Journal of Universal Computer Science, vol. 14, no. 12 (2008), 2034-2058
submitted: 1/9/07, accepted: 1/6/08, appeared: 28/6/08 © J.UCS

ASMs in Service Oriented Architectures

Michael Altenhofen, Andreas Friesen, Jens Lemcke
(SAP Research, CEC Karlsruhe, 76131 Karlsruhe, Germany
{`firstname.lastname`}`@sap.com`)

Abstract: We give a survey on work we did in the past where we have successfully applied the ASM methodology to provide abstract models for a number of problem areas that are commonly found in Service Oriented Architectures (SOA). In particular, we summarize our work on (1) service behavior mediation, (2) service discovery, and (3) service composition, showing that the corresponding solutions can be described as variations of a fundamental abstract processing model—the *Virtual Provider*.

Key Words: process mediation, service discovery, workflow composition
Category: D.2.1, H.4.3, I.6.5

1 Introduction

The term Service Oriented Architecture (SOA) has gained an increasing popularity as an architectural blueprint for designing flexible information systems. SOAs are built from autonomous functional entities, *services*, which inter-operate through well-defined, message-based interfaces that abstract from both programming languages and implementation platforms.

The most prominent incarnation of this architectural blueprint is based on the technical specification set defined by the World Wide Web Consortium (W3C) where services expose their functionality as so-called *Web*[1] *services*. The corresponding W3C technical report on the Web Services Architecture [see W3C 2004] describes a basic engagement model that identifies, among others, the following three fundamental entities: the service *requester* invoking functionality from a service *provider* through message exchanges. The definition of these message exchanges has been published and thus can be retrieved through a public service *discovery service*.

Although the underlying principles are not radically new, the SOA style has given raise to a lot of research in the recent years which tries to address the following problems:

- *Service Mediation.*
  What if requester and provider do not perfectly agree on the interaction protocol? Is there a way to *mediate* [see Wiederhold 1997] in heterogeneous scenarios where mismatches may occur both on the message content and the message flow level?

[1] The technical specifications in this area make heavily use of other specifications issued by the W3C that are related to the World Wide Web, e.g. using HTTP as one possible transport protocol for Web service messages, hence the name *Web* services.

- *Service Discovery.*
  Given a number of advertised service descriptions, which one fits best the requesters' needs, both in functional and non-functional terms?
- *Service Composition.*
  How can new service functionality be exposed by composing it from available services?

For these problems, we developed formal ASM [see Börger and Stärk 2003] models that have been described individually in detail in other publications [see Altenhofen et al. 2005, Friesen and Börger 2006, Lemcke and Friesen 2007]. In this paper, we give a survey of these solutions while emphasizing their strong relationship through one fundamental abstract model, the so-called *Virtual Provider*. In [Section 2] we introduce the Virtual Provider model, which allows the construction of mediators, especially for protocol mismatches. This VP model has also formed the basis for our semantic service discovery specification which be described in [Section 3]. In [Section 4], we then report on our achievements concerning automatic service composition. Finally, we relate our work to other approaches found in the literature in [Section 5] before we conclude in [Section 6].

2 Service Mediation through Virtual Providers

2.1 Informal Model of Virtual Providers

The general architecture of the Virtual Provider (VP) component is based on the assumption that we mostly deal with two-way conversations using the Request-Reply pattern [see Hohpe and Woolf 2003]: The *service requester* sends a message (*request*) to the *provider*, which, in turn, returns a message (an *answer* or *reply*) to the requester. These two-way messages are exchanged via appropriate message *channels*. The Virtual Provider can now be seen as an intermediary component (or *proxy*) in the communication model that intercepts the message flow between the parties involved, thus being able to compensate mismatches in the interaction protocols that these different parties expect. From the requester's point of view it acts like a provider, hence the name *Virtual Provider*. This core functionality of the VP leads to the architecture in [Fig. 1], where we abstract from scheduling and message-passing functionality.

For the overall execution model we follow the SOA assumption that the message exchange between the service requester and the service provider carries all information to establish and maintain context and state, i. e., there is no out-of-band communication between the interacting parties. Processing an incoming request can only be done through *one* control structure, namely a *finite sequence* of subrequests, but with a significant modification: Each sequential subrequest can itself be further decomposed into an arbitrary large set of *parallel* outgoing requests which can be forwarded independently of each other. With this, we

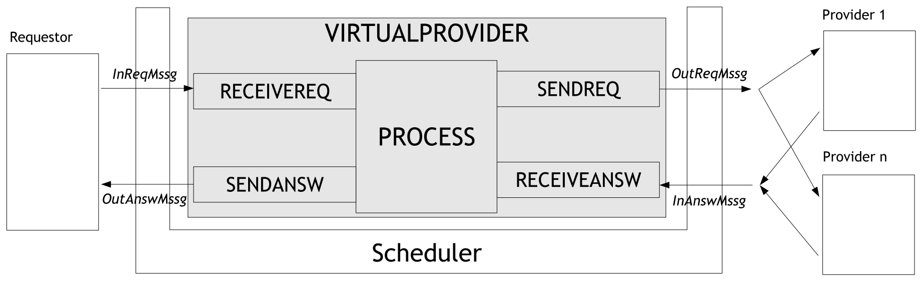

Figure 1: Virtual Provider Architecture.

end up with a simple hierarchical processing structure of so-called *seq/par trees*, that allows us to describe rich interaction schemes in a modular fashion. More sophisticated control flows can be achieved via VP composition as we will see in [Section 2.3]. Note that this way of structuring the control flow may lead to a different decomposition of the message exchange with requesters and providers than using the various control structures that are available in standard orchestration languages, such as BPEL4WS [see Andrews et al. 2003]: The overall processing sequence can be derived from the dependencies between subsequent messages (later requests may require parts from replies to former requests), but within a sequencing step i we group the outgoing requests that independently contribute to the result of i. With the expressive power of control structures in BPEL4WS, process designers are free to choose among a set of alternative approaches will eventually make it harder to identify these dependencies and, thus, to detect inconsistencies or errors.

2.2 Formal Model of Virtual Providers

Based on the architecture outlined in [Fig. 1], we have defined the VP as a modular abstract machine [see Altenhofen et al. 2005]:

```
MODULE VirtualProvider =
  **choose** $M \in \{\textsc{ReceiveReq}, \textsc{SendAnsw}, \textsc{Process}, \textsc{SendReq}, \textsc{ReceiveAnsw}\}$
    $M$
```

The first machine ReceiveReq is used to receive service request messages (elements of a set *InReqMssg* of legal incoming request messages) from requesters. Once a new incoming message has been fully received, a new internal request object is created which is appropriately initialized with an internal representation of the relevant data extracted from the request message. This particularly includes status information which is used to trigger further request processing:

```
ReceiveReq($inReqMsg$) =
  if $ReceivedReq(inReqMsg)$ then
    if $NewRequest(inReqMsg)$ then
      CreateNewReqObj($inReqMsg, ReqObj$)
    else
      let $r = prevReqObj(inReqMsg)$ in
        RefreshReqObj($r, inReqMsg$)
```

