Considering Realistic Web Service Features for Semi-Automatic Composition.*

Jens Lemcke and Andreas Friesen

SAP Research CEC Karlsruhe, SAP AG, 76131 Karlsruhe, Germany
{firstname.lastname}@sap.com

Abstract. This paper extends the Web service composition engine that we developed in an earlier work. The composition engine is capable of preserving transactional requirements of Web services in the generated orchestration. In this paper, we show that relevant behavioral features of realistic Web services can be explicated by the modeling constructs the formal model underlying our composition approach provides. These features include non-deterministic outcomes of Web service operations, transactional vs. informational behavior, silent Web service abortion, and different strategies to handle acknowledgments.
In addition, we present two slight changes in the working of the composition unit that were needed to allow for some of the named realistic features and thus provide a greater flexibility for the application of our composer. The theoretic concepts are illustrated by our prototype implementation, and the execution of the composition result in a real Web service environment.

1 Introduction

Flexible collaboration and fast integration of different companies are key factors of a successful business. *Service-oriented architectures* (SOA) provide a standard means to communicate via providing and consuming *operations* of *Web services* defined in an *executable Web service description* language, e. g., WSDL.[1] However, integrating the IT systems of different companies—i. e., creating a *Web service orchestration*—remains a highly costly, and mostly manual effort.

The complexity of the integration task results from several reasons which have been only partly addressed by traditional Web service research [1]. First, Web services abstract a company's IT system which actually is an interface to a running *business process* in the company. Therefore, a Web service has to be understood as a complex workflow of Web service operations that could be formulated in WSBPEL,[2] UML activity diagrams [2], or SAWSDL.[3] For this reason, treating Web service operations as atomic actions for integration [3] does not suffice. Second, the response of a Web service invocation can usually not be influenced or foreseen by the invoker. Some work

* This material is based upon work partially supported by the EU funding under the project FUSION (FP6 - 027385). This paper reflects the author's views and the community is not liable for any use that may be made of the information contained therein.

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

[2] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[3] http://www.w3.org/2002/ws/sawsdl/spec/

deals with this by creating an integration that needs to be adapted at run time when those exceptions occur [4]. However, this puts additional load on the engine executing the orchestration—i. e., the *orchestration engine*—and neglects to resolve potential issues already during the initial integration. Third, Web services may be transactional. That is, they perform some action that cannot be undone. Therefore, the definition and proper treatment of desired *successful*, and acceptable *unsuccessful* executions of the orchestration are essential when integrating real Web services.

Our approach is most closely related to [5] which considers all the aspects mentioned above. Our work is however more light-weight, because it does not require a complete logical model of the Web service behavior as [5] does. Our approach is very close to existing Web service definitions and requires only the minimal addition of a state annotation which can be done in SAWSDL. In this paper, we review the process of designing and running a collaborative business process as originally laid out in [6] and thereby detail how our model is capable of observing realistic Web service features and how our composer—with slight improvements—preserves them in the generated orchestration. The rest of this paper is structured based on figure 1. We quickly go through the actions in the diagram and explain the data entities later in the text.

We differentiate between design time and run time activities. At *design time*, the Web service provider firstly converts the executable Web service definitions into our internal *Web service model*. During this task, the Web service provider must correctly model the features the Web services provide. The realistic features we allow to model will be detailed in section 2. Based on this model, an orchestration designer secondly defines the requirements a generated orchestration must fulfill. This is detailed in section 3. From these requirements, the automatic composer generates a collaborative business process if possible. The slight adaption to the composer that was needed to respect some of the realistic Web service features are sketched in section 4. At this point, we switch our view to the *run time*. The executable Web service descriptions need to be deployed on some Web server where they can be accessed from other peers in the network. The output of our composer will be deployed on an orchestration engine. A possible example for this deployment was given in [6] via a set of *abstract state machines* (ASM) [7]. We use the same set of ASMs for the actual execution of the generated collaborative business process in this paper. We present the run-time view of all participants in section 5.

2 Modeling Web services

A Web service consists of *operations*. Each operation transports at least one *message*. An operation can be of one of the following types.

- A *one-way* operation consist of an input, only.
- A *notification* operation only contains a single output.
- A *request-response* operation expects an input before it sends an output. Instead of the second message, a fault message can be communicated.
- 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.

