $\equiv$
  return *received* $\in \{\mathrm{true}, \mathrm{false}\}$ **in**
    ... native implementation, returning truth value

Linking to WSDL. In order to apply our abstract model in the real world, we provide a linking from WSDL to our Web service abstraction. A Web service description in WSDL also consists of a set of operations. Each operation transports a set of parts. Each part corresponds to a variable in our abstraction. In WSDL, operations can be of different type. [4] One-way operations consist of an input only and correspond to the input operations in our abstraction (INVOKEWEBSERVICE). Contrarily, notification operations only contain a single output and correspond to an output operation in our model (RECEIVEDFROMWEBSERVICE). There are two more operation types in WSDL that include an input, an output and a fault in different orders. A request-response operation expects an input before it sends an output. Vice-versa, a solicit-response operation starts with an output to be understood as a request and expects an input as the answer afterward. Instead of the second message, a fault message can be communicated in both message types. Our abstraction of Web services is not yet sufficient to correctly model the request-response and solicit-response message types of WSDL. We describe how to cover these operation types in the following section by using the states introduced there.

3.2 Behavioral specification

Abstract model. Since our aim is to use a Web service as a public interface to an internal business process of a company, it is not sufficient to only represent different, stand-alone operations of a Web service. A business process is a workflow that consists of tasks connected via control flow information. In order to represent this additional information, our abstract view of Web services foresees a possibility to link different operations to complex flows. For this purpose, we introduce the notion of state to Web service descriptions. Figure 5 on the following page presents a taxonomy of different types of states that we use in our Web service model illustrated with some concrete exemplary states. A Web service must contain exactly one initial state and at least one successful, final state. In addition, a Web service may make use of arbitrarily many intermediary and other successful or unsuccessful, final states.

The states of a Web service are used to order its operations. The full definition of a Web service given below therefore contains a state transition function (ST). A state transition consists of a pre-state, a set of input (IN) or output variables (OUT) and a posterior state. A state transition fully defines a Web service operation in our model. The later description of our composition algorithm relies on the following formalization of Web services residing in the repository.

[4] http://www.w3.org/TR/wsdl

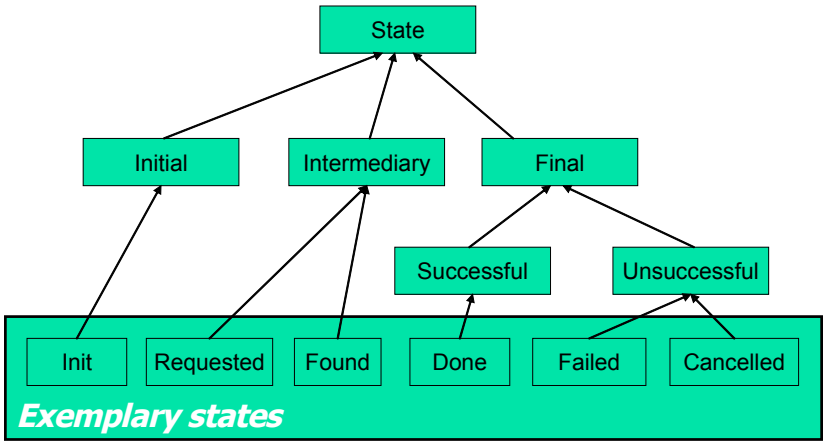

Fig. 5. Web service states. The upper part of the picture shows a state taxonomy whereas the lower part presents some exemplary states for illustration. The arrows denote an is-a hierarchy. The exemplary states correspond to the states of the Old School Web service in Fig. 3 on page 4.

Definition 4 (Web service).

$$
\begin{array}{rcll}
\mathrm{WebService} & := & \langle \mathrm{IN}, \mathrm{OUT}, \mathrm{S}, s_{\mathrm{init}}, \mathrm{S}_{\mathrm{suc}}, \mathrm{S}_{\mathrm{fail}}, \mathrm{ST} \rangle & \\
\mathrm{IN}, \mathrm{OUT} & \ldots & & \text{sets of input and output variables} \\
\mathrm{S} & \ldots & & \text{set of states} \\
s_{\mathrm{init}} & \in & \mathrm{S} & \text{(initial state)} \\
\mathrm{S}_{\mathrm{fin}} & := & \mathrm{S}_{\mathrm{suc}} \cup \mathrm{S}_{\mathrm{fail}},\ \mathrm{S}_{\mathrm{suc}} \cap \mathrm{S}_{\mathrm{fail}} = \emptyset & \text{(final states)} \\
\mathrm{S}_{\mathrm{suc}}, \mathrm{S}_{\mathrm{fail}} & \subset & \mathrm{S} & \text{(un/successful, final states)} \\
\mathrm{ST} & = & \mathrm{ST}_{\mathrm{in}} \cup \mathrm{ST}_{\mathrm{out}} & \text{(state transition function)} \\
\mathrm{ST}_{\mathrm{in}} & : & \mathrm{S} \times (2^{\mathrm{IN}} \setminus \{\emptyset\}) \rightarrow \mathrm{S} & \text{(input state transitions)} \\
\mathrm{ST}_{\mathrm{out}} & : & \mathrm{S} \times (2^{\mathrm{OUT}} \setminus \{\emptyset\}) \rightarrow \mathrm{S} & \text{(output state transitions)}
\end{array}
$$

The following restrictions on Web services complement the definition above.

- Each input transition of a Web service must be uniquely identifiable by the set of variables consumed.
- The state transitions of a Web service must form a directed tree.
- The leave states of a Web service's state transition graph must be final states.
- Each Web service's state transition graph must have exactly one root which is its initial state (s_{init}).
- There must be no input variable in a Web service that also is an output variable of the same service.

Currently, we do neither allow loops in a Web service's behavior, nor the repeated execution of a Web service. The current assumption about the valid behavior of a component service is very restrictive. Therefore, we consider this to be subject to future extension.

Linking to WSDL. By the states, we are able to express the operation types request-response and solicit-response of WSDL. We represent a request-response operation by three state transition where the posterior state of the input operation equals the pre-state of the output and the fault operation. Correspondingly, we represent a solicit-response operation by three state transitions where the posterior state of the output operation equals the pre-state of the input and the fault operation. We use Fig. 3 on page 4 for illustration. The state transitions bordered by the states init, failed and found in the Old School Web service could be the representation of a request-response operation. The state transitions bordered by the states init, failed and done of the User Web service could be the representation of a solicit-response operation.

Linking to SAWSDL. For demonstrating how our model can represent complex Web service definitions in the real world, we utilize an extended version of SAWSDL. In addition to the definitions possible with WSDL, SAWSDL can be used to further annotate the different components of a Web service description. We suggest an annotation that adds the pre-state and posterior state information to each single message definition.

3.3 Deriving abstraction of Web services

In the previous sections, we have presented a formal model of Web services. We introduced state transitions that suggest its execution. However, we did not yet formally define how we expect a Web service definition to be executed. For this purpose, we provide three control state ASMs (EXECUTE, SEND, RECEIVE, see Fig. 4 on page 6) in this section that define the execution of a Web service based on its definition given in the previous sections. The machines of one Web service communicate via specific states of its variables. We define the state of a variable as follows.

Definition 5 (Variable state).

$$\begin{aligned}\mathrm{varState}: \; \{ (\mathit{wsID}, v) \mapsto \mathit{state} : \; & \mathit{wsId} \in \mathrm{ID},\, v \in (\mathrm{IN}^{\mathit{wsId}} \cup \mathrm{OUT}^{\mathit{wsId}}), \\ & \mathit{state} \in \{ \mathrm{undef}, \mathrm{initialized}, \mathrm{processed} \} \}\end{aligned}$$

For each Web service, we define exactly one ASM that performs its state changes (EXECUTE). All other ASMs (e. g., SEND, RECEIVE) can trigger a state change of a Web service only indirectly by writing to input variables and by consuming output variables. We go now through these machines in detail. The figures 1 and 4 provide an illustration of the interrelation of the different machines.

Advancing a Web service's state. The EXECUTE machine consists of a set of update rules changing the state of a Web service depending on its current state and the state of its variables. Whenever a service behavior contains a state transition whose pre-state (s_{pre}) matches the current state ($\mathrm{wsState}$) of the Web service and all variables ($v_1, v_2, \ldots, v_{|V|}$) communicated during this state transition have been $\mathrm{initialized}$, then its state evolves to the posterior state (s_{post}).