Request processing is handled in the Process submachine, which builds up and processes the *seq/par tree* for the current request object *currReqObj*. It uses the SubProcessIterator submachine to iterate over the sequence in that tree, forwarding the parallel subrequests for each sequential step and ultimately collecting the answers from all subrequests. If all outstanding answers have been received, the answer to the overall request will be compiled and returned to the service requester:

```
Process($currReqObj$) =
  if $status(currReqObj) = started$ then
    SubProcessIterator($currReqObj$)
  if $status(currReqObj) = compileAnswer$ then
    CompileOutAnswMsg for $currReqObj$
    $status(currReqObj) := deliver$
where
  CompileOutAnswMsg for $o$ =
    if $AnswToBeSent(o)$ then
      $SentAnswToMailer(outAnsw2Msg(outAnswer(o))) := true$
  $compileAnswer = yes(FinishedSubReqProcessg)$
```

In a SubProcessIterator step, the immediate subrequests of *currReqObj* will be processed *in order*, as defined by an iterator over that finite set of *sequential subrequests SeqSubReq*(*currReqObj*).

```
SubProcessIterator($currReqObj$) =
  InitializeIterator($currReqObj$) seq
  IterateSubReqProcessg($currReqObj$)
    until $FinishedSubReqProcessg$
where
  $yes(FinishedSubReqProcessg) = compileAnswer$
  $no(FinishedSubReqProcessg) = initStatus($IterateSubReqProcessg$)$
```

The processing of a single (sequential) subrequest *seqSubReq* is defined as follows: For each element *s* of the set of parallel subsubrequest *ParSubReq*(*seqSubReq*) of *seqSubReq*, it starts to FeedSendReq with a request message for *s*, namely by setting *SentReqToMailer*(*outReq2Msg*(*s*)) to *true*. Here, *outReq2Msg*(*s*) transforms the outgoing request into the format for an outgoing request message, which has to be an element of *OutReqMssg*. Since these request messages are processed independently of each other, FeedSendReq elaborates simultaneously for each *s* an *outReqMsg*(*s*). Simultaneously, the machine also Initializes the to be computed *AnswerSet*(*seqSubReq*) before assuming *status* value *waitingForAnswers*, where it remains until *AllAnswersReceived*.

When *AllAnswersReceived*, a submachine ConcludeStep lets the iterative subprocess ProceedToNextSubReq.

While *waitingForAnswers*,i. e. the predicate *AllAnswersReceived* is not yet true, ReceiveAnsw inserts for every *ReceivedAnsw*(*inAnswMsg*) the retrieved internal *answer*(*inAnswMsg*) representation into *AnswerSet*(*seqSubReq*) of the current sequential subrequest *seqSubReq*, which is supposed to be retrievable as *requester* of the incoming answer message.

```
IterateSubReqProcessg =
  if $status(currReqObj) = initStatus($IterateSubReqProcessg$)$ then
    FeedSendReq with $ParSubReq(seqSubReq(currReqObj))$
    Initialize($AnswerSet(seqSubReq(currReqObj))$)
    $status(currReqObj) := waitingForAnswers$
  if $status(currReqObj) = waitingForAnswers$ then
    ConcludeStep
where
  FeedSendReq with $ParSubReq(seqSubReq)$ =
    forall $s \in ParSubReq(seqSubReq)$
      $SentReqToMailer(outReq2Msg(s)) := true$
  ConcludeStep =
    if $AllAnswersReceived$ then
      ProceedToNextSubReq
      $status(currReqObj) := Nxt(status(currReqObj))$
    $Nxt(waitingForAnswers) = testStatus(FinishedSubReqProcessg)$
```

For the definition of ReceiveAnsw we use as parameter the *AnswerSet* function. It provides for every *requester* r, which may have triggered sending some subrequests to subproviders, the *AnswerSet*(r), where to insert (the internal representation of) each *answer* contained in the incoming answer message:

```
ReceiveAnsw($inAnswMsg, AnswerSet$)[1] =
  if $ReceivedAnsw(inAnswMsg)$ then
     insert $answer(inAnswMsg)$ into $AnswerSet(requester(inAnswMsg))$
```

Finally, the two machines that forward requests to provides and return compiled answers back to the service requester are defined symmetrically:

```
SendAnsw($outAnswMsg, SentAnswToMailer$) =
  if $SentAnswToMailer(outAnswMsg)$ then
    Send($outAnswMsg$)
SendReq($outReqMsg, SentReqToMailer$) =
  if $SentReqToMailer(outReqMsg)$ then
    Send($outReqMsg$)
```

[1] Without loss of generality we assume this machine to be preemptive (i. e. *ReceivedAnsw*(*inAnswMsg*) gets false by firing ReceiveAnsw for *inAnswMsg*).

2.3 Complex VP Models

In those situations where the simple execution model of a single VP turns out to be insufficient, one way to construct more complex control structures is through cascading VP instances via composition: This is achieved by connecting the sending and receiving communication interfaces in an appropriate fashion:

- SendReq of VP_i with the ReceiveReq of VP_{i+1}, which implies that in the message passing environment, the types of the sets *OutReqMssg* of VP_i and *InReqMssg* of VP_{i+1} match (via some data mediation).
- SendAnsw of VP_{i+1} with the ReceiveAnsw of VP_i, which implies that in the message passing environment, the types of the sets *OutAnswMssg* of VP_{i+1} and *InAnswMssg* of VP_i match (via some data mediation).

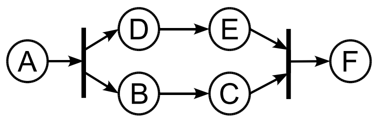

Figure 2: A Complicated Control Flow Example.

For example, the control flow of [Fig. 2] can be realized by a composition of three VPs as indicated in [Fig. 3]: Upon the InitialRequest, VP1 sends a first subrequest for A, followed by sending two parallel requests to subproviders VP2 and VP3 respectively, whose answers in turn trigger VP1 to send the final subrequest for F before eventually providing its FinalAnswer. VP2 and VP3 themselves can be structured sequentially, VP2 handling the request sequence B and C, and VP3 handling the sequence D and E, respectively. Note that these two sequences (and in particular their termination behavior) are independent of each other. Without the possibility to compose our VPs, the only way to realize such a structured control flow would be via programming the corresponding internal VP behavior.

Furthermore, one could extend the VP model by replacing the simple communication patterns with more complex ones. ReceiveReq and SendAnsw are identified in [Barros and Börger 2005] as basic bilateral service interaction patterns, namely as mono-agent ASM modules Receive and Send. The FeedSendReq submachine together with SendReq in Process realise an instance of the basic multilateral mono-agent service interaction pattern called OneToManySend in [Barros and Börger 2005], whereas the execution of ReceiveAnsw in IterateSubReqProcessg until *AllAnswersReceived* is an instance of the basic multilateral mono-agent OneFromManyReceive pattern from [Barros and Börger 2005]. One can refine VP to concrete business process

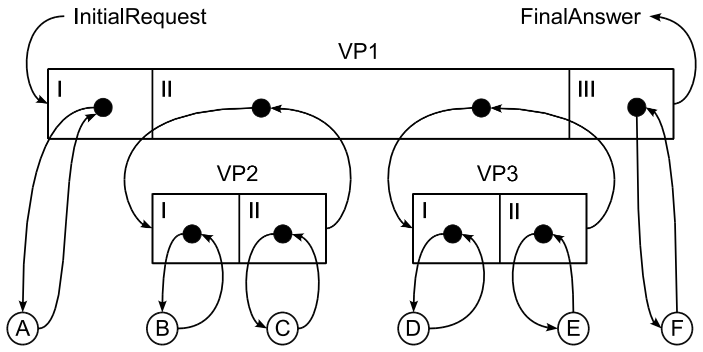

Figure 3: Virtual Provider Composition.

applications by enriching the communication flow structure built from basic service interaction patterns as analyzed in [Barros and Börger 2005].