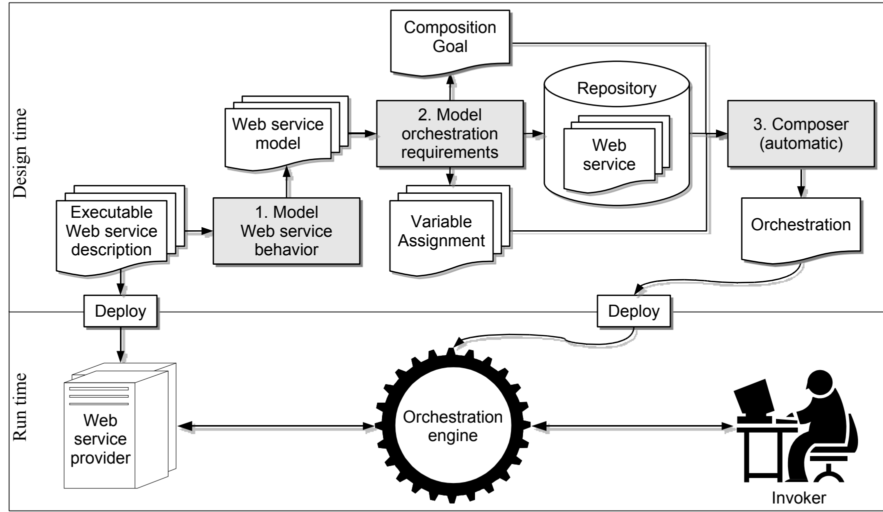

Fig. 1. Architecture for designing and running a collaborative business process.

We restrict the Web services to those whose operations cannot execute concurrently. In the following, we detail the possible behavior specifications on top of Web service operations.

2.1 Building blocks for behavior modeling

A Web service including behavior is defined via *states* (S) and *state transitions* (ST). Each state transition has either *input* or *output variables* attached (IN and OUT). A state transition with an input variable stands for the receipt of an *input message*. A state transition with an output variable stands for the sending of an *output message*. Some states may be defined to be *successful* ($\mathrm{S_{suc}}$), or *unsuccessful, final states* ($\mathrm{S_{fail}}$). The definition of a successful, final state allows the Web service provider to indicate where the execution of a Web service ends after serving its purpose. On the other hand, an unsuccessful, final state marks the end of a Web service execution without reaching its objective. Figure 2 shows how the operations in the beginning of section 2 are expressed using the described model.

$$
\begin{aligned}
\mathrm{WebService} &:= \langle \mathrm{IN},\ \mathrm{OUT},\ \mathrm{S},\ s_{\mathrm{init}},\ \mathrm{S_{suc}},\ \mathrm{S_{fail}},\ \mathrm{ST}\rangle \\
\mathrm{S},\ \mathrm{IN},\ \mathrm{OUT} &\ \ldots\ \text{sets of states, input and output variables} \\
s_{\mathrm{init}} &\in \mathrm{S} \\
\mathrm{S_{fin}} &:= \mathrm{S_{suc}} \cup \mathrm{S_{fail}},\ \mathrm{S_{suc}} \cap \mathrm{S_{fail}} = \emptyset \\
\mathrm{S_{suc}}, \mathrm{S_{fail}} &\subset \mathrm{S} \\
\mathrm{ST} &: \mathrm{S} \times ([\ 2^{\mathrm{IN}} \ \cup\ 2^{\mathrm{OUT}}\ ] \setminus \{\emptyset\}) \rightarrow \mathrm{S}
\end{aligned}
$$

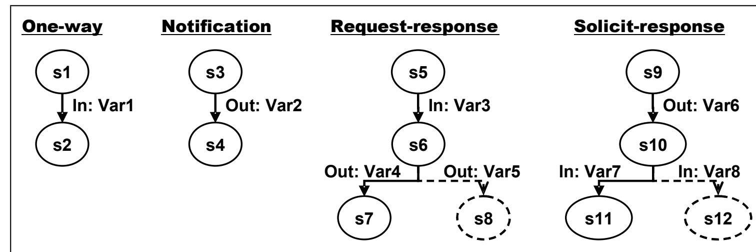

Fig. 2. General Web service operations. A circle denotes a state, a line represents a state transition. A label attached to a line stands for the communication of a variable. The label of an input transition begins with "In:". Output transition labels start with "Out:". Dotted lines and circles express that the diagram without these elements would also be valid.