EXECUTE($wsId$) $\equiv$ **do forall** $(\, s_{\text{pre}},\, V,\, s_{\text{post}} \,) \in \text{ST}^{wsId}$
  if wsState($wsid$) $= s_{\text{pre}}$ **and** varState($wsId$, v_1) $=$ initialized **and** ...
    and varState($wsId$, $v_{|\,V\,|}$) $=$ initialized **then**
    wsState($wsId$) $:= s_{\text{post}}$

where
  $v_x \in V,\ \ x = 1\ ..\ |\,V\,|$

Invoking operations. The SEND machine observes the state of the assigned Web service and forwards input variables to the real Web service implementation (INVOKEWEBSERVICE). This only happens under the condition that a Web service has passed an input state transition and all variables (i_1, i_2, ..., $i_{|\,I\,|}$) communicated during this state transition are still initialized. After invoking the Web service implementation, the variables communicated are marked as being processed. This prevents the invocation from reoccurring.

SEND($wsId$) $\equiv$ **do forall** $\{\, (\, I,\, s_{\text{post}} \,) :\ (\, s_{\text{pre}},\, I,\, s_{\text{post}} \,) \in \text{ST}^{wsId}_{\text{in}} \,\}$
  if wsState($wsid$) $= s_{\text{post}}$ **and** varState($wsId$, i_1) $=$ initialized **and** ...
    and varState($wsId$, $i_{|\,I\,|}$) $=$ initialized **then**
    INVOKEWEBSERVICE($wsId$, I)
    do forall $i \in I$
      varState($wsId$, i) $:=$ processed

where
  $i_x \in I,\ \ x = 1\ ..\ |\,I\,|$

Receiving operation responses. The RECEIVE machine forwards variables received from a real Web service implementation to our Web service model. The appropriate time for this is when a Web service execution stands right before an output state transition (s_{pre}), and the Web service implementation has produced the output already. The result of the receiving is that the variables communicated are marked as initialized. This prevents the RECEIVE machine from acting again and triggers the EXECUTE machine.

RECEIVE($wsId$) $\equiv$ **do forall** $\{\, (\, s_{\text{pre}},\, O \,) :\ (\, s_{\text{pre}},\, O,\, s_{\text{post}} \,) \in \text{ST}^{wsId}_{\text{out}} \,\}$
  if wsState($wsid$) $= s_{\text{pre}}$ **and** RECEIVEDFROMWEBSERVICE($wsId$, O) **then**
    do forall $o \in O$
      varState($wsId$, o) $:=$ initialized

where
  $o_x \in O,\ \ x = 1\ ..\ |\,O\,|$

4 Goal definition

The objective of this work is to compose a set of business processes represented as a set of Web services provided in a repository to a collaborative business process represented as an orchestration of the participating Web services. In this section, we define the properties a correct composition has to fulfill.

We define the correctness of a composition based on the states that all participating Web services can potentially reach in the end of the execution of the orchestration. Such a set of states is called Goal. We differentiate between primary goals ($\mathrm{PrimGoal}$) and recovery goals ($\mathrm{RecGoal}$). Both types of goals are used to describe the requirements of a correct composition ($\mathrm{CompGoal}$).

Definition 6 (Correct orchestration). *An orchestration is correct if and only if the orchestration only has the following properties.*

- *Each execution results in a system state that is part of the composition goal.*
- *There must be a theoretic execution the leads to a system state defined as one of the primary goals.*

By this definition, we ensure transactionality of the Web services. One thus has the possibility to specify that either all Web services have to reach a successful state or no Web service must reaches a successful state. For our student transfer example it would be bad if the Old School successfully unregisters a student, but the New School fails in registering the student.

Definition 7 (Composition goal).

$$
\begin{aligned}
\mathrm{CompGoal} &:= \langle \mathrm{PrimGoal}, \mathrm{RecGoal} \rangle \\
\mathrm{PrimGoal} &\subseteq \mathrm{Goal} \quad \ldots \text{ set of primary goals} \\
\mathrm{RecGoal} &\subseteq \mathrm{Goal} \quad \ldots \text{ set of recovery goals} \\
\mathrm{Goal} &:= 2^{\{\, \mathit{wsId} \mapsto \mathit{wsState} \colon \mathit{wsId} \in \mathrm{ID},\, \mathit{wsState} \in (\mathrm{S}_{\mathrm{fin}}^{\mathit{wsId}} \cup \{\, s_{\mathrm{init}}^{\mathit{wsId}} \,\})\,\}}
\end{aligned}
$$

We illustrate the goal definition by giving possible primary and recovery goals for the Web services of our exemplary repository in Table 1 on the next page.

5 Orchestration model

In this section we describe our formal model of an orchestration. The execution of an orchestration involves the execution of the ASMs of the participating Web service (EXECUTE, SEND, RECEIVE). We so far left open the definition of connecting communicating Web services to each other. Such communication occurs at specific execution states. In order to keep the association between a Web service and its respective state, we define the tuple $\mathrm{WSState}$.

Definition 8 (Web service state).

$$
\mathrm{WSState} := \{\, \mathit{wsId} \mapsto \mathit{wsState} : \ \mathit{wsId} \in \mathrm{ID},\, \mathit{wsState} \in \mathrm{S}^{\mathit{wsId}} \,\}
$$

No	*Type*	User	OldSchool	HeadQuarter	NewSchool
pg1	*primary*	done	done	done	done
rg1	*recovery*	init	init	init	init
rg2	*recovery*	failed	init	init	init
rg3	*recovery*	init	failed	init	init
rg4	*recovery*	failed	failed	init	init
rg5	*recovery*	init	cancelled	init	init
rg6	*recovery*	failed	cancelled	init	init
rg7	*recovery*	init	init	failed	init
rg8	*recovery*	failed	init	failed	init
rg9	*recovery*	init	failed	failed	init
rg10	*recovery*	failed	failed	failed	init
rg11	*recovery*	init	cancelled	failed	init
rg12	*recovery*	failed	cancelled	failed	init
rg13	*recovery*	init	init	cancelled	init
rg14	*recovery*	failed	init	cancelled	init
rg15	*recovery*	init	failed	cancelled	init
rg16	*recovery*	failed	failed	cancelled	init
rg17	*recovery*	init	cancelled	cancelled	init
rg18	*recovery*	failed	cancelled	cancelled	init
rg19	*recovery*	init	init	init	failed
rg20	*recovery*	failed	init	init	failed
rg21	*recovery*	init	failed	init	failed
rg22	*recovery*	failed	failed	init	failed
rg23	*recovery*	init	cancelled	init	failed
rg24	*recovery*	failed	cancelled	init	failed
rg25	*recovery*	init	init	failed	failed
rg26	*recovery*	failed	init	failed	failed
rg27	*recovery*	init	failed	failed	failed
rg28	*recovery*	failed	failed	failed	failed
rg29	*recovery*	init	cancelled	failed	failed
rg30	*recovery*	failed	cancelled	failed	failed
rg31	*recovery*	init	init	cancelled	failed
rg32	*recovery*	failed	init	cancelled	failed
rg33	*recovery*	init	failed	cancelled	failed
rg34	*recovery*	failed	failed	cancelled	failed
rg35	*recovery*	init	cancelled	cancelled	failed
rg36	*recovery*	failed	cancelled	cancelled	failed

Table 1. Exemplary goals. The table shows goals that one might want to define for the Web services shown in Fig. 3 on page 4. The recovery goals cover all possible combinations of unsuccessful, final states and thus state transactionality of all Web services as requirement for successful composition.

We model the orchestration of a set of Web services as a set of copy rules that fire at certain states of the Web service executions. The set of Web services' states contained in a copy rule can be understood as part of the rule's firing condition. In addition, a copy rule contains a set of variable assignments (VarAss). They are to be interpreted as the actions to be executed upon the rule's firing.

Definition 9 (Copy rule).

$$\text{CopyRule} := \langle S, A \rangle, \quad S \subseteq \text{WSState}, \quad A \subseteq \text{VarAss}$$

Based on the ASM interpretation denoted above, we define the ASM model for a set of copy rules as follows. In addition to the contents of a rule's condition and update described above, we have to take care that a copy rule does not fire twice. Therefore, we enrich its condition by a check whether all input variables of VarAss are still undefined (undef) and set their state to initialized in the rule's updates. This will trigger the EXECUTE machine of the receiving Web service.