3 Service Discovery

It actually turned out that the underlying abstract processing model of the VP could be easily extended to serve different application scenarios. For instance, in the context of the DIP project we had the task to provide a specification for a distributed semantic web service (SWS) discovery framework for Web services modeled in Web Service Modeling Ontology (WSMO). The term *semantic* is used here to indicate that the service discovery should not be based on syntactical, e. g., keyword based search, but rather on matching semantically described *goals* (e. g., WSMO goals) with semantic annotations of web services (e. g., WSMO capability descriptions). The overall solution is specified as a distributed architecture, where a *service discovery provider* will either distribute goal queries to peer entities or will forward a query to a local *discovery engine*. For both entities in this architecture, the specifications could be directly derived from the VP model by fairly simple refinements. The main reason for distributing semantic discovery services is the computationally very expensive semantic matchmaking procedure of discovery goals and web service capability descriptions.

Notational changes: Since the SWS discovery framework deals with discovery *goals* instead of *requests*, all artifacts in the VP model that relate to requests have been renamed appropriately; e. g. *ReceiveReq* has been turned into *ReceiveGoal*, *SendAnsw* turned into *SendSetOfWS*, *Process* turned into *ProcessGoal*, *currReqObj* turned into *currReqGoal*, etc. Hence, at the highest level of abstraction the SWS Discovery Service Provider has been derived from the original VP by applying a simple renaming procedure:

MODULE DiscoveryServiceProvider =

```
choose $M \in \{\textsc{ReceiveGoal}, \textsc{SendSetOfWS}, \textsc{ProcessGoal}, \textsc{SendGoal}\} \cup$
$\{\textsc{ReceiveSetOfWS}\}$
  $M$
```

In our SWS discovery framework the same goal query is distributed without any changes to different locations (service discovery providers). The service discovery providers reply with sets of discovered web service descriptions which only need to be subsequently aggregated into a global set of discovered web service descriptions representing the total answer to the original discovery query. This leads to a *simplified control sequence*, i. e., discovery goal queries are simply forwarded to peer entities. Thus, the sequential part of the *seq/par tree* is always of length 1, making the subprocess iteration step simpler, see the upper part of [Fig. 4]. Finally, query forwarding requires some *refinements* of the VP model in order to deal with exceptions (most notably loop detections and time outs), see the gray shaded part of [Fig. 4].

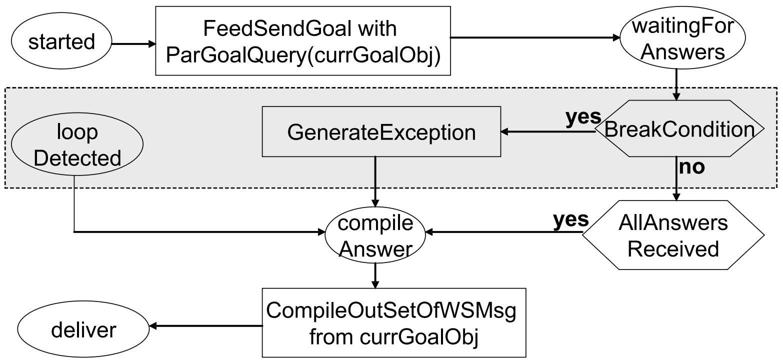

Figure 4: Refinement of Process(Goal) ASM.

In detail, the definition in [Fig. 4] expresses that each call of ProcessGoal for a *started* goal object *currGoalObj* triggers to FeedSendGoal with a goal query to be sent out for every relevant discovery location l of the current *currGoalObj*. Those immediate goal queries, elements of a set *ParGoalQuery(currGoalObj)*, are assumed to be processable by other discovery service providers independently of each other, FeedSendGoal elaborates simultaneously for each l an *outGoalMsg(l)*. Simultaneously, ProcessGoal updates the status of *currGoalObj* to *status(currGoalObj) := waitingForAnswers*, where *currGoalObj* remains until *AllAnswersReceived* or *BreakCondition* becomes *true*. ProcessGoal call for a *loopDetected* updates the status of *currGoalObj* to *status(currGoalObj) := compileAnswer*. As long as during *waitingForAnswers*, *AllAnswersReceived(currGoalObj)* is not yet *true* and *Break-*

Condition(currGoalObj) is *false*, ReceiveSetOfWS inserts for every *ReceivedSetOfWS(inSetOfWSMsg)* the retrieved internal *setOfWS(inSetOfWSMsg)* representation into *SetOfWS(currGoalObj)* of the currently processed goal *currGoalObj*, which is supposed to be retrievable as goal of the incoming answer message. Once ProcessGoal finds *BreakCondition(currGoalObj)* or *AllAnswersReceived(currGoalObj)* becomes *true* then *currGoalObj* assumes status value *compileAnswer*. Once, ProcessGoal finds *status(currGoalObj)* has value *compileAnswer* it compiles from *currGoalObj* (which allows to access *SetOfWS(currGoalObj))* an answer of the current discovery location and updates the status of *currGoalObj* to *status(currGoalObj) := deliver*. For the sake of comparison with the original version of the Process machine we also provide here the textual definition of the ProcessGoal.

```
ProcessGoal(currGoalObj) =
  if status(currGoalObj) = started then
    FeedSendGoal with ParGoalQuery(currGoalObj)
    status(currGoalObj) := waitingForAnswers
  if status(currGoalObj) = loopDetected then
    status(currGoalObj) := compileAnswer
  if status(currGoalObj) = waitingForAnswers then
    if BreakCondition(currGoalObj) then
      GenerateException(currGoalObj)
      status(currGoalObj) := compileAnswer
    else
      if AllAnswersReceived(currReqObj) then
        status(currGoalObj) := compileAnswer
  if status(currGoalObj) = compileAnswer then
    CompileOutSetOfWSMsg from currGoalObj
    status(currGoalObj) := deliver
```

Exhaustive motivation for the proposed distributed semantic service discovery framework and full treatment of the model refinements is described in [Altenhofen et al. 2006].

4 Service Composition

Our work on protocol mediation with the VP has eventually led to a new approach for *service composition*. We interpret multi-party communications as an *orchestration problem*, i. e., finding a mediator that could steer the interactions among those parties, where one party is considered the initiator of the conversation defining the *required interface*. For this, we have developed a mathematical model of (Web) services based on ASMs and a composition algorithm that is able to construct an *transactional* execution plan that guarantees that any possible execution trace in that plan will reach an expected end state for all parties, i. e., either a success or recovery state [see Lemcke and Friesen 2007]. Since all artifacts in that model are specified as ASMs, we can actually simulate the execution plan in the CoreASM execution engine [see Farahbod et al. 2005].

4.1 Enriched VP Model

In order to create the implementation of a VP ensuring transactional behavior, we need to use a more detailed representation of requester, providers and orchestration which we give in this section. In addition, we line out how the extended definitions correlate with the original VP ASMs defined in [Section 2.2]. Corresponding ASMs carry the same name as the original with an asterisk attached (*).