Operation concatenations. Operations of a Web service can be concatenated by sharing states in the state transition model. We do however neither allow directed nor undirected cycles in the state transitions graph. In general, we differentiate two kinds of Web services: *initiators* and *providers*.

Initiators. Every collaborative business process must be invoked by some user. The user sees the initiator like a regular, *passive* Web service that can be triggered by sending an initial message that starts the orchestration transparently to the user. In contrast, the other participants of the collaborative business process see the initiator as the only *proactive* Web service among themselves that later on starts the execution of a collaborative business process. This is the view we take in this paper. Consequently, the following defines how the Web service model of an initiator must look.

1. The first operations of an initiator must be one or more notification operations.
2. Each final operation of an initiator must be a one-way operation.
3. Every other operation of an initiator must be a request-response operation.

The User Web service in figure 3 is an example of an initiator. We call the other Web services in the picture providers.

Providers. Every concatenation of operations of a provider Web service must adhere to the following definitions.

1. The first operations of a provider must be one or more one-way or request-response operations.
2. Every first one-way operation must also be a final operation of this provider.
3. Each final operation of a provider must be a one-way or a request-response operation.
4. Every other operation of a provider must be a request-response operation.

State transition restrictions. The definitions of allowed concatenations of Web service operations above reflect in the following restrictions that complete the definition of the state transitions forming the Web service model.

- The states and state transitions form a directed tree.
- Input and output transitions always alternate.

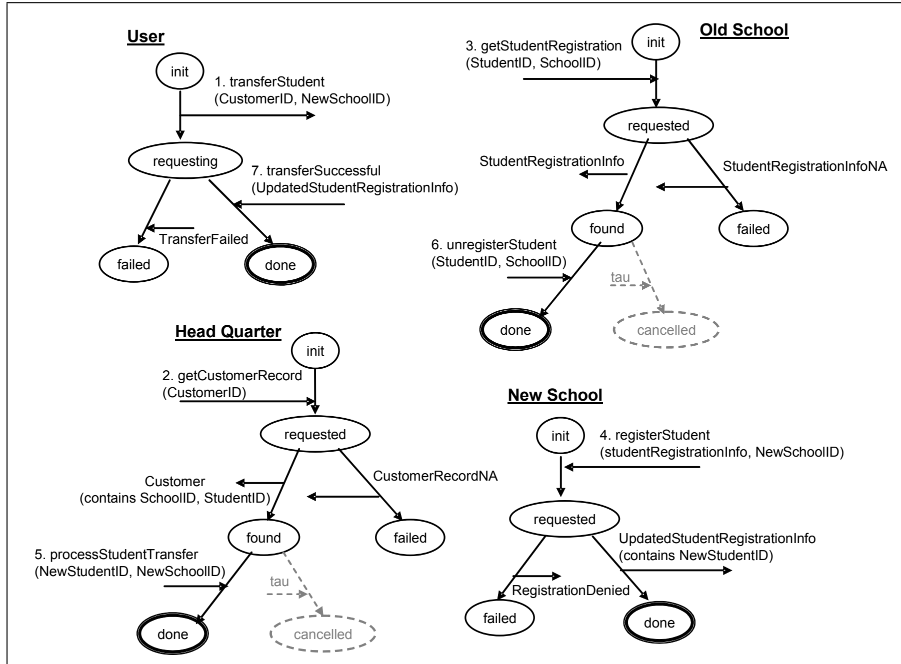

Fig. 3. Example participants of a collaborative business process. Underlined text denotes the ID of a Web service. Each graph describes a Web service's behavior as a finite state machine. Thereby, ellipses denote states, arrows between states denote state transitions, and arrows leaving or arriving at state transitions denote output or input messages. Text at messages starting with a capital letter denotes a variable name.

Implicit abortion. Web service providers sometimes assume that it is alright to leave a Web service in an intermediary state. The started interaction just times out and the Web service can deal with this. In our model, this would mean that the Web service can be put to an final, unsuccessful state from this particular, intermediary state. The specialty of this state transition is that it does not require any communication. For the purpose to model such behavior, we invent "tau"-transitions. A "tau"-transition will be fired if needed to reach some recovery goal for the respective Web service. Examples can be found in figure 3. The following restrictions apply for "tau"-transitions.