$\textsc{Copy}(\mathit{rules} \subseteq \text{CopyRule}) \equiv$ **do forall** $(\mathit{states}, \mathit{varAss}) \in \mathit{rules}$
  if $\text{wsState}(\mathit{wsId}_1) = s_1$ **and** $\ldots$ **and** $\text{wsState}(\mathit{wsId}_{|\mathit{states}|}) = s_{|\mathit{states}|}$
    and $\text{varState}(\mathit{wsId}_{\text{out}_1}, o_1) = \text{initialized}$ **and** $\ldots$
    and $\text{varState}(\mathit{wsId}_{\text{out}_{|\mathit{varAss}|}}, o_{|\mathit{varAss}|}) = \text{initialized}$
    and $\text{varState}(\mathit{wsId}_{\text{in}_1}, i_1) = \text{undef}$ **and** $\ldots$
    and $\text{varState}(\mathit{wsId}_{\text{in}_{|\mathit{varAss}|}}, i_{|\mathit{varAss}|}) = \text{undef}$ **then**
  do forall $((\mathit{wsId}_{\text{out}}, o), (\mathit{wsId}_{\text{in}}, i)) \in \mathit{varAss}$
    $\text{varState}(\mathit{wsId}_{\text{in}}, i) := \text{initialized}$

where
  $(\mathit{wsId}_k, s_k) \in \mathit{states}, \quad k = 1 \,..\, |\mathit{states}|$
  $((\mathit{wsId}_{\text{out}_n}, o_n), (\mathit{wsId}_{\text{in}_n}, i_n)), \quad n = 1 \,..\, |\mathit{varAss}|$

For illustrating the definition of copy rules, we provide an exemplary copy rule relating to the repository in Fig. 3 on page 4. The copy rule states that when the User Web service is in the state *requesting* and all other Web services are in their initial state (*init*), then the content of variable CustomerID of the User Web service is copied to the variable CustomerID of the Head Quarter Web service. For our orchestration model, the real copying of the variable content is not relevant. We only care for the changed state of the variables involved. Thus, the result of the following copy rules is that the state of the variable CustomerID of the Head Quarter Web service is initialized afterwards.

$$\begin{aligned} \mathit{copyRule}_{ps_{10}} = \; & (\{(U, \mathit{requesting}), (O, \mathit{init}), (H, \mathit{init}), (N, \mathit{init})\}, \\ & \{((U, \mathit{CustID}), (H, \mathit{CustID}))\}) \end{aligned}$$

Finally, we define our model of executing an orchestration using the ORCHESTRATE ASM. This ASM continuously invokes the Web-service-specific ASMs (EXECUTE, SEND, RECEIVE) and the ASM representation of the copy rules (COPY) in any order. The copy rules that are executed by the COPY machine are defined by the REACHCOMPGOAL ASM which is the main machine of our composition algorithm. Its details are given in the following sections.

ORCHESTRATE($cg \in \text{CompGoal}$, $A \subseteq \text{VarAss}$) $\equiv$
  iterate choose $\{ M :\ M \equiv \text{SEND}(wsId) \vee M \equiv \text{RECEIVE}(wsId)$
    $\vee\ M \equiv \text{EXECUTE}(wsId) \vee M \equiv \text{COPY}(rules),\ wsId \in \text{ids}(cg),$
    $rules = \text{REACHCOMPGOAL}(cg, A) \}$
    M

Linking to real-world execution. In principle, there are multiple ways to generate executable code for our orchestration. One could be the translation to a WSBPEL description. However, since WSBPEL follows a sequential programming paradigm, the transformation of our copy rules to WSBPEL is not straight forward. We have some preliminary ideas, but do not give a solution to this here, and leave this issue to future work. In this paper, we rather propose to directly execute the ASM interpretation of the copy rules shown above. For this, we utilize all ASM specifications given in this paper. The managing ASM would be the ORCHESTRATE machine. We will demonstrate an execution of the orchestration generated for the repository in Fig. 3 on page 4 at the end of this paper in Sect. 7.

6 Composition algorithm

This section describes the composition algorithm in detail by the a of ASMs. Each ASM represents a module of the algorithm. The dependencies of the modules are depicted in Fig. 6 on the facing page. The figure also groups the modules with respect to their purpose. Based on the grouping, we structure the content of this section as follows. Section 6.1 reduces the problem of composing complex Web services to the composition of smaller units. Section 6.2 explains how our algorithm ensures a correct composition by properly ordering the composition of the smaller units. The smaller units are then composed by the core of our composition algorithm. Section 6.3 gives a high-level overview of the core composition algorithm. After a complete example in Sect. 6.4, Sect. 6.5 presents the details of the core composition algorithm. The modules residing in the utility group of the picture are explained upon their first usage.

6.1 Dividing the composition problem

In this section, we show how to break down the composition problem into smaller pieces. The definition of these pieces bases on the different potential execution paths through the state transitions of the single Web services in reaching their final states. We call the set consisting of exactly one potential execution path of each participating Web service a variant.

Definition 10 (Variant).

$$\text{Variant} \ := \ 2^{\{\, wsId \mapsto Transs:\ wsId \in \text{ID},\ Transs \subseteq \text{ST}^{wsId} \,\}}$$

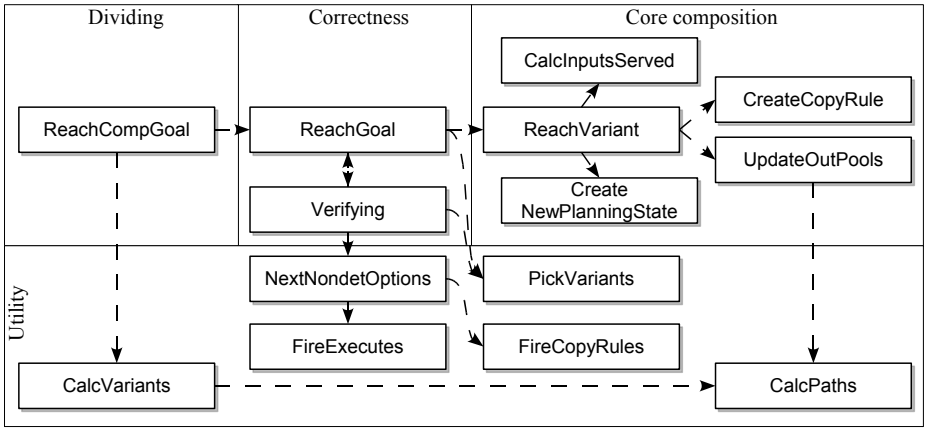

Fig. 6. Implementation modules. Each box represents a module of the implementation, each arrow denotes a module dependency. The boxes in the back group the modules based on their purpose.

The ASM starting the composition is called REACHCOMPGOAL. The purpose of the REACHCOMPGOAL machine is to initialize our composition algorithm. First, is identifies possible Web service executions (CALCVARIANTS) and hands them over to REACHGOAL. Second, it defines that the composition can only be successful if at least one primary goal can be achieved by providing the primary goals as the second parameter of REACHGOAL. It also ensures that every possible execution of the resulting copy rules ends in one of the composition goals (CompGoal) by assigning *allowedGoals* as the fourth parameter of REACHGOAL. A more detailed examination of the parameters of REACHGOAL follows in the next section. If it is not possible to generate a correct orchestration for any of the primary goals, the result is the empty set. Third, since REACHGOAL is invoked recursively for some kind of simulation that is introduced later on, we need to keep track of the current state of the simulation and thus introduce the simulation state (SimState).

Definition 11 (Simulation state).

$$\mathrm{SimState} \;:=\; 2^{\mathrm{WSState}}$$

The computation is started by calling REACHGOAL with the initial states of all Web services as a starting point (*initialSs*).

REACHCOMPGOAL($cg \in \mathrm{CompGoal}$, $A \subseteq \mathrm{VarAss}$) $\equiv$
return *copyRules* **in let**
$pgs = \mathrm{primaryGoals}(cg)$,
$initialSs = \{\, wsId \mapsto s : \; wsId \in \mathrm{ids}(pg),\, pg \in pgs,\, s = s_{\mathrm{init}}^{wsId} \,\}$
$allowedGoals = \mathrm{primaryGoals}(cg) \cup \mathrm{recoveryGoals}(cg)$
$vnts = \bigcup_{goals \,\in\, allowedGoals} \mathrm{CALCVARIANTS}(goals)$ **in**
$(fail,\, copyRules\,) :=$ REACHGOAL($vnts$, pgs, $initialSs$, $allowedGoals$, A)

The first activity of REACHCOMPGOAL is the calculation of all variants. A variant is computed as the cross product (*crossProduct*) of all possible execution paths of the Web services involved in a goal (g) to reach g.