4.1.1 Handling of Requester and Providers

For the sake of generating the VP implementation, we treat the required interface defined by the initiator of a conversation as a special Web service interface among the interfaces of all provider Web services. This fact is represented in [Fig. 5] by just referring to participant of the orchestration as Web services rather than requester and providers.

It is important to differentiate two possible views on the required interface.

1. The outside world later invoking the VP sees the requester like a regular, passive Web service that can be triggered by sending an initial message. Thus, its first action from the *outside* point of view must involve some receiving activity.
2. The providers see the requester as the only proactive Web service among themselves that later on triggers the execution of the VP. Thus, its first action from the *inside* point of view must involve some sending activity.

In all diagrams showing the participants of an orchestration, and for generating the implementation of the VP, we use the inside view. Between both the inside and the outside view can easily be converted by swapping the direction of all communications.[1]

4.1.2 Web Services

For us, a Web service is defined via a set of possible input and output messages, referred to as input and output variables (IN and OUT), a set of states (S), and a state transition function (ST). Please note that the Web service definition includes business process information by the state transition function. We currently restrict ST to a bipartite, directed tree. Bipartite means that input and output transitions alternate. Multiple input transitions leaving the same state model a user-determined decision. Multiple output transitions model service behavior out of the user's sphere of influence. We refer to this as non-deterministic behavior. [Fig. 6] shows four exemplary Web services.

[1] That means, outputs become inputs, and inputs become outputs.

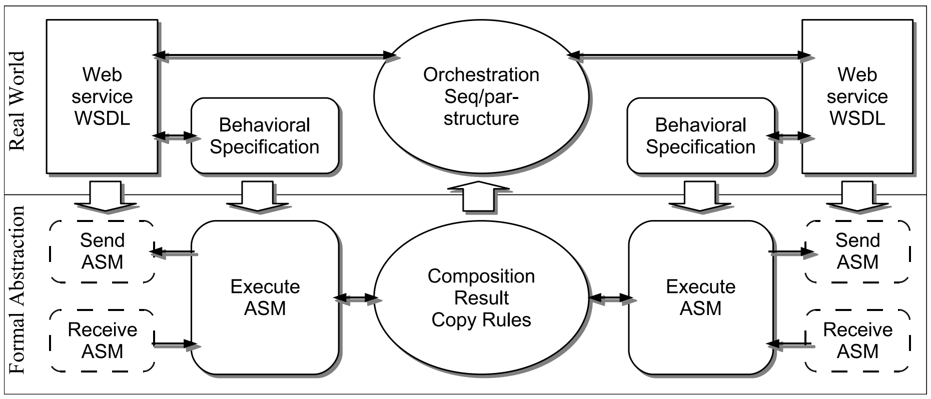

Figure 5: Two views on the interaction model.

$$
\begin{aligned}
\text{WebService} &:= \langle \text{IN}, \text{OUT}, \text{S}, s_{\text{init}}, \text{ST} \rangle \\
\text{IN}, \text{OUT}, \text{S} &\ldots \text{ sets of input and output variables and states} \\
s_{\text{init}} &\in \text{S} \\
\text{ST} &= \text{ST}_{\text{in}} \cup \text{ST}_{\text{out}} \\
\text{ST}_{\text{in}} &: \text{S} \times (2^{\text{IN}} \setminus \{\emptyset\}) \rightarrow \text{S} \\
\text{ST}_{\text{out}} &: \text{S} \times (2^{\text{OUT}} \setminus \{\emptyset\}) \rightarrow \text{S}
\end{aligned}
$$

We refuse business logic encoded into Web service behavior. Messages whose content influences subsequent Web service behavior must be classified into different variables. We are hence only interested in a variable's status rather than its value and introduce the varState function.

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

We now define the execution of Web services. As denoted in [Fig. 5], a Web service interface is abstracted by the ASMs SendReq* and ReceiveAnsw*. The ASMs Send* and Receive* contained are hooks to access the underlying Web service implementation.

```
SendReq*($\mathit{wsId}$) ≡
  **do forall** $\{ (I, s_{\text{post}}) : \ (s_{\text{pre}}, I, s_{\text{post}}) \in \text{ST}_{\text{in}}^{\mathit{wsId}} \}$
    **if** wsState($\mathit{wsId}$) $= s_{\text{post}}$
        **and** varState($\mathit{wsId}, i_1$) = initialized **and** ...
        **and** varState($\mathit{wsId}, i_{|I|}$) = initialized **then**
      Send*($\mathit{wsId}, I$)
      **forall** $i \in I$ **do** varState($\mathit{wsId}, i$) := processed
**where**
  $i_x \in I, \ x = 1 \,..\, |I|$
```

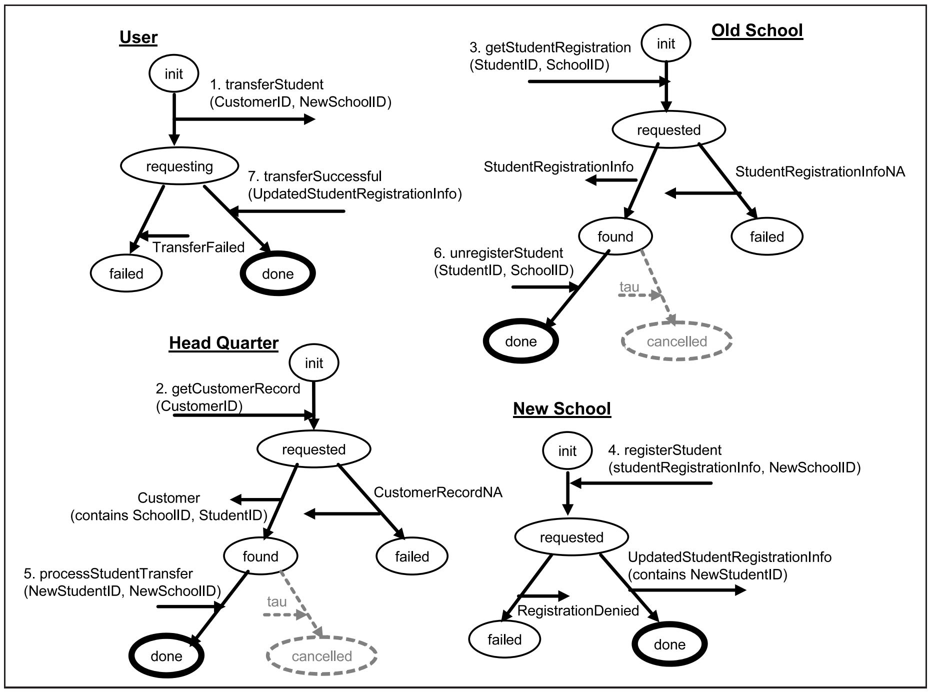
User
init
1. transferStudent (CustomerID, NewSchoolID)
requesting
7. transferSuccessful (UpdatedStudentRegistrationInfo)
TransferFailed
failed
done
Old School
3. getStudentRegistration (StudentID, SchoolID)
init
requested
StudentRegistrationInfo
StudentRegistrationInfoNA
found
failed
6. unregisterStudent (StudentID, SchoolID)
tau
done
cancelled
Head Quarter
init
2. getCustomerRecord (CustomerID)
requested
Customer (contains SchoolID, StudentID)
CustomerRecordNA
found
failed
5. processStudentTransfer (NewStudentID, NewSchoolID)
tau
done
cancelled
New School
init
4. registerStudent (studentRegistrationInfo, NewSchoolID)
requested
UpdatedStudentRegistrationInfo (contains NewStudentID)
RegistrationDenied
failed
done

Figure 6: Example 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. The "tau"-transitions are virtual transitions ("syntactic sugar"). They translate as follows. The pre-state is a final, unsuccessful state. The transition and its posterior state do not exist.