- A "tau"-transition can only be placed instead of an input transition.
- A "tau"-transition must directly lead to a final, unsuccessful state.
- There must be alternative input transitions for a "tau"-transition.
- A "tau"-transition must not start from a final state.
- There must be no transition following a "tau"-transition.

2.2 Modeling determinism

We differentiate Web service behavior based on who is in control of the process flow. In the case that the invoker of an operation is in control, we talk about *deterministic*

behavior. In the case that the subsequent process flow depends on the Web service implementation, we talk about *non-deterministic* behavior. We lay out how specific operations and their concatenations can impose non-deterministic behavior in the following. Examples for both may be found in figure 3.

Determinism of operations. Every request-response operation is deterministic if it contains exactly one output transition. Every one-way operation is deterministic as well. A request-response operation is non-deterministic if it contains more than one output transition. Through triggering the input transition, the invoker may decide whether an operation should be started or not. We call this a deterministic decision. In the case a Web service has multiple ways to answer an invocation, the invoker cannot know upfront which of the alternatives the actual answer will be. We call this non-deterministic behavior. A notification operation per se is deterministic, because the invoker knows that a message will be sent. However, a notification operation may participate in non-deterministic behavior as we show in the next section.

Determinism of operation concatenations. In contrast to the former, we now look at concatenations of operations. Non-determinism results from concatenations of operations only when at least two notification operations start in the same state. In this situation, the invoker cannot know which notification the Web service would send. To sum up the discussion, the identification of non-deterministic behavior is simple when we look at the state transition model.

- Every output state transition that starts from the same state as another output state transition is non-deterministic.
- Every other state transition is deterministic.

2.3 Transaction mechanisms

In general, we differentiate between *informational* and *transactional* Web services. A transactional Web service operation usually has some side effects that the invoker would not like to observe if the operation was not explicitly called and an execution does not end up in the primary goal. For example, consider a Web service operation that takes money from your bank account and books a flight ticket for you. This shall, for example, only happen in the case that another operation successfully rented a hotel room for you. Here it is important that the participating Web service operations are either both completed or neither of them. We call this transactional behavior. In another case, a Web service could provide a currency exchange rate without side effects, that is, this Web service does not charge for its invocation. The invoker does not care whether this Web service was completed or not. We call this informational behavior.

Modeling transactional Web services. We now describe how the state annotations introduced in section 2.1 may indicate transactional behavior of Web services.

- A successful state directly after an input transition marks an informational Web service operation.

– A transactional Web service operation does not have a successful state directly after an input transition.

After defining transactional Web services, we show in the following how different features Web services provide to manage transactions can be expressed in the Web service model.

One-shot transactions. A Web service provider may have decided to not provide any resolution mechanism for a transactional Web service operation. Thus, once a one-shot transaction has started, it cannot be undone any more. An example is provided on the left hand side of figure 4. The respective, successful, final states are represented by a bold border. Please note that there is no successful, final state defined directly behind the input transition. Thus, a collaborative business process may either execute the full transition or not at all in order to reach a primary goal.

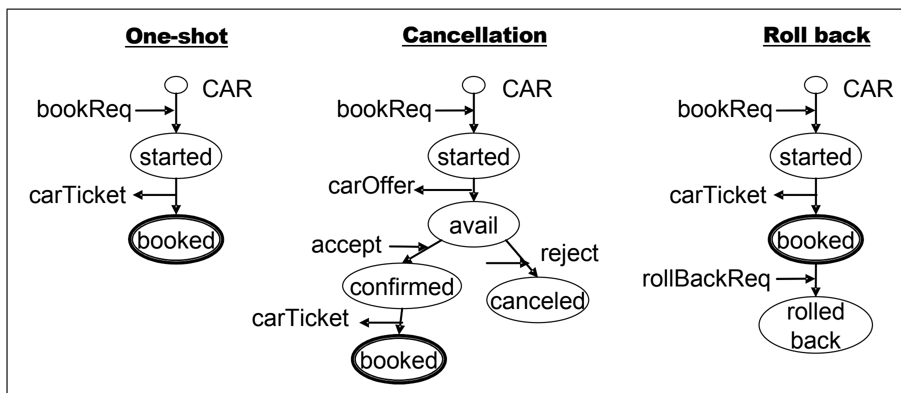

Fig. 4. Example Web service with different transaction recovery features..