CALCVARIANTS($g \in \text{Goal}$) $\equiv$
  return *crossProduct* **in seq**
    forall (*wsId*, s) $\in g$ **do** paths(*wsId*) := CALCPATHS(ST^{wsId}, s)
    crossProduct := $\times_{wsId \in \text{ids}(g)}$ [{ *wsId* } $\times$ paths(*wsId*)]

The possible execution paths to reach a specific state (s) of a Web service (*wsId*) are computed as follows. In the end, the paths location will contain a set of paths, where each path is a set of transition rules. In the beginning, the paths location is initialized with exactly one path consisting of the one transition that directly leads to the specified state. Now, those transitions directly leading to an existing transition in a *path* in paths are iteratively added to that *path*. The calculation is performed as long as some paths grow. Therefore, we store the overall size of all paths (calcSumOfLengths) during the preceding iteration in oldSumOfLengths.

CALCPATHS($T \subseteq \text{ST}^{wsId}$, $s \in \text{S}^{wsId}$) $\equiv$
  step
    oldSumOfLengths := 0
    paths := { { (s_{pre}, V, s_{post}) $\in T$: $s_{\text{post}} = s$ } }
  step while calcSumOfLengths(paths) > oldSumOfLengths **do**
    oldSumOfLengths := calcSumOfLengths(paths)
    forall *path* $\in$ paths **do**
      step paths := paths $\setminus$ { *path* }
      step do forall *rule* $\in$ { (s_{pre}, V_1, $s_{\text{pre}_{\text{ex}}}$) $\in T$:
          ($s_{\text{pre}_{\text{ex}}}$, V_2, $s_{\text{post}_{\text{ex}}}$) $\in$ *path* }
        paths := paths $\cup$ { *path* $\cup$ *rule* }
  step result := paths

where
  calcSumOfLengths(paths) $\equiv$
    $|\,path_1\,| + |\,path_2\,| + \ldots + |\,path_{|\,\text{paths}\,|}\,|$
  $path_x \in$ paths, $\quad x = 1 \,..\, |\,\text{paths}\,|$

6.2 Computing correct orchestrations

For one variant, the creation of copy rules can be achieved by our core composition algorithm (REACHVARIANT) which is explained in the following section. The copy rules created by REACHVARIANT ensure that the given goal can be reached in this variant. Due to potential non-deterministic behavior of the participating Web services, it may happen that the execution of the orchestration leaves one of the Web services' path along the variant, or even leave the path to its final state that is part of the defined goal. The result of our composition has to ensure that in such a case an alternative path is taken that leads to any other desired final state. This is ensured by VERIFYING. With this high-level understanding, we first go in detail through the implementation of REACHGOAL. Second, we explain VERIFYING and third, we detail the simulation of the created copy rules that is part of VERIFYING.

Reach goal. The aim of REACHGOAL is to return copy rules ensuring a correct orchestration for at least one of the given *goals* only considering the given variants (*vnts*). For this, it first identifies all variants (*goalVnt*) that lead to the *goals* (CALCVARIANTS). Second, it tries to compose each of the variants (REACHVARIANT). This results in some copy rules (regCopyRules). Third, the algorithm creates copy rules (altCopyRules) for each non-deterministic branch in the theoretic execution of regCopyRules (VERIFYING). The created copy rules either provide a correct orchestration of that branch, or VERIFYING fails (altFail). If a correct orchestration could be generated for at least one variant in the end, the corresponding copy rules (*oneVariantCopyRules*) are finally returned.

```
REACHGOAL(vnts ⊆ Variant, mandatGoals ⊆ Goal, ss ∈ SimState,
    allowedGoals ⊆ Goal, A ⊆ VarAss) ≡ return (fail, copyRules ) in
  if mandatGoals = ∅ then fail := false par copyRules := ∅
  else
    step fail := true par copyRules := ∅ par variantCopyRules := ∅
    step do forall goalVnt ∈ PICKVARIANTS(vnts, ss, mandatGoals)
      step ( regFail, regCopyRules ) :=
          REACHVARIANT(goalVnt, finState(goalVnt), A)
      step if not regFail then
        step ( altFail, altCopyRules ) :=
            VERIFYING(vnts, goalVnt, regCopyRules, ss, allowedGoals, A)
        step if not altFail then
          fail := false
          variantCopyRules := variantCopyRules
              ∪ { filterRules(regCopyRules, altCopyRules) }
    step if variantCopyRules ≠ ∅ then
      choose oneVariantCopyRules ∈ variantCopyRules
        copyRules := oneVariantCopyRules
```

where
filterRules(C_1, C_2) := { c : $c_1 \in C_1$, $c_2 \in C_2$,
$c = \begin{cases} c_2, & \nexists\, c_1 : \text{states}(c_1) = \text{states}(c_2) \\ c_1, & \text{otherwise} \end{cases}$ }
finState(*vnt* ∈ Variant) := { (*wsId*, s) : (s_{pre}, V, s_{post}) ∈ T,
(*wsId*, T) ∈ *vnt*, ∄ (s_{post}, V, s_{next}) ∈ T }

For further computation, only those variants (*pickedVnts*) are considered out of the given variants (*vnts*) that pass the given state (*ss*) and lead to the given *goal*.

PICKVARIANTS(*vnts* ⊆ Variant, *ss* ∈ SimState, *goals* ⊆ Goal) ≡
return *pickedVnts* **in**
pickedVnts := { *vnt* ∈ *vnts* : ∀ *path* ∈ *vnt*, *wsId* = *id*(*path*),
F = transs(*path*), (s_{pre_1}, V_1, s_{post_1}) ∈ F, (*wsId*, s_{pre_1}) ∈ *ss*,
(s_{pre_2}, V_2, s_{post_2}) ∈ F, (*wsId*, s_{post_2}) ∈ *goal*, *goal* ∈ *goals* }

Verifying. Through REACHVARIANT in REACHGOAL, we ensure that a composition can be generated that steers the execution along the specific variant. However, this path of execution may depend on the non-deterministic behavior of other Web services that cause a deviation from this path. For this case, VERIFYING ensures that there exists a successful composition for each non-deterministically deviating path. The result of VERIFYING is either the set of copy rules that ensure the successful composition, or a notification of failure if no successful composition exists for all non-deterministic deviations.

We now detail the functioning of VERIFYING. In order to give the full picture, we link our description to REACHGOAL where necessary. The overall process is depicted in Fig. 7 on page 20.

1. First, REACHGOAL tries to reach a variant (REACHVARIANT).
2. Second, VERIFYING simulates the execution of the given copy rules (*cr*) starting from the given state *ss* (NEXTNONDETOPTIONS). The simulation stops at the first point of non-determinism and returns all different, non-deterministic options that can occur at the current point of execution (options).
3. Third, our objective implies that there must be a successful composition for each of the options. The different options are depicted in Fig. 7 by the multiple lines leaving "2. NO". Since each option may be reached through different variants (optionVnts), we need to ensure that there exists a successful composition for at least one of the variants for each option (*optionVnt*). Ensuring successful composition for a variant exactly is the objective of the ASM REACHGOAL. Such a call is represented by "3.2. RG" in the figure. In contrast to the initial call of REACHGOAL, we now only care that one of our *allowedGoals* can be reached. We therefore provide *allowedGoals* as second and fourth parameter of REACHGOAL. Also, we want to restrict the variants to be considered by REACHGOAL to the variants relevant for the current option (optionVnts). Finally, we provide the simulation state of the non-deterministic option (*option*) as the starting state for

REACHGOAL. Please note that the call to REACHGOAL is recursive. This is depicted by the labels starting with "3.1." in Fig. 7. In the case that REACHGOAL was successful, we collect the copy rules generated (optionCopyRules) in the return variable *copyRules*.

4. Fourth, we continue the original simulation up to the next point of non-determinism deviating from the original variant (*vnt*).
5. Fifth, each non-determinism found is elaborated as described before. The process continues for all non-deterministic deviations from the original variant.