SendReq* differs from SendReq in that it potentially sends multiple variables that belong to the same message, and it checks for the correct current state of the corresponding Web service before sending the message. The check *SentAnswToMailer* is implemented by requiring every variable's state to be initialized.

```
$\textsc{ReceiveAnsw}^*(\mathit{wsId}) \equiv$
  do forall $\{\, (s_{\mathrm{pre}}, O) : \ (s_{\mathrm{pre}},\ O,\ s_{\mathrm{post}}) \in \mathrm{ST}_{\mathrm{out}}^{\mathit{wsId}} \,\}$
    if $\mathrm{wsState}(\mathit{wsid}) = s_{\mathrm{pre}}$ and $\textsc{Receive}^*(\mathit{wsId}, O)$ then
      forall $o \in O$ do $\mathrm{varState}(\mathit{wsId}, o) :=$ initialized
where
  $o_x \in O,\ x = 1 \,..\, |\, O \,|$
```

ReceiveAnsw* again differs from ReceiveAnsw in that it checks for the correct Web service state and maintains individual variable states. The corresponding concept to the notion of the *AnswerSet* used in [Section 2.2] is the state space of Web services and variables that is maintained and accessed from all Web service models in the scope of one orchestration. This can also be seen in the UpdateBelief ASM which defines the actual behavior of a Web service based on its state transitions.

```
UpdateBelief($wsId$) $\equiv$ do forall $(s_{\mathrm{pre}},\ V,\ s_{\mathrm{post}}) \in \mathrm{ST}^{wsId}$
  if wsState($wsid$) $= s_{\mathrm{pre}}$
      and varState($wsId, v_1$) = initialized and ...
      and varState($wsId, v_{|V|}$) = initialized then
    wsState($wsId$) := $s_{\mathrm{post}}$
where
  $v_x \in V,\ x = 1\,..\,|V|$
```

The UpdateBelief machine does not have a direct equivalent in the original VP ASMs, because the original definition did not talk about states of Web services. Since we have to update the VP's belief of the progress of the Web services in the system, the UpdateBelief machine has to be invoked at appropriate times during the execution of the Process machine.

4.1.3 Communication

In the previous section, we have introduced a mathematical definition of Web services. In this section, we show how we represent communication between different Web services. Since we now have to refer to Web services and their variables in a global manner, we need to add a unique identification to each such statement. We thus introduce two sets that uniquely identify states and variables of specific Web services.

$$\begin{aligned} \mathrm{WSState} &:= \{\, wsId \mapsto wsState :\ wsId \in \mathrm{ID}, wsState \in \mathrm{S}^{wsId} \,\} \\ \mathrm{Variable} &:= \{\, (wsId, v) :\ wsId \in \mathrm{ID}, v \in (\mathrm{IN}^{wsId} \cup \mathrm{OUT}^{wsId}) \,\} \end{aligned}$$

Correspondingly, we define a possible communication as a tuple of a globally unique output variable of one Web service and a globally unique input variable of another Web service.

$$\begin{aligned} \mathrm{VarAss} := \{\, &((wsId_1, o), (wsId_2, i)) :\ wsId_1, wsId_2 \in \mathrm{ID}, \\ &wsId_1 \neq wsId_2, o \in \mathrm{OUT}^{wsId_1}, i \in \mathrm{IN}^{wsId_2} \,\} \end{aligned}$$

Based on the Web service state and variable assignment definitions, we now define a concrete communication as a copy rule: $\mathrm{CopyRule} := \langle S, A \rangle,\ S \subseteq \mathrm{WSState},\ A \subseteq \mathrm{VarAss}$. 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 [Fig. 6].

$$\begin{aligned} c_{ps_{10}} = \ & (\{ (\mathrm{U}, \mathrm{requesting}), (\mathrm{O}, \mathrm{init}), (\mathrm{H}, \mathrm{init}), (\mathrm{N}, \mathrm{init}) \}, \\ & \{ ((\mathrm{U}, \mathrm{CustID}), (\mathrm{H}, \mathrm{CustID})) \}) \end{aligned}$$

As denoted, the execution of a copy rule corresponds to changing the states of the variables involved. Since a copy rule contains multiple assignments of variables, it may triggers the further sending of multiple messages via the respective SendReq* ASMs. The effect of this behavior strongly corresponds with the functioning of the SubProcessIterator machine defined in [Section 2.2].

```
$\textsc{SubProcessIterator}^*(\mathit{rules} \subseteq \mathrm{CopyRule}) \equiv$ do forall $(\mathit{states}, \mathit{varAss}) \in \mathit{rules}$
  if $\mathrm{wsState}(\mathit{wsId}_1) = s_1$ and ...
      and $\mathrm{wsState}(\mathit{wsId}_{|\,\mathit{states}\,|}) = s_{|\,\mathit{states}\,|}$
      and $\mathrm{varState}(\mathit{wsId}_{\mathrm{out}_1}, o_1) = \mathrm{initialized}$ and ...
      and $\mathrm{varState}(\mathit{wsId}_{\mathrm{out}_{|\,\mathit{varAss}\,|}}, o_{|\,\mathit{varAss}\,|}) = \mathrm{initialized}$
      and $\mathrm{varState}(\mathit{wsId}_{\mathrm{in}_1}, i_1) = \mathrm{undef}$ and ...
      and $\mathrm{varState}(\mathit{wsId}_{\mathrm{in}_{|\,\mathit{varAss}\,|}}, i_{|\,\mathit{varAss}\,|}) = \mathrm{undef}$ then
    do forall $((\mathit{wsId}_{\mathrm{out}}, o), (\mathit{wsId}_{\mathrm{in}}, i)) \in \mathit{varAss}$
      $\mathrm{varState}(\mathit{wsId}_{\mathrm{in}}, i) := \mathrm{initialized}$
where
  $(\mathit{wsId}_k, s_k) \in \mathit{states}, \quad k = 1 \,..\, |\,\mathit{states}\,|$
  $((\mathit{wsId}_{\mathrm{out}_n}, o_n), (\mathit{wsId}_{\mathrm{in}_n}, i_n)), \quad n = 1 \,..\, |\,\mathit{varAss}\,|$
```

4.1.4 Orchestration

In the former sections, we have presented mathematical models for Web services and their communication. The alternating triggering of the SendReq* and ReceiveAnsw* abstract state machines is the orchestration of the respective set of Web services and thus the implementation of the Virtual Provider. This functionality corresponds to the Process machine of [Section 2.2].

```
$\textsc{Process}^*(cg \in \mathrm{CompGoal}, A \subseteq \mathrm{VarAss}) \equiv$
  choose $\{ M : \ M \equiv \textsc{UpdateBelief}(\mathit{wsId}) \ \vee$
      $M \equiv \textsc{SubProcessIterator}^*(\mathit{rules}), \mathit{wsId} \in \mathrm{ids}(cg),$
      $\mathit{rules} = \textsc{ReachCompGoal}(cg, A) \}$
    $M$
```

Through the way we defined the copy rules and their execution, we ensure that the behavior of Process* resembles the processing of the seq/par-structure introduced in [Section 2.2]. Firstly, a copy rule contains a set of variable assignments that may trigger multiple Web services. Thus, copy rules implement parallel invocation of Web services. Secondly, only after all Web services have responded, the next copy rule may fire. This is ensured via belief over the states of the variables and Web services. Thus, copy rules implement sequencing of the parallel Web service invocations. The only missing piece in the definition above is the actual creation of the copy rules. This will be done by the machine ReachCompGoal which will be explained in the following section.