Canceling transactions. One way to recover from a started transaction is to provide a cancellation feature. A Web service with cancellation feature usually does not return a booking confirmation in the first place. It will rather provide an offer that the invoker can accept or reject later on. The rejection thereby cancels the transaction. An example is shown in the center of figure 4. The cancellation feature can also be implemented using a "tau"-transition instead of the reject message in figure 4 if the explicit reject message is not required by the Web service. A "tau"-transition could also be a third alternative starting from state "avail" if the Web service provider accepts both a reject message and a silent cancellation.

Rolling back transactions. Another way to recover from a started transaction is to roll back after its completion. The model for this looks very similar to the model for one-shot Web services. First, some transaction completes. The Web service now should

be in a primary goal state. Afterwards, the system accepts a roll back message that puts the Web service to a recovery state. An example of the roll-back feature is shown on the right hand side of figure 4.

3 Modeling orchestration requirements

After detailing single Web service models, we now look at a set of Web service models that are to be composed in a later step. When modeling orchestration requirements—step 2 in figure 1—the orchestration designer performs a Web service selection and defines a repository of Web services which should be used in the orchestration. A repository must include exactly one initiator. Requirements on the orchestration that can be explicated using our model are: define allowed communications between Web services (section 3.1), mediate different ways of acknowledgment handling (section 3.2), and force transactionally correct orchestrations (section 3.3).

3.1 Basic communication

The potential communication of a set of Web services is defined via variable assignments. In general, variables are local to Web services. that means, that even variables of different Web services carrying the same names are distinct. If the value of an output variable can be taken from one Web service and used as the value for an input variable of another Web service, a variable assignment needs to be defined explicitly. A variable assignment thus consists of naming the sending Web service and the source output variable, plus the name of the receiving Web service and its target input variable. The following is an example of explicitly assigning a flight ticket sent by a flight booking Web service to the initiator.

$$[\ (\mathsf{FLIGHT}, \mathsf{flightTicket}) \ \rightarrow \ (\mathsf{USER}, \mathsf{flightTicket})\ ]$$

The definition of variable assignments affects the generated orchestration in the following way. Paths in the state transition model of a Web service that start with an *input* transition that consumes at least one variable for which no variable assignment is defined, will never be executed when the collaborative business process runs.

The definition of variable assignments strongly corresponds to the purpose of the participating Web services. In one orchestration, there must only be one provider for one piece of information. In terms of modeling constructs, this means that there must be no two variable assignments mapping different output variables on the same input variable. In the case that multiple Web services provide the same information, a disambiguation has to occur by the orchestration designer. We do not consider this decision part of the composition. In addition, we disallow malformed Web services consuming their own outputs. Thus, there must be no variable assignment mapping an output variable of one Web service to an input variable of the same Web service.

3.2 Acknowledgment handling

Web service providers may have different policies how they handle acknowledgments. Some may send acknowledging confirmation messages, but other Web services may not

expect those in response. Through variable assignments, the orchestration designer has the possibility to neglect acknowledgment messages not needed by other participants in the orchestration. In particular, *output* variables for which no variable assignment was defined, will be received by the collaborative business process, if the sending transition is executed when the collaborative business process runs. However, they will in no case be forwarded to any participant in the collaborative business process.

3.3 Transactionally correct orchestrations

The correctness of a composition can be defined 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* ($\mathrm{RecGoal}$). The state annotation of successful and unsuccessful, final states described in section 2.1 may be understood as a suggestion that mainly successful, final states should be used for the definition of a primary goal, and mainly unsuccessful, final states should be used in a recovery goal. Both types of goals are used to describe the requirements of a correct orchestration ($\mathrm{CompGoal}$). We define an orchestration to be correct if and only if it 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 reach a successful state. This property is especially important for transactional Web services as defined in section 2.3. In a student transfer use case, it would be bad if the current (old) school successfully unregistered a student, but the desired (new) school failed in registering the student.

$$\mathrm{CompGoal} := \langle \mathrm{PrimGoal}, \mathrm{RecGoal} \rangle$$

We illustrate the goal definition by giving possible primary and recovery goals for our example Web services in table 1.