The verifying is done when the simulation stagnates (oldss $=$ *ss*) or the generation of alternative copy rules fails (globalFail). Stagnation may happen when simulation reached an *allowedGoal*. In this case, we return the collected *copyRules*. In every other case of stagnation and in any case of failure, we return an empty set of *copyRules* and a failure notification (*fail*).

```
VERIFYING(vnts ⊆ Variant, vnt ∈ Variant, cr ⊆ CopyRule, ss ∈ SimState,
    allowedGoals ⊆ Goal, A ⊆ VarAss) ≡ return (fail, copyRules) in
  step
    oldss := ∅
    globalFail := false
  step while oldss ≠ ss and not globalFail do
    step
      oldss := ss
      (ss, options) := NEXTNONDETOPTIONS(vnts, vnt, cr, ss)
    step if oldss ≠ ss then do forall option ∈ options
      step optionVnts := PICKVARIANTS(vnts, option, allowedGoals)
      step
        if optionVnts ≠ ∅ then
          step (optionUncomposable, optionCopyRules) :=
              REACHGOAL(optionVnts, allowedGoals, option, allowedGoals, A)
          step
            copyRules := copyRules ∪ optionCopyRules
            if optionUncomposable then globalFail := true
        else globalFail := true
  step
    if not globalFail and ss ∈ allowedGoals then fail := false
    else
      fail := true
      copyRules := ∅
```

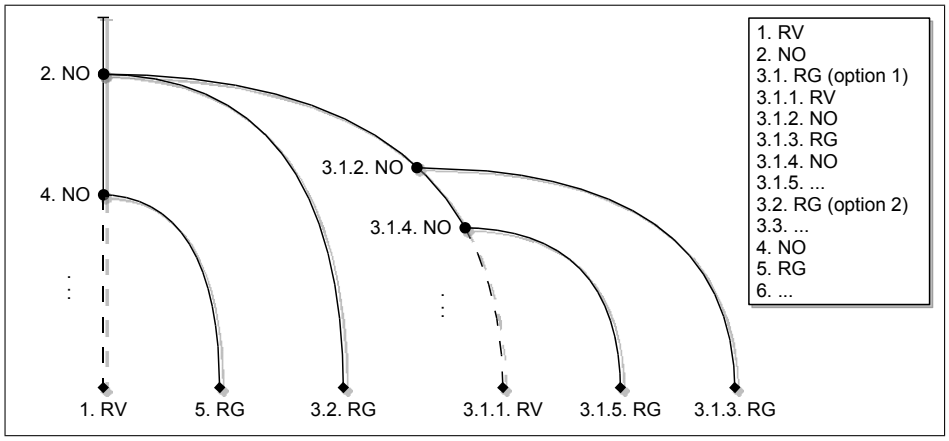

Fig. 7. Recursive computation of REACHGOAL. The abbreviations represent the modules REACHVARIANT (RV), NEXTNONDETOPTIONS (NO), and REACHGOAL (RG). A circle or diamond next to an abbreviation denotes the simulation state that was the input or the output of the respective module. The perpendicular line end denotes the initial simulation state. Each circle denotes a simulation state prior to some non-deterministic options. Each diamond denotes a goal. A solid line represents copy rules ensuring a correct, partial orchestration from its upper to its lower simulation state. A dashed line stands for copy rules to be elaborated on. The legend on the right hand side presents the sequence of module invocations corresponding to the picture.

Simulation. The NEXTNONDETOPTIONS machine performs a simulation of the current copy rules in order to determine the next non-determinism in their application on a real, collaborative Web service execution starting from simulation state *ss*. This is achieved by simulating the firing of all EXECUTE machines of all Web services alternated with applying the copy rules (FIRECOPYRULES).

NEXTNONDETOPTIONS(*vnts* $\subseteq$ Variant, *vnt* $\in$ Variant, *cr* $\subseteq$ CopyRule,
ss $\in$ SimState) $\equiv$ **return** (*ss*, options) **in**
step oldss := $\emptyset$ **par** options := $\emptyset$
step (*ss*, options) := FIREEXECUTES(*vnts*, *vnt*, *ss*, *cr*, $\emptyset$) // start initiator
step while oldss $\neq$ *ss* **and** options $= \emptyset$ **do**
oldss := *sss*
step *inputs* := FIRECOPYRULES(*ss*, *cr*) // these should be deterministic
step (*ss*, options) := FIREEXECUTES(*vnts*, *vnt*, *ss*, *cr*, *inputs*)

The main task when simulating the application of copy rules is to identify the inputs that are served by the copy rules applicable in the current simulation state (*ss*).

FIRECOPYRULES(*ss* $\in$ SimState, *cr* $\subseteq$ CopyRule) $\equiv$ **return** *inputs* **in**
inputs := $\{$ (*wsId*$_{\text{in}}$, *i*) : ((*wsId*$_{\text{out}}$, *o*), (*wsId*$_{\text{in}}$, *i*))
$\in$ assignments(*copyRule*), *ss* = states(*copyRule*), *copyRule* $\in$ *cr* $\}$

During simulating the EXECUTE rules of all Web services, we identify non-deterministic options as follows. First, we advance the simulation state for all input transitions with all inputs served. Second, we examine all the states (S in *ndStates*) for each Web service with non-deterministic branches that can be directly reached at the current state of simulation (*ss*) and that do not appear in the transitions (T) of the current variant (*vnt*). Third, we collect the states of the Web services without non-determinism (*detStates*). Fourth, we calculate all non-deterministic deviations from the current variant (*vnt*) by creating the cross product of the states in *ndStates* and *detStates*. Finally, we calculate the simulation state (*ss*) for each Web service that has to be evaluated by the verification after the non-deterministic options were checked. We set the next state to the directly following simulation state (s_{post}) of the current variant (*vnt*) if there are no active transition alternatives directly following the current simulation state. A transition is active in the following cases.

- It is an output transition.
- It is an input transition and all of its input variables can be served.

$$
\begin{aligned}
&\textsc{FireExecutes}(\mathit{vnts} \subseteq \mathrm{Variant},\ \mathit{vnt} \in \mathrm{Variant},\ \mathit{ss} \in \mathrm{SimState},\\
&\quad \mathit{cr} \subseteq \mathrm{CopyRule},\ \mathit{inputs} \subseteq \mathrm{Variable}) \equiv \mathbf{return}\ (\mathit{ss}, \mathit{options})\ \mathbf{in}\\
&\mathbf{step}\ \mathit{ss} := \{\, (\mathit{wsId}, s) :\ (\mathit{wsId}, s_{\mathrm{ss}}) \in \mathit{ss},\ (\mathit{wsId}, T) \in \mathit{vnt},\ t \in T,\\
&\qquad t = (s_{\mathrm{ss}}, V, s_{\mathrm{post}}),\\
&\qquad s = \begin{cases} s_{\mathrm{post}}, & V \subseteq \mathrm{IN}^{\mathit{wsId}},\ \forall i \in V : (\mathit{wsId}, i) \in \mathit{inputs}\\ s_{\mathrm{ss}}, & \text{otherwise} \end{cases} \quad \}\\
&\mathbf{step}\\
&\quad \mathit{ndStates} := \{\, (\mathit{wsId}, S) :\ (\mathit{wsId}, s_{\mathrm{ss}}) \in \mathit{ss},\ (\mathit{wsId}, T) \in \mathit{vnt},\\
&\qquad t_{\mathrm{vnt}} \in T,\ t_{\mathrm{nd}} \in \mathrm{ST}^{\mathit{wsId}},\\
&\qquad t_{\mathrm{vnt}} = (s_{\mathrm{ss}}, O_{\mathrm{vnt}}, s_{\mathrm{post}_{\mathrm{vnt}}}),\ t_{\mathrm{nd}} = (s_{\mathrm{ss}}, O_{\mathrm{nd}}, s_{\mathrm{post}_{\mathrm{nd}}}),\\
&\qquad s_{\mathrm{post}_{\mathrm{vnt}}} \neq s_{\mathrm{post}_{\mathrm{nd}}},\ s_{\mathrm{post}_{\mathrm{nd}}} \in S,\ O_{\mathrm{vnt}}, O_{\mathrm{nd}} \in \mathrm{OUT}^{\mathit{wsId}}\,\}\\
&\quad \mathit{detStates} := \{\, (\mathit{wsId}, S) :\ (\mathit{wsId}, s_{\mathrm{ss}}) \in \mathit{ss},\ (\mathit{wsId}, T) \in \mathit{vnt},\\
&\qquad t_{\mathrm{vnt}} \in T,\ t_{\mathrm{vnt}} = (s_{\mathrm{ss}}, V_{\mathrm{vnt}}, s_{\mathrm{post}_{\mathrm{vnt}}}),\\
&\qquad [\ V_{\mathrm{vnt}} \in \mathrm{IN}^{\mathit{wsId}},\ s_{\mathrm{ss}} \in S\ ] \vee [\ V_{\mathrm{vnt}} \in \mathrm{OUT}^{\mathit{wsId}},\ \nexists t_{\mathrm{nd}} \in \mathrm{ST}^{\mathit{wsId}},\\
&\qquad t_{\mathrm{nd}} = (s_{\mathrm{ss}}, O_{\mathrm{nd}}, s_{\mathrm{post}_{\mathrm{nd}}}),\ s_{\mathrm{post}_{\mathrm{vnt}}} \neq s_{\mathrm{post}_{\mathrm{nd}}},\ s_{\mathrm{post}_{\mathrm{vnt}}} \in S\ ]\,\}\\
&\mathbf{step}\\
&\quad \mathit{options} := \times_{(\mathit{wsId}, S) \in (\mathit{ndStates} \cup \mathit{detStates})} [\ \{\mathit{wsId}\} \times S\ ]\\
&\quad \mathit{ss} := \{\, (\mathit{wsId}, s) :\ (\mathit{wsId}, s_{\mathrm{ss}}) \in \mathit{ss},\ (\mathit{wsId}, T) \in \mathit{vnt},\ t \in T,\\
&\qquad t = (s_{\mathrm{ss}}, V, s_{\mathrm{post}}),\ s = \begin{cases} s_{\mathrm{post}}, & V \subseteq \mathrm{OUT}^{\mathit{wsId}}\\ s_{\mathrm{ss}}, & \text{otherwise} \end{cases} \quad \}
\end{aligned}
$$

6.3 Core composition algorithm overview

In this section, we give a high-level explanation of our core composition algorithm. Our core composition algorithm works iteratively from the final states of each Web service to their initial states. Therefore, we need to keep track of the current state of the backchaining and thus introduce the planning state ($\mathrm{PlState}$).