For completeness, we reformulate the VP module using the enriched VP ASMs from this section. Please note that the ASMs ReceiveReq and SendAnsw are missing in the new module definition below. The reason is that we handle requestors the same way as providers. Thus, the functionalities of ReceiveReq and SendAnsw are covered by ReceiveAnsw* and SendReq*, respectively.

```
MODULE $\textsc{VirtualProvider}^*(cg \in \text{CompGoal}, A \subseteq \text{VarAss}) \equiv$
  choose $\{\, M :\ M \equiv \textsc{Process}^*(cg, A) \ \vee$
        $M \equiv \textsc{SendReq}^*(wsId) \ \vee\ M \equiv \textsc{ReceiveAnsw}^*(wsId), wsId \in \text{ids}(cg) \,\}$
    $M$
```

4.2 VP Generator

[Fig. 7] shows the architecture of the VP generator. In the environment of a service-oriented architecture, the inputs to the VP generator would be WSDL files, an annotation of their states as well as the allowed variable assignments. The transform blocks in the picture convert the inputs to the internal representations described in the former section. We do not detail the real-world representation of variable assignments, because there is no standard for this to our knowledge. In the following, we walk through the remaining components of our composition system.

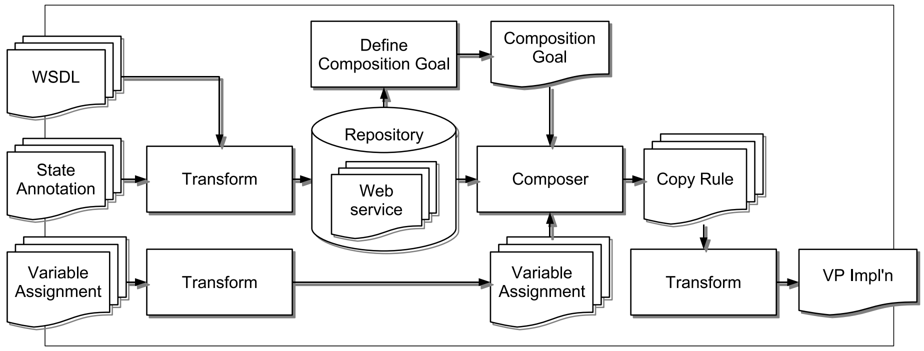

Figure 7: VP Generator Architecture.

4.2.1 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: $\text{Repository} := 2^{\text{ID}\times\text{WebService}}$. A repository should only include a single requester. A requester is a Web service whose first action is sending an output message. [Fig. 6] shows a set of Web services that make up an exemplary repository. The User Web service acts as requester.

4.2.2 Define Composition Goal

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 (PrimGoal) and recovery goals (RecGoal). Both types of goals are used to describe the requirements of a correct orchestration (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. For our student transfer example it would be bad if the Old School successfully unregistered a student, but the New School failed in registering the student.

$$\begin{aligned}
\text{CompGoal} &:= \langle \text{PrimGoal}, \text{RecGoal} \rangle \\
\text{PrimGoal} &\subseteq \text{Goal} \quad \ldots \text{ set of primary goals} \\
\text{RecGoal} &\subseteq \text{Goal} \quad \ldots \text{ set of recovery goals} \\
\text{Goal} &:= 2^{\{\, wsId \mapsto wsState:\ wsId \in \text{ID}, wsState \in \text{S}^{wsId} \,\}}
\end{aligned}$$

We illustrate the goal definition by giving possible primary and recovery goals for the Web services of our example repository in [Tab. 1].

4.2.3 Composer

This section describes the composition algorithm in detail by a set of ASMs. Each ASM represents a module of the algorithm. The interaction of all modules is depicted in [Fig. 8]. The following sections concentrate on the dividing part, the correctness part, and the main module of the core composition part of [Fig. 8]. [Section 4.2.3.1] reduces the problem of composing complex Web services to the composition of smaller units. [Section 4.2.3.2] explains how our algorithm ensures a correct composition by properly ordering the composition of the smaller units.

ID	User	OldSchool	HeadQuarter	NewSchool
pg_1	done	done	done	done
rg_8	failed	init	failed	init
rg_{10}	failed	failed	failed	init
rg_{12}	failed	cancelled	failed	init
rg_{16}	failed	failed	cancelled	init
rg_{26}	failed	init	failed	failed
rg_{28}	failed	failed	failed	failed
rg_{30}	failed	cancelled	failed	failed
rg_{34}	failed	failed	cancelled	failed
rg_{36}	failed	cancelled	cancelled	failed

Table 1: Exemplary goals.

The smaller units are then composed by the core of our composition algorithm. [Section 4.2.3.3] gives a high-level overview of the core composition algorithm.

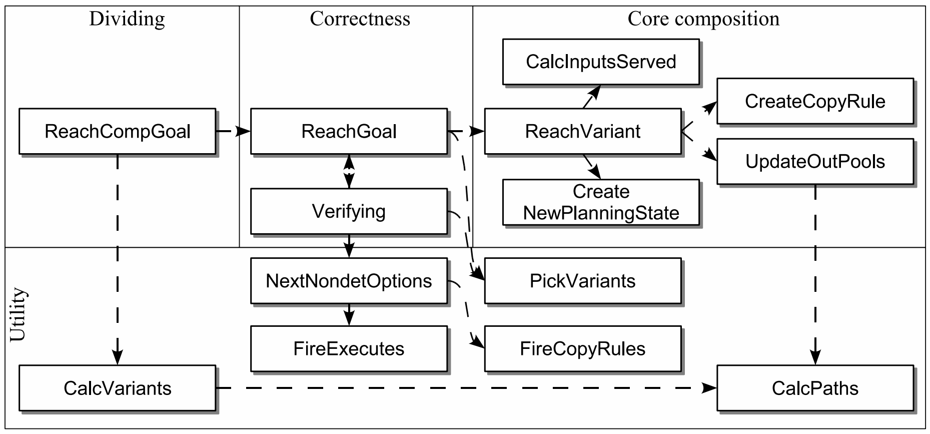

Figure 8: ASM Modules of the composer.

4.2.3.1 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 $:= 2^{\{\, wsId \mapsto Transs:\ wsId \in \mathrm{ID}, Transs \subseteq \mathrm{ST}^{wsId}\,\}}$.

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 to ReachGoal. It also ensures that every possible execution of the resulting copy rules ends in one of the composition goals (CompGoal). 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 a *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*).

4.2.3.2 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.

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.

4.2.3.3 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 a *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.

Further detail can be found in [Lemcke and Friesen 2007]. We rather stick with this high-level description and provide an example in the next section.