4 Composer

The composer—step 3 in figure 1—creates an orchestration in terms of *copy rules*. Copy rules will be detailed in section 4.1. In section 4.2, we briefly sketch the composer and line out changes to the original specification in [6] that were made in order to cover the features described in section 2 and 3. Section 4.3 illustrates the changed algorithm by composing the Web services of our example.

4.1 Orchestration model

An orchestration consists of directions when a specific communication between two Web services has to occur. Such a statement is represented in our model by a copy

No	*Type*	User	OldSchool	HeadQuarter	NewSchool
pg1	*primary*	done	done	done	done
rg8	*recovery*	failed	init	failed	init
rg10	*recovery*	failed	failed	failed	init
rg12	*recovery*	failed	cancelled	failed	init
rg16	*recovery*	failed	failed	cancelled	init
rg26	*recovery*	failed	init	failed	failed
rg28	*recovery*	failed	failed	failed	failed
rg30	*recovery*	failed	cancelled	failed	failed
rg34	*recovery*	failed	failed	cancelled	failed
rg36	*recovery*	failed	cancelled	cancelled	failed

Table 1. Exemplary goals.

rule. A copy rule consists of the set of all participating Web services' states and a set of variable assignments that will be used to transfer data at the respective time in execution.

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

The following is an example for a copy rule stating to copy the value of CustomerID from the User to the Head Quarter Web service when all Web services are in their initial state and the User Web service has started (see figure 3).

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

4.2 Changes to the composer

The composer is specified via a set of ASM machines whose names and interactions are depicted in figure 5 [6]. The entry point is the machine REACHCOMPGOAL which splits the composition problem to smaller pieces that we call *variants*. Each variant describes one way through the state transition graphs of all participating Web services to reach a primary goal. The machine REACHGOAL actually ensures that such a way can be composed and is safe to execute, if every potential non-determinism can be resolved by proper copy rules. The former of these tasks is performed by the REACHVARIANT machine. This machine originally did a backchaining trying to reach the initial state of every Web service considering their communications when starting from the goal. This is okay at the first invocation of REACHVARIANT. However, upon generation of copy rules for one variant, REACHGOAL calls VERIFYING in order to check for proper resolving strategies for each potential non-deterministic deviation from this path. For the proof, VERIFYING now recursively utilizes REACHGOAL.

The first change we incorporated is that the backchaining of REACHVARIANT does not end when it hits the initial state, but rather when it reaches the global state from which the former REACHGOAL was invoked. This possesses the advantage that messages sent in execution oder before the non-deterministic deviation point are not required to be consumed in the recovery branch. When we consider figure 3, this releases

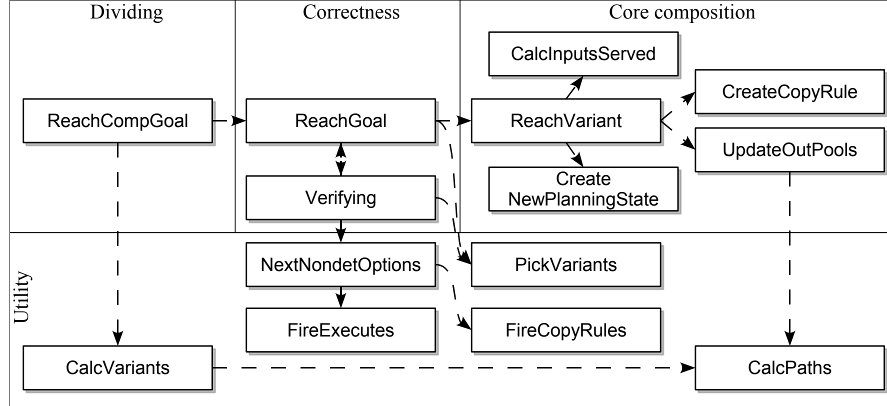

Fig. 5. ASM Modules of the composer. Each box represents a module of the implementation, each arrow denotes a module dependency. The boxes in the back group the modules based on their purpose.

us from having to catch NewSchoolID from the User Web service if the Head Quarter does not know the customer. For this change, it was necessary to adapt the done criterion in REACHVARIANT.