Definition 12 (Planning state).

$$\mathrm{PlState} := 2^{\mathrm{WSState} \times \{ IN, OUT \}}$$

The composition algorithm takes the following inputs.

- A variant of the possible Web service executions, i. e. a specific execution path for each participating Web service.
- An initial planning state, derived from the given goal.
- A set of possible variable assignments.

The general idea of the composition is to create copy rules for matching outputs and inputs of different Web services in the current planning state (*ps*) and to add them to the set *copyRules* (CREATECOPYRULE). After this has been done, the planning state will proceed toward the initial states of the Web services (CREATENEWPLANNINGSTATE) and the algorithm reiterates. The composition of a variant is aborted if no valid composition could be achieved (*fail*), the planning state consists of only initial states (done) or the composition came to a dead end, i. e., the planning state remained the same for two iterations. The latter case may occur if not all output variables of a service are consumed by other services. During composition, such a Web service's planning state will not proceed any further toward its initial state.

For the creation of the copy rules in the current planning state as highlighted above, some preliminary calculations have to be performed. First, we identify all output variables of all Web services that are available for this variant ($\mathrm{outPool}$). Second, we identify all input transitions of all Web services that directly lead to the current planning state ($\mathrm{adjInTrans}$). Note that for one Web service there is exactly one such transition, because the calculation bases on a variant. Third, we match all inputs of the identified input transitions with available outputs (CALCINPUTSSERVED). The correspondences for this matching are taken from the given, possible variable assignments (*A*). After creating the copy rules, we update the $\mathrm{outPool}$ locations in order to only contain all output variables that will be consumed at a later stage of the composition.

We formally define this behavior below. In the following section, we present the advertised high-level steps in more detail.

REACHVARIANT($vnt \in$ Variant, $g \in$ Goal, $A \subseteq$ VarAss) $\equiv$
return ($fail, copyRules$) $\in$
step
oldps := $\emptyset$
fail := false
copyRules := $\emptyset$
ps := $\{ (wsId, s, m) \in \text{PlState} : (wsId, s) \in g,$
$$m = \begin{cases} OUT, & hasAdjacentOutTrans(wsId, s) \\ IN, & hasAdjacentInTrans(wsId, s) \\ \text{undef}, & s = s^{wsdl}_{\text{init}} \end{cases} \}$$
forall $wsId \in$ ids(ps) **do** outPool($wsId$) := $\{ o \in O : (s_{\text{pre}}, O, s_{\text{post}}) \in Rules, (wsId, Rules) \in vnt, O \subseteq \text{OUT}^{wsId} \}$
step while not *fail* **and not** done(ps) **and not** ps = oldps **do**
oldps := ps
step do forall $wsId \in$ ids(ps)
adjInTrans($wsId$) := $\{ (s_{\text{pre}}, I, s_{\text{post}_1}) \in Rules :$
$(s_{\text{post}_1}, O, s_{\text{post}_2}) \in Rules, I \in \text{IN}^{wsId}, (wsId, Rules) \in vnt,$
$(wsId, s_{\text{post}_2}, IN) \in \text{ps} \vee (wsId, s_{\text{post}_1}, IN) \in \text{ps} \}$
step ($fail$, currAss) :=
CALCINPUTSSERVED(ids(ps), A, adjInTrans, outPool)
step if not *fail* **then**
copyRules :=
copyRules $\cup$ { CREATECOPYRULE(ps, currAss, adjInTrans) }
step outPool := UPDATEOUTPOOLS(vnt, ps, currAss, A, outPool)
step ps :=
CREATENEWPLANNINGSTATE(vnt, ps, adjInTrans, outPool)

where
done($ps \in$ PlState) $\equiv \forall (wsId, s, m) \in ps : s = s^{wsId}_{\text{init}} \vee$
$[\exists (s^{wsId}_{\text{init}}, V, s) \in \text{ST}^{wsId}, m = IN, V \subseteq \text{OUT}^{wsId}]$
hasAdjacentOutTrans($wsId$, s_{post}) := $\exists (s_{\text{pre}}, O, s_{\text{post}}) \in \text{ST}^{wsId}_{\text{out}}$
hasAdjacentInTrans($wsId$, s_{post}) := $\exists (s_{\text{pre}}, I, s_{\text{post}}) \in \text{ST}^{wsId}_{\text{in}}$

6.4 Composition example

In this section, we demonstrate the functioning of our composition algorithm based on the example provided earlier. We walk through the ASMs and present their results. We start with the first call of REACHVARIANT triggered by REACHCOMPGOAL. The result is a set of copy rules that may lead the orchestration of the exemplary Web services to

the primary goal pg_1 as defined above.

$$\begin{aligned}
copyRule_{pg_1} = \ & (\ \{(U, requesting), (O, found), (H, found), (N, done)\}, \\
& \{((N, UpdRegInfo), (U, UpdRegInfo)), \\
& \ ((H, Customer), (O, StudID)), \\
& \ ((H, Customer), (O, SchoolID)), \\
& \ ((N, NewStudID), (H, NewStudID)), \\
& \ ((U, NewSchoolID), (H, NewSchoolID))\}\ ) \\
copyRule_{ps_8} = \ & (\ \{(U, requesting), (O, found), (H, found), (N, init)\}, \\
& \{((O, StudRegInfo), (N, StudRegInfo)), \\
& \ ((U, NewSchoolID), (N, NewSchoolID))\}\ ) \\
copyRule_{ps_9} = \ & (\ \{(U, requesting), (O, init), (H, found), (N, init)\}, \\
& \{((H, Customer), (O, StudID)), \\
& \ ((H, Customer), (O, SchoolID))\}\ ) \\
copyRule_{ps_{10}} = \ & (\ \{(U, requesting), (O, init), (H, init), (N, init)\}, \\
& \{((U, CustID), (H, CustID))\}\ )
\end{aligned}$$

Now, we simulate the execution of the copy rules above. We find out that the first non-determinism occurs in Web service H after executing $copyRule_{ps_{10}}$. The option is $\{(U, requesting), (O, init), (H, failed), (N, init)\}$. Subsequently, the reachable, allowed goals are the following.

$$\begin{aligned}
allowedGoal_1 = \ rg_8 = \ & \{(U, failed), (O, init), (H, failed), (N, init)\} \\
allowedGoal_2 = rg_{10} = \ & \{(U, failed), (O, failed), (H, failed), (N, init)\} \\
allowedGoal_3 = rg_{12} = \ & \{(U, failed), (O, cancelled), (H, failed), (N, init)\} \\
allowedGoal_4 = rg_{26} = \ & \{(U, failed), (O, init), (H, failed), (N, failed)\} \\
allowedGoal_5 = rg_{28} = \ & \{(U, failed), (O, failed), (H, failed), (N, failed)\} \\
allowedGoal_6 = rg_{30} = \ & \{(U, failed), (O, cancelled), (H, failed), (N, failed)\}
\end{aligned}$$

For a successful composition, it is required that at least one variant for each of the options can be successfully composed. Since the behavior of each Web service is represented as a tree in our example, the number of goals directly determines the number of variants. For our case, this means that at least one of the allowed goals must be successfully composable. Our algorithm finds out that composition might be possible only for rg_8. We present the resulting copy rules below.

$$\begin{aligned}
copyRule_{rg_8} = \ & (\ \{(U, requesting), (O, init), (H, failed), (N, init)\}, \\
& \{((H, Fail), (U, Fail))\}\ ) \\
copyRule_{ps_{12}} = \ & (\ \{(U, requesting), (O, init), (H, init), (N, init)\}, \\
& \{((U, CustID), (H, CustID))\}\ )
\end{aligned}$$