4.2.4 Example

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.

$$\begin{aligned} c_{pg_1} = \; & (\,\{\,(\mathrm{U}, \mathrm{requesting}), (\mathrm{O}, \mathrm{found}), (\mathrm{H}, \mathrm{found}), (\mathrm{N}, \mathrm{done})\,\},\\ & \;\{\,((\mathrm{N}, \mathrm{UpdRegInfo}), (\mathrm{U}, \mathrm{UpdRegInfo})), ((\mathrm{H}, \mathrm{Customer}), (\mathrm{O}, \mathrm{StudID})),\\ & \quad ((\mathrm{H}, \mathrm{Customer}), (\mathrm{O}, \mathrm{SchoolID})), ((\mathrm{N}, \mathrm{NewStudID}), (\mathrm{H}, \mathrm{NewStudID})),\\ & \quad ((\mathrm{U}, \mathrm{NewSchoolID}), (\mathrm{H}, \mathrm{NewSchoolID}))\,\}); \end{aligned}$$

$$\begin{aligned}
c_{ps_8} = {} & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{found}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{init})\,\},\\
& \{\,((\mathrm{O},\mathrm{StudRegInfo}),(\mathrm{N},\mathrm{StudRegInfo})),\\
& \quad ((\mathrm{U},\mathrm{NewSchoolID}),(\mathrm{N},\mathrm{NewSchoolID}))\,\});\\
c_{ps_9} = {} & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{init}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{init})\,\},\\
& \{\,((\mathrm{H},\mathrm{Customer}),(\mathrm{O},\mathrm{StudID})),((\mathrm{H},\mathrm{Customer}),(\mathrm{O},\mathrm{SchoolID}))\,\});\\
c_{ps_{10}} = {} & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{init}),(\mathrm{H},\mathrm{init}),(\mathrm{N},\mathrm{init})\,\},\\
& \{\,((\mathrm{U},\mathrm{CustID}),(\mathrm{H},\mathrm{CustID}))\,\})
\end{aligned}$$

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 $\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{init}),(\mathrm{H},\mathrm{failed}),(\mathrm{N},\mathrm{init})\,\}$. Subsequently, the reachable, allowed goals are rg_8, rg_{10}, rg_{12}, rg_{26}, rg_{28} and rg_{30}.

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

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 list the copy rules below.

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

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 option we find is $\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{failed}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{init})\,\}$. The allowed, reachable goals are rg_{16} and rg_{34}.

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

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

$$\begin{aligned}
c_{rg_{16}} = {} & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{failed}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{init})\,\},\\
& \{\,((\mathrm{O},\mathrm{Fail}),(\mathrm{U},\mathrm{Fail}))\,\});\\
c_{ps_{14}} = {} & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{init}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{init})\,\},\\
& \{\,((\mathrm{H},\mathrm{Customer}),(\mathrm{O},\mathrm{SchoolID})),((\mathrm{H},\mathrm{Customer}),(\mathrm{O},\mathrm{StudID}))\,\});
\end{aligned}$$

$$\begin{aligned} c_{ps_{15}} = \; & (\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{init}),(\mathrm{H},\mathrm{init}),(\mathrm{N},\mathrm{init})\,\},\\ & \;\{\,((\mathrm{U},\mathrm{CustID}),(\mathrm{H},\mathrm{CustID}))\,\}) \end{aligned}$$

Simulating the copy rules above yields no more non-determinism. Thus, we continue the simulation of the original copy rules and find the last non-deterministic option $(\{\,(\mathrm{U},\mathrm{requesting}),(\mathrm{O},\mathrm{found}),(\mathrm{H},\mathrm{found}),(\mathrm{N},\mathrm{failed})\,\})$. The only reachable, allowed goal is rg_{36}. We give the copy rules resulting from its composition below. The copy rules for this option do not contain any new non-determinism.

$$allowedGoal_9 = rg_{36} \;=\; \{\,(\mathrm{U},\mathrm{failed}),(\mathrm{O},\mathrm{cancelled}),(\mathrm{H},\mathrm{cancelled}),(\mathrm{N},\mathrm{failed})\,\}$$

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

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.

5 Related Work

The innovation of our approach lies in providing a coherent framework to specify service mediation, service discovery, and service composition. The idea to use the ASM modeling framework for providing a high-level specification of service mediators has been stimulated by work carried out in the EU-funded research project DIP.[1] There, ASMs had been used to describe both *WSMO choreographies* and *WSMO orchestrations*, but the technical report [Scicluna et al. 2006] actually did not specify how such a choreography would be linked up to a corresponding orchestration other than by sharing some states. Although perfectly legal from

[1] See `http://dip.semanticweb.org`

a conceptual point of view, we felt that this coupling was actually *too loose* and that a more rigid model would be needed, especially when it comes to (formally) proving system properties. Further investigation of the problem revealed that we were actually looking for a model that could be used to describe *service behavior mediators*, i. e., mediation on the message flow level.

Regarding the distributed model for semantic service discovery, we can now find publications that describe a similar architectural approach [see Arabshian and Schulzrinne 2006, Bianchini et al. 2008], although none of them provides a formal execution model.

It turns out that service mediation and service composition can be seen as two perspectives on the same task. The task to map a requested interface with multiple provided interfaces is called service mediation. The task to compile an orchestration of a set of provided interfaces with certain properties that represent the added value of the generated application is called service composition. In our case, following a specific requested interface is one of those properties.

In this respect, approaches to service mediation and service composition are related to our service mediation formalization and VP generator. The type of service mediation and composition we pursue most appropriately falls under the category of workflow composition as we observe complex behavioral requirements of the provided and requested interfaces. Related approaches include work performed using pi-calculus, e.g. [Puhlmann 2006], others use merged finite state machines [see Küster et al. 2007] or similar representations. These approaches on the one hand concentrate on merging workflows but have on the other hand no facility for exception management or complex goal definition. Consequently, our approach is most related to [Pistore et al. 2005] which uses model checking on a finite state representation of joined abstract BPEL models. The drawback of this solution is that its computation is in the range of seconds to minutes due to the use of general purpose tools. Our approach in contrast does not compute a joined finite state machine, but operates on the ASM specification directly. Thus, we can reach better performance in the range of fractions or a few seconds by avoiding the state space explosion.

We would like to note that process composition as performed utilizing AI planning mostly assumes atomic services—i. e. without complex behavioral requirement [see Rao and Su 2004]. The work carried out in schema matching [see Shvaiko and Euzenat 2005] is somewhat orthogonal, yet complementary, to our approach as it is concerned with relating data formats of every two participants to each other. We abstract from this in two ways. (1) When presenting our formal model of service mediation, the VPs' implementation may indeed contain some code mapping data structures to one another. Our formalization rather focuses on the sec/par structure of the communication protocol. (2) When talking about service composition, we have to assign matching input to output variables as one of the prerequisites to start the automatic composer. This however is not concerned with the actual data structure or values, but rather concentrates on the fact that data from some variable A has to be assigned to some variable B on the *conceptual level.* The actual copying of the run-time values from A to B on the *technical level* may indeed include a complex value mapping which is out of scope of our work, but addressed, e. g., in [Drumm et al. 2007].

6 Conclusion

In this paper, we have presented our work on formal models for protocol mediation, service discovery, and service composition generating a transactional execution plan.