The second change we incorporated is that an output variable without variable assignment does not break the composition any more. This was necessary to allow the acknowledge handling that we described in section 3.2. For this, we mainly adapted the handling of output variables in REACHVARIANT.

4.3 Example

We now illustrate the composition of the exemplary Web services given in figure 3. The first call of REACHVARIANT is 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.

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

$$
\begin{aligned}
copyRule_{ps_9} \;=\;& (\{\,(\,U,\ requesting\,),\ (\,O,\ init\,),\ (\,H, found\,),\ (\,N,\ init\,)\,\},\\
& \{\,(\,(\,H,\ Customer\,),\ (\,O,\ StudID\,)\,),\\
& \quad (\,(\,H,\ Customer\,),\ (\,O,\ SchoolID\,)\,)\,\}\ );\\
copyRule_{ps_{10}} \;=\;& (\{\,(\,U,\ requesting\,),\ (\,O,\ init\,),\ (\,H,\ init\,),\ (\,N,\ init\,)\,\},\\
& \{\,(\,(\,U,\ CustID\,),\ (\,H,\ CustID\,)\,)\,\}\ )
\end{aligned}
$$

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 rg_8, rg_{10}, rg_{12}, rg_{26}, rg_{28} and rg_{30}.

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

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 rule below.

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

The simulation of the copy rule above reveals no more non-determinism. Thus, we can continue our simulation of the original copy rules. The next non-deterministic option we find is $\{\,(\,U,\ requesting\,),\ (\,O, failed\,),\ (\,H, found\,),\ (\,N,\ init\,)\,\}$. The allowed, reachable goals are rg_{16} and rg_{34}.

$$
\begin{aligned}
&allowedGoal_7 = rg_{16} = \{\,(\,U, failed\,),\ (\,O, failed\,),\ (\,H,\ cancelled\,),\ (\,N,\ init\,)\,\}\\
&allowedGoal_8 = rg_{34} = \{\,(\,U, failed\,),\ (\,O, failed\,),\ (\,H,\ cancelled\,),\ (\,N, failed\,)\,\}
\end{aligned}
$$

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

$$
\begin{aligned}
copyRule_{rg_{16}} \;=\;& (\{\,(\,U,\ requesting\,),\ (\,O, failed\,),\ (\,H, found\,),\ (\,N,\ init\,)\,\},\\
& \{\,(\,(\,O,\ Fail\,),\ (\,U,\ Fail\,)\,)\,\}\ )
\end{aligned}
$$

Simulating the copy rule above yields no more non-determinism. Thus, we continue the simulation of the original copy rules and find the last non-deterministic option

$(\{\,(\,U,\ requesting\,),\ (\,O, found\,),\ (\,H, found\,),\ (\,N, failed\,)\,\})$. The only reachable, allowed goal is rg_{36}. We give the copy rule resulting from its composition below. The copy rule for this option do not contain any new non-determinism.

$$\begin{aligned} allowedGoal_9 &= rg_{36} \\ &= \{\,(\,U, failed\,),\ (\,O,\ cancelled\,),\ (\,H,\ cancelled\,),\ (\,N, failed\,)\,\} \end{aligned}$$

$$\begin{aligned} copyRule_{rg_{36}} &= (\{\,(\,U,\ requesting\,),\ (\,O, found\,),\ (\,H, found\,),\ (\,N, failed\,)\,\}, \\ &\quad\ \{\,(\,(\,N,\ Fail\,),\ (\,U,\ Fail\,)\,)\,\}\,) \end{aligned}$$

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.

5 Execution

In [6], we have provided an operational semantics for our Web service model in terms of ASMs. We now instantiate these and execute them in the CoreASM[4] system [8] to cover the lower part (run time) of figure 1. For the demonstration, we choose the Web service interfaces shown in figure 3 and provide a simple implementation which we deploy on a local Web server. We annotate their primary and recovery goals shown in table 1 using a graphical tool that we built for this purpose (see figure 6). Invoking the composer on these inputs yields the copy rules presented in section 4.3. Finally, we feed the generated copy rules to the ASM definitions from [6], and run them via CoreASM.