The simulation of the copy rules above reveals no more non-determinism. Thus, we can continue our simulation of the original copy rules. The next non-deterministic op-

tion we find is $\{ (U, \mathit{requesting}), (O, \mathit{failed}), (H, \mathit{found}), (N, \mathit{init}) \}$. The allowed, reachable goals are given below.

$$\mathit{allowedGoal}_7 = rg_{16} = \{ (U, \mathit{failed}), (O, \mathit{failed}), (H, \mathit{cancelled}), (N, \mathit{init}) \}$$
$$\mathit{allowedGoal}_8 = rg_{34} = \{ (U, \mathit{failed}), (O, \mathit{failed}), (H, \mathit{cancelled}), (N, \mathit{failed}) \}$$

From the goals above, only rg_{16} can be reached. We give the respective copy rules below.

$$\begin{aligned}
\mathit{copyRule}_{rg_{16}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{failed}), (H, \mathit{found}), (N, \mathit{init}) \}, \\
& \{ ((O, \mathit{Fail}), (U, \mathit{Fail})) \}) \\
\mathit{copyRule}_{ps_{14}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{init}), (H, \mathit{found}), (N, \mathit{init}) \}, \\
& \{ ((H, \mathit{Customer}), (O, \mathit{SchoolID})), \\
& \quad ((H, \mathit{Customer}), (O, \mathit{StudID})) \}) \\
\mathit{copyRule}_{ps_{15}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{init}), (H, \mathit{init}), (N, \mathit{init}) \}, \\
& \{ ((U, \mathit{CustID}), (H, \mathit{CustID})) \})
\end{aligned}$$

From simulating the copy rules above, we find out that there is no more non-determinism. Thus, we continue the simulation of the original copy rules and find the last non-deterministic option ($\{ (U, \mathit{requesting}), (O, \mathit{found}), (H, \mathit{found}), (N, \mathit{failed}) \}$). We give the only reachable, allowed goal and the copy rules resulting from its composition below. The copy rules for this option do not contain any new non-determinism.

$$\mathit{allowedGoal}_9 = rg_{36} = \{ (U, \mathit{failed}), (O, \mathit{cancelled}), (H, \mathit{cancelled}), (N, \mathit{failed}) \}$$

$$\begin{aligned}
\mathit{copyRule}_{rg_{36}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{found}), (H, \mathit{found}), (N, \mathit{failed}) \}, \\
& \{ ((N, \mathit{Fail}), (U, \mathit{Fail})) \}) \\
\mathit{copyRule}_{ps_{17}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{found}), (H, \mathit{found}), (N, \mathit{init}) \}, \\
& \{ ((O, \mathit{StudRegInfo}), (N, \mathit{StudRegInfo})), \\
& \quad ((U, \mathit{NewSchoolID}), (N, \mathit{NewSchoolID})) \}) \\
\mathit{copyRule}_{ps_{18}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{init}), (H, \mathit{found}), (N, \mathit{init}) \}, \\
& \{ ((H, \mathit{Customer}), (O, \mathit{SchoolID})), \\
& \quad ((H, \mathit{Customer}), (O, \mathit{StudID})) \}) \\
\mathit{copyRule}_{ps_{19}} = \; & (\{ (U, \mathit{requesting}), (O, \mathit{init}), (H, \mathit{init}), (N, \mathit{init}) \}, \\
& \{ ((U, \mathit{CustID}), (H, \mathit{CustID})) \})
\end{aligned}$$

At this stage our algorithm has ensured that the primary goal for the example (pg_1) could be reached and there exist deterministic resolutions for each non-deterministic deviation from the intended execution path to an allowed recovery goal. Therefore we can claim that the example is successfully composable. The copy rules our algorithm returns contain the copy rules for reaching the primary goal and all non-deterministic deviations from the intended path, i. e. all copy rules shown in this section.

6.5 Core composition algorithm details

In this section, we go in detail through the individual steps of the core composition algorithm as presented in Sect. 6.3 on page 22. Each of these steps is presented as an ASM.

Input and output assignments. The matching of input variables in $\mathrm{adjInTrans}$ and output variables in the different $\mathrm{outPools}$ is specified in CALCINPUTSSERVED. First, we build the subset of all possible variable assignments (A) that can be assigned in the current planning state ($\mathrm{currAss}$). Second, we check whether all input transitions ($\mathrm{adjInTrans}$) of all Web services can be served by the $\mathrm{currAss}$. If this is not the case, the composition of this variant has failed. This is because the $\mathrm{outPools}$ can only shrink during the iterations of REACHVARIANT. Thus, input variables that cannot be served right away, cannot be served at any time during composition.

CALCINPUTSSERVED(*services* $\subseteq \mathrm{ID}$, $A \subseteq \mathrm{VarAss}$, $\mathrm{adjInTrans}$, $\mathrm{outPool}$) $\equiv$
  return (*fail*, $\mathrm{currAss}$) **in**
    step $\mathrm{currAss} := \{ ((wsId_{\mathrm{out}}, o), (wsId_{\mathrm{in}}, i)) \in A : i \in I,$
        $(s_{\mathrm{pre_{in}}}, I, s_{\mathrm{post_{in}}}) \in \mathrm{adjInTrans}(wsId_{\mathrm{in}}), o \in \mathrm{outPool}(wsId_{\mathrm{out}}) \}$
    step
      if $\nexists ((wsId_{\mathrm{out}_1}, o_1), (wsId_{\mathrm{in}_1}, i_1)) \in A :$
          $((wsId_{\mathrm{out}_2}, o_2), (wsId_{\mathrm{in}_1}, i_1)) \in A :$
          $wsId_{\mathrm{out}_1} \neq wsId_{\mathrm{out}_2} \vee o_1 \neq o_2$ **then**
        do forall $wsId_{\mathrm{in}} \in services$
          let $unservedVars = \{ i : (s_{\mathrm{pre}}, I, s_{\mathrm{post}}) \in \mathrm{adjInTrans}(wsId_{\mathrm{in}}),$
              $i \in I, ((wsId_{\mathrm{out}}, o), (wsId_{\mathrm{in}}, i)) \notin \mathrm{currAss} \}$ **in**
            if $unservedVars \neq \emptyset$ **then** *fail* $:= \mathrm{true}$
      else *fail* $:= \mathrm{true}$

Copy rule creation. A copy rule contains information about all variable assignments that are possible in the current planning state (*ps*). The first component of a copy rule contains the states of all involved Web services that are a prerequisite for the copy rule to be applied in an execution. This is calculated as follows. The line numbers refer to CREATECOPYRULE below.

- Lines 3 and 4 ensure that only states for involved Web services are collected.
- If a Web service acts as the *source* of a variable assignment, its state must be its current planning state (s_{p}). This is, because the Web service must be in the state following the output transition in order to have this output available during execution (line 5).
- If a Web service acts as the *target* of a variable assignment, its state must be the state preceding the input transition served (line 6).

The second component of a copy rule consists of the $\mathrm{currAss}$ themselves.

CREATECOPYRULE($ps \in \mathrm{PlState}, currAss \subseteq \mathrm{VarAss}, \mathrm{adjInTrans}) \equiv$
return (*states*, *currAss*) **in**
let $states = \{ (wsId, s) \in \mathrm{WSState} : (wsId, s_{\mathrm{ps}}, m) \in ps,$
$([\ \mathrm{adjInTrans}(wsId) = \emptyset \ \wedge\ s = s_{\mathrm{ps}}\ ]$
$\vee [\ (s_{\mathrm{pre}}, I, s_{\mathrm{post}}) \in \mathrm{adjInTrans}(wsId) \ \wedge\ s = s_{\mathrm{pre}}\ ])\}$ **in**
skip

Adjust output pools. The output pools are used to keep track of available output variables for the variable assignments and to determine whether each output variable has at least been assigned once to another Web service during composition. Therefore, we remove an output variable from the output pool, only if no Web service ($wsId_{\mathrm{any}}$) relies on it in any of its input transitions (($s_{\mathrm{pre}}, I, s_{\mathrm{post}}$)) with respect to possible variable assignments (A) on the way back from the current planning state (s) to the initial state (*path*). Only if no output variables of an output rule appear in a Web service's output pool, the planning state can proceed over such an output transition rule as explained in the following section.