Each of our technologies individually extends the original service-oriented architecture with features important to real-world applications. (a) Our formal model for mediation allows to theoretically prove that a requested interface can be implemented by the provided interfaces [see Altenhofen et al. 2006]. (b) Our approach for service discovery allows to distribute the discovery transparently to the traditional discovery request and provisioning interfaces. This is done by implementing the traditional discovery request interface via a VP that distributes the request to multiple instances of the traditional discovery provisioning interface. We thus seamlessly integrate with existing discovery infrastructure. (c) Our VP generator is capable of creating an orchestration that links requested and provided interfaces based on user-defined transactional requirements.

The combination of our technologies has the potential to implement complicated, desirable features of service-oriented architectures. (a) With the subsequent invocation of discovery and composition, we can prove whether there is an orchestration for a requested interface based on the discovered services. For this, we would have to automate the goal definition by upfront naming successful and unsuccessful states in our Web service specifications that then would be used to generate potential primary and recovery goals. (b) If we interlink the composition with discovery the other way around, we are able to implement dynamic discovery of provider services at the runtime of our VP, as this is proposed in some semantic Web use cases.

In addition to the integration of our techniques as described, future work on the individual components may include the following. (a) Our VP generator allows to define variable assignments that involve complex message transformations to be considered during the composition process. In a real implementation of our approach, generic mediators performing those transformations should be known upfront and incorporated into the seq/par-structure the VP generator creates. This allows for an extreme flexibility in the services that can be orchestrated to fulfill a requested interface. (b) In the case that no orchestration could be generated out of the given provided interfaces for the given requested interface, we could extend the VP generator to propose missing mediator interfaces necessary to complete the composition. The mediation interfaces would in a second step have to be implemented by a human programmer.

References

[Altenhofen et al., 2006] Altenhofen, M. ; Börger, E. ; Friesen, A. ; Lemcke, J.: A high-level specification for virtual providers. In: *International Journal of Business Process Integration and Management. Inderscience Publishers* Special Issue 2006 (2006)

[Altenhofen et al., 2005] Altenhofen, M. ; Börger, E. ; Lemcke, J.: An Execution Semantics for Mediation Patterns. In: *Proc. of 2nd WSMO Implementation Workshop WIW'2005*. Innsbruck, Austria : CEUR Workshop Proceedings, June 6-7 2005. – URL `CEUR-WS.org/Vol-134/lemcke-wiw05.pdf`. – ISSN 1613-0073, online CEUR-WS.org/Vol-134/lemcke-wiw05.pdf. – ISSN 1613-0073

[Andrews et al., 2003] Andrews, T. ; F., Curbera ; H., Dholakia ; Goland, Y. ; Klein, J. ; F., Leymann ; Liu, K. ; Roller, D. ; Smith, D. ; Thatte, S. ; Trickovic, I. ; Weerawarana, S.: Business Process Execution Language for Web Services (BPEL4WS). Specification Version 1.1, 5 May 2003. URL `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`, 2003. – Technical Report

[Arabshian and Schulzrinne, 2006] Arabshian, K. ; Schulzrinne, H.: An Ontology-based Hierarchical Peer-to-Peer Global Service Discovery System. In: *Journal of Ubiquitous Computing and Intelligence (JUCI)* (2006)

[Barros and Börger, 2005] Barros, A. ; Börger, E.: A Compositional Framework for Service Interaction Patterns and Communication Flows. In: Lau, K.-K. (Eds.) ; Banach, R. (Eds.): *Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)* Ed. 3785, Springer, 2005, pp. 5–35

[Bianchini et al., 2008] Bianchini, D. ; De Antonellis, V. ; Melchiori, M.: Flexible Semantic-based Service Matchmaking and Discovery. In: *World Wide Web Journal* (2008). – URL `http://www.springerlink.com/content/4514473142836174/`

[Börger and Stärk, 2003] Börger, E. ; Stärk, R. F.: *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003

[Drumm et al., 2007] Drumm, Christian ; Lemcke, Jens ; Oberle, Daniel: Business Process Management And Semantic Technologies. In: Cardoso, Jorge (Eds.) ; Hepp, Martin (Eds.) ; Lytras, Miltiadis D. (Eds.): *The Semantic Web: Real-World Applications from Industry* Ed. 6. Springer, 2007, pp. 209–239. – ISBN 978-0-387-48531-7

[Farahbod et al., 2005] Farahbod, R. ; Gervasi, V. ; Glässer, U.: CoreASM: An extensible ASM execution engine. In: *Proc. of the 12th Intl Workshop on Abstract State Machines. Paris, France*, 2005

[Friesen and Börger, 2006] Friesen, A. ; Börger, E.: A high-level specification for Semantic Web Service Discovery Services. In: *ICWE '06: Workshop proceedings of the sixth international conference on Web engineering*, 2006

[Hohpe and Woolf, 2003] Hohpe, G. ; Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321200683

[Küster et al., 2007] Küster, Jochen M. ; Ryndina, Ksenia ; Gall, Harald: Generation of Business Process Models for Object Life Cycle Compliance. In: Alonso, Gustavo (Eds.) ; Dadam, Peter (Eds.) ; Rosemann, Michael (Eds.): *BPM* Ed. 4714, Springer, 2007, pp. 165–181. – ISBN 978-3-540-75182-3

[Lemcke and Friesen, 2007] Lemcke, J. ; Friesen, A.: Composing Web-service-like abstract state machines (ASMs). In: *Workshop on Web Service Composition and Adaptation (WSCA07). 2007 IEEE International Conference on Web Services, ICWS 2007.*, 2007

[Pistore et al., 2005] Pistore, Marco ; Traverso, Paolo ; Bertoli, Piergiorgio ; Marconi, Annapaola: Automated Synthesis of Composite BPEL4WS Web Services. In: *ICWS*, IEEE Computer Society, 2005, pp. 293–301. – ISBN 0-7695-2409-5

[Puhlmann, 2006] Puhlmann, Frank: Why Do We Actually Need the Pi-Calculus for Business Process Management? In: Abramowicz, Witold (Eds.) ; Mayr, Heinrich C. (Eds.): *BIS* Ed. 85, GI, 2006, pp. 77–89. – ISBN 3-88579-179-X

[Rao and Su, 2004] Rao, Jinghai ; Su, Xiaomeng: A Survey of Automated Web Service Composition Methods. In: Cardoso, Jorge (Eds.) ; Sheth, Amit P. (Eds.): *SWSWPC* Ed. 3387, Springer, 2004, pp. 43–54. – ISBN 3-540-24328-3

[Scicluna et al., 2006] Scicluna, J. ; Polleres, A. ; Roman, D.: *Ontology-based Choreography and Orchestration of WSMO Services*. WSMO Technical Report 2006. 2006. – URL `http://www.wsmo.org/TR/d14/v0.2/`

[Shvaiko and Euzenat, 2005] Shvaiko, Pavel ; Euzenat, Jérôme: A Survey of Schema-Based Matching Approaches. In: *J. Data Semantics IV*, 2005, pp. 146–171

[W3C, 2004] W3C: *Web Services Architecture. W3C Working Group Note*. 11 Feb 2004. – URL `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`

[Wiederhold, 1997] Wiederhold, G.: Mediators in the Architecture of Future Information Systems. In: Huhns, Michael N. (Eds.) ; Singh, Munindar P. (Eds.): *Readings in Agents*. San Francisco, CA, USA : Morgan Kaufmann, 1997, pp. 185–196