We trace the working of the system from three different angles. First, we display the interactions of the user (see figure 7). Second, we observe the operations of the copy rules orchestrating the participating Web services (see figure 8). And third, we trace the invocation of the providers (see figure 9).

6 Conclusion

The current industry adoption of service-oriented architectures—such as IBM's "service sciences[5]" and SAP's "enterprise SOA[6]"—lets us expect an increase in the number of Web services offered in complex business environments and thus a need for their proper integration respecting their relevant features imposed by the underlying business processes. In this paper, we have shown how our Web service model is capable of expressing these features, and how our composition algorithm respects them where existing approaches fall short or require a lot of specification work on top of existing Web service definitions.

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

[5] http://www-304.ibm.com/jct09002c/university/scholars/skills/ssme/index.html

[6] http://www.sap.com/platform/esoa/index.epx

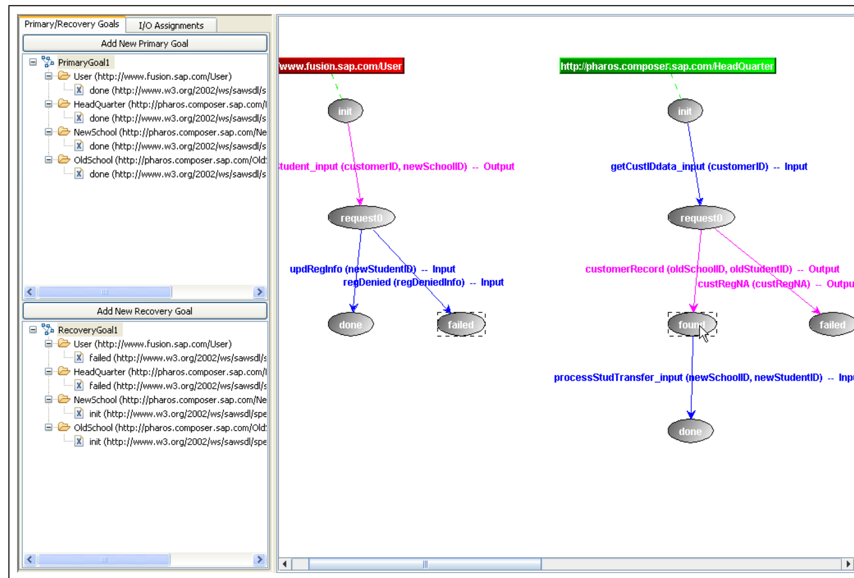

Fig. 6. UI to define a composition goal.

```
User sent customer id "1" and new school id "Uni-KA" to CBP.

User received new student id "1285476" from CBP.
```

Fig. 7. Trace of user interaction.

References

1. 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
2. Object Management Group: UML v. 2.0 specification. (2003)
3. Cardoso, J., Sheth, A.: Semantic e-workflow composition. J. Intell. Inf. Syst. **21** (2003) 191–225
4. Meyer, H., Overdick, H., Weske, M.: Plngine: A system for automated service composition and process enactment. In: Proceedings of the WWW Service Composition with Semantic Web Services (wscomps05), Compiegne, France, University of Technology of Compiegne (2005) 3–12 ISBN 2-913923-18-6.
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
6. Lemcke, J., Friesen, A.: Composing web-service-like abstract state machines (ASMs). In: IEEE SCW, IEEE Computer Society (2007) 262–269
7. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Berlin, Heidelberg (2003)
8. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: 12th Int'l Workshop on Abstract State Machines, Paris, France (2005)

```
-----------------------------------------------------------------------------------------------------------
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. Trace of orchestration engine.

```
call "getCustIDdata()" of HeadQuater is successful!
input - customer id: 1
output - old school id: Uni-KL
output - old student id: 345736

call "getStudRegData()" of OldSchool is successful!
input - old school id: Uni-KL
input - old registration info: 345736
output - student registration info: YES!

call "setStudRegistration()" of NewSchool is successful!
input - new school id: Uni-Ka
input - new registration info: YES!
output - new student id: 1285476

call "processStudTransfer()" of HeadQuater is successful!
input - new school id: Uni-Ka
input - new student id: 1285476

call "unregStudent()" of OldSchool is successful!
input - old school id: Uni-KL
input - old registration info: 345736
```

Fig. 9. Trace of provider invocations.