UPDATEOUTPOOLS($vnt \in \mathrm{Variant}$, $ps \in \mathrm{PlState}$, $currAss \subseteq \mathrm{VarAss}$,
$A \subseteq \mathrm{VarAss}, \mathrm{outPool}) \equiv$ **return** outPool **in**
do forall $\{ (wsId_{\mathrm{out}}, o) : ((wsId_{\mathrm{out}}, o), (wsId_{\mathrm{in}}, i)) \in currAss \}$
let $futureUse = \{ i \in I : ((wsId_{\mathrm{out}}, o), (wsId_{\mathrm{in}}, i)) \in A, (s_{\mathrm{pre}}, I, s_{\mathrm{post}})$
$\in path, path \in$ CALCPATHS(T, s), $(wsId_{\mathrm{any}}, T) \in vnt,$
$(wsId_{\mathrm{any}}, s, m) \in ps, ((wsId_{\mathrm{out}}, o), (wsId_{\mathrm{in}}, i))$
$\notin currAss \}$ **in**
if $futureUse = \emptyset$ **then** $\mathrm{outPool}(wsId_{\mathrm{out}}) := \mathrm{outPool}(wsId_{\mathrm{out}}) \setminus \{ o \}$

Subsequent planning state. At the end of an iteration of REACHVARIANT, the new planning state is calculated based on the current planning state (*ps*) as defined in CREATENEWPLANNINGSTATE. We differentiate the following cases which correspond to the alternatives for allocating variable s in CREATENEWPLANNINGSTATE below.

1. The planning state for a Web service in *output* mode whose all outputs of its adjacent output transition *are not* members of any Web services' outPool proceeds one step toward the initial state. The rationale for this is that a Web service must already have reached the state after an output in order that the output can be accessed by other Web services.
2. The planning state for a Web service in *output* mode where some of the outputs of its adjacent output transition *are* members of any Web service's outPool remains at the current planning state.
3. The planning state for a Web service in *input* mode proceeds one step toward the initial state.
4. If the planning state for a Web service represents its *initial* state, it remains as it is.

$$
\begin{aligned}
&\textsc{CreateNewPlanningState}(\mathit{vnt} \in \text{Variant}, \mathit{ps} \in \text{PlState}, \text{adjInTrans},\\
&\quad \text{outPool}) \equiv \textbf{return } \mathit{newPs} \textbf{ in}\\
&\textbf{step do forall } \mathit{wsId} \in \text{ids}(\mathit{ps})\\
&\quad \text{adjOutTrans}(\mathit{wsId}) := \{\, (\, s_{\text{pre}},\, O,\, s_{\text{post}}\,) \in \mathit{Rules} :\ (\, \mathit{wsId},\, \mathit{Rules}\,)\\
&\qquad \in \mathit{vnt},\, (\, \mathit{wsId},\, s_{\text{post}},\, \mathit{OUT}\,) \in \mathit{ps},\, O \in \text{OUT}^{\mathit{wsId}}\,\}\\
&\textbf{step } \mathit{newPs} := \{\, (\, \mathit{wsId},\, s,\, m\,) :\ (\, \mathit{wsId},\, s_{\text{ps}},\, m_{\text{ps}}\,) \in \mathit{ps},\\
&\qquad (\, s,\, m\,) = \begin{cases}
(\, s_{\text{ps}},\ \mathit{IN}\,) & : (\, s_{\text{pre}},\, O,\, s_{\text{post}}\,) \in \text{adjOutTrans}(\mathit{wsId}),\\
 & \quad O \cap \text{outPool}(\mathit{wsId}) = \emptyset\\
(\, s_{\text{ps}},\ \mathit{OUT}\,) & : (\, s_{\text{pre}},\, O,\, s_{\text{post}}\,) \in \text{adjOutTrans}(\mathit{wsId}),\\
 & \quad O \cap \text{outPool}(\mathit{wsId}) \neq \emptyset\\
(\, s_{\text{pre}},\, \mathit{OUT}\,) & : (\, s_{\text{pre}},\, I,\, s_{\text{post}}\,) \in \text{adjInTrans}(\mathit{wsId})\\
(\, s_{\text{ps}},\ m_{\text{ps}}\,) & : \text{adjInTrans}(\mathit{wsId}) = \emptyset,\\
 & \quad \text{adjOutTrans}(\mathit{wsId}) = \emptyset
\end{cases}\\
&\}
\end{aligned}
$$

7 Executing exemplary orchestration

In Sect. 6.4 on page 23, we have shown how our composition algorithm derives copy rules steering each possible execution of the orchestration to a desired successful or unsuccessful goal. With this information, we are now able to instantiate the ASM machines that carry out the execution of the Web services and the copy rules defined in Sect. 3.3 on page 9 and Sect. 5 on page 11. Instead of connecting the ASMs INVOKEWEBSERVICE and RECEIVEDFROMWEBSERVICE to real Web service implementations, we generate output messages for each such call for demonstration purposes. In addition, every firing of the COPY machine generates a textual output as well.

The above mentioned ASMs are directly feed into the CoreASM[5] system [3] which is used for their execution. At each point of an non-deterministic reaction of any Web service, our implementation comes up with a message box letting the user perform the choice. The result of this execution is listed in Fig. 8 on the facing page and thus exemplarily shows the correctness of the generated copy rules.

8 Conclusion and Outlook

In this paper, we have described a composition algorithm that generates correct Web service compositions respecting user-defined primary and recovery business process composition goals. We believe, primary and recovery goals are an essential property of collaborative business processes which is not addressed by most work in the area of process composition [4]. This notion was inspired by the approach of Pistore and others, e. g. [5], who use model checking on a combined FSM of all business processes involved which leads to a state explosion for realistic examples. We hope to bypass this problem by analyzing the business process descriptions directly and restricting the business processes to loop-free trees in the beginning.

[5] http://www.coreasm.org/

```
-----------------------------------------------------------
execute User: requesting
-----------------------------------------------------------
copy ps10: {HeadQuarter.CustomerID} := initialized
-----------------------------------------------------------
execute HeadQuarter: requested
-----------------------------------------------------------
send HeadQuarter: {HeadQuarter.CustomerID} := processed
-----------------------------------------------------------
receive HeadQuarter: {HeadQuarter.Customer} := initialized
-----------------------------------------------------------
execute HeadQuarter: found
-----------------------------------------------------------
copy ps9: {OldSchool.getStudRegDataSchoolID,
OldSchool.getStudRegDataStudentID} := initialized
-----------------------------------------------------------
execute OldSchool: requested
-----------------------------------------------------------
send OldSchool: {OldSchool.getStudRegDataSchoolID,
OldSchool.getStudRegDataStudentID} := processed
-----------------------------------------------------------
receive OldSchool: {OldSchool.StudentRegistrationInfo} :=
initialized
-----------------------------------------------------------
execute OldSchool: found
-----------------------------------------------------------
copy ps8: {NewSchool.StudentRegistrationInfo,
NewSchool.NewSchoolID} := initialized
-----------------------------------------------------------
execute NewSchool: requested
-----------------------------------------------------------
send NewSchool: {NewSchool.StudentRegistrationInfo,
NewSchool.NewSchoolID} := processed
-----------------------------------------------------------
receive NewSchool:
{NewSchool.UpdatedStudentRegistrationInfo} :=
initialized
-----------------------------------------------------------
execute NewSchool: done
-----------------------------------------------------------
copy pg1: {HeadQuarter.NewStudentID,
User.UpdatedStudentRegistrationInfo,
HeadQuarter.NewSchoolID, OldSchool.unregStudentStudentID,
OldSchool.unregStudentSchoolID} := initialized
-----------------------------------------------------------
execute User: done
execute HeadQuarter: done
execute OldSchool: done
-----------------------------------------------------------
send User: {User.UpdatedStudentRegistrationInfo} :=
processed send HeadQuarter: {HeadQuarter.NewStudentID,
HeadQuarter.NewSchoolID} := processed send OldSchool:
{OldSchool.unregStudentStudentID,
OldSchool.unregStudentSchoolID} := processed
-----------------------------------------------------------
```

Fig. 8. Exemplary execution of an orchestration.

There are four aspects we plan to work on in the future. First, the mathematically founded definition of our algorithm should allow us to formally prove important properties of our algorithm, e. g. that the orchestrations generated are always correct with respect to our correctness definition. Second, we will finalize our initial ideas on a translation from copy rules to WSBPEL. Third, we will extend our composition algorithm for the handling of arbitrary acyclic graphs. We will also have a look into loop handling. And fourth, we will complete the implementation of the composition system which will allow us to compare its execution time to the performance of related approaches.

References

1. Object Management Group: UML v. 2.0 specification. OMG (2003)
2. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Berlin, Heidelberg (2003)
3. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: 12th Int'l Workshop on Abstract State Machines, Paris, France (March 2005)
4. Rao, J., Su, X.: A survey of automated web service composition methods. In Cardoso, J., Sheth, A.P., eds.: SWSWPC. Volume 3387 of Lecture Notes in Computer Science., Springer (2004) 43–54
5. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated composition of web services by planning at the knowledge level. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI, Professional Book Center (2005) 1252–